UiT The Arctic University of Norway

Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering

Visual SLAM Approach with a Low-Power Remote Robot Agent

A Practical Implementation and Proposal for Improvements

Stian Endrè Jakobsen

DTE-3900 Master's thesis in Applied Computer Science, Narvik, May 2023

"It has been discovered that C++ provides a remarkable facility for concealing the trivial details of a program — such as where its bugs are."
–David Keppel

"If you lie to the compiler, it will get its revenge."
–Henry Spencer

"Oft expectation fails, and most oft there
Where most it promises; and oft it hits
Where hope is coldest, and despair most fits."
–William Shakespeare

Abstract

This master's thesis examines the viability of using a low-performance, low-power-consumption remote robot agent to collect sensor data for use in a vSLAM system. Selections for the components of the robot agent were made based on results from a small initial study, the robot was assembled, and software to run it was created. A remote server application was created to communicate with the robot agent and receive sensor data from it, subsequently processing and using the data with ORB-SLAM3.

The robot agent successfully collects and transmits sensor data to the server application, which successfully processes and uses the data in a vSLAM approach using ORB-SLAM3. Results indicate that the approach is viable, but due to hardware limitations overall performance is lower than intended. An analysis was performed and proposals for improvements to mitigate the limitations were suggested. Suggestions for further work was given.

Acknowledgements

I would like to thank my supervisors Rune Dalmo and Børre Bang for their input, tips, and advice during the course of my thesis work.

I would also like to thank my classmates who faced the perils of higher education together with me.

Lastly, I would like to extend thanks to my family for their care, support, and limitless patience—without which it is highly likely this thesis would not have come to fruition. In particular, a huge thank you to my father for his support and aid with some of the practical aspects of the robot assembly process.

Contents

List of Figures

List of Tables

List of Abbreviations

RAM Random Access Memory

SRAM Static Random Access Memory

SLAM Simultaneous Localization and Mapping

vSLAM Visual SLAM

viSLAM Visual-Inertial SLAM

CPU Central Processing Unit

GPU Graphics Processing Unit

VR Virtual Reality

CNN Convolutional Neural Network

ORB Oriented FAST and rotated BRIEF

CV Computer Vision

OS Operating System

STL Standard Template Library (C++ library)

ROS Robot Operating System

MP Megapixel(s)

IP Internet Protocol

LwIP Lightweight IP

RTOS Real-Time Operating System

API Application Programming Interface

DHCP Dynamic Host Configuration Protocol

POSIX Portable Operating System Interface

JPEG Joint Photographic Experts Group (image file format)

1 Introduction

In this chapter we describe the background, objectives, the fundamental principles which form the basis for the project of this thesis, in addition to the results of preliminary planning work done prior to the commencement of the core thesis project work, including a description of facets at the core of this thesis as well as the current state-of-the-art. The aforementioned contents ultimately lay the foundation for the approach which is described in the last section of this chapter, which the remaining thesis work derived its approach from.

1.1 Background

During the master's course in Virtual Reality, Graphics, and Animation in the last semester before this master's thesis began, the students were acquainted with a programmable robot agent with which the students were intended to carry out a form of room mapping, the results of which would be used to create a virtual representation of the room in which the robot had moved. In this specific scenario, the students' control of the robot was limited to pre-programmed movement instructions, and as such could not be meaningfully controlled in real-time. Additionally, the sensor data—which laid the basis for the creation of the aforementioned virtual room—was cumbersome to both extract and parse owing to a lack of sensor sensitivity and communications limitations with the robot agent.

The struggles and challenges during the course of the aforementioned virtual reality (VR) project laid the foundations for an interest in better approaches to solving such a room mapping project. One idea that presented itself was the use of camera sensors to identify obstacles, as opposed to the sole motion-based collision detection of the VR project, due to a belief that these would not only provide an opportunity to prevent collisions rather than requiring them, but also allowing for other types of recognition should the need arise.

Further study into this topic lead to an accidental tumble down the rabbit hole of the Simultaneous Localization and Mapping (SLAM) problem, specifically the *visual* approach to solving the problem by using optical sensors, such as cameras. The initial idea that presented itself was the creation of a custom robot agent platform to address the shortcomings faced in the previous project, while relying on the work done in the Visual SLAM approach. The initial literature study into the topic revealed that a notable amount of the approaches described in literature relied on robot agents with significant amounts of computational power, resulting in a corresponding hefty cost of components for each respective agent, to the point of becoming prohibitively expensive in the case of a master's thesis project. Additionally, the aforementioned computational power demands proportional amounts of electricity, which in turn results in increased requirements with regard to the size of the robot chassis and motor power, owing to the size of battery packs with the required capacity. Thus, a desire to attempt to mitigate these challenges arose whence this project thesis was born.

1.2 Task description

The main core objective of this thesis was to examine the viability of solving or mitigating some of the challenges described in section 1.1 through an approach relying on low-performance, inexpensive components for the robot agent in conjunction with a sufficiently powerful remote host platform where the power balancing act of mobile platforms would not be an issue.

The original task given in the task description, which can be found in Appendix A, was divided into four parts, of which the literature study and selection of materials was part one. As such, the remaining three research and development parts were as follows

1. Creating a robot agent platform using a microcontroller and cameras, capable of communicating wirelessly with the main application host which processes data and makes decisions for SLAM

2. Creating a host application which uses image processing and/or AI techniques and frameworks to process sensor data, primarily photos, provided by the robot agent, in order to make decisions and gradually map rooms according to SLAM

3. Creating a client application which provides a visualization in real time of the results from the SLAM approach created in part 2

where the final part of creating the client application was an optional bonus task if timed allowed. Having completed the main development sub-objectives, the remaining scientific research objective was to validate the results obtained from the created applications and ascertain to what extent they were viable as a SLAM approach, with particular emphasis on any shortcomings or strengths of such an inexpensive, low-performance approach.

1.3 Theory

Given that one of the core objectives of this project was to achieve a form of SLAM with certain novel requirements, one of the biggest aspects of this project dealt with the concept of SLAM. As such, this section gives a brief introduction to the concept of SLAM and its facets that are most relevant to this thesis project.

As previously mentioned, the process of mapping a room through sensor data with a mobile agent, whilst simultaneously keeping track of the agent's position within the room, is referred to as Simultaneous Localization and Mapping; also known as SLAM. The concept of SLAM has been achieved using a multitude of different types of sensors, such as ultrasonic sensors [1], LiDAR [2], and optical cameras [3] in the form of both single-lens cameras providing monocular images and dual-lens cameras providing binocular images [4]. Given that the initial thesis task description, as seen in Appendix A, forced the usage of at least two cameras as sensors, it was predominantly the visual SLAM approach (vSLAM) that was of interest to this project, as well as visual-inertial SLAM (viSLAM) given that the main distinguishing factor between the two is the addition of an inertial measurement unit (IMU) [5] and there was an early inclination to use an IMU with the project should time allow.

SLAM as a concept can be traced at least as far back as 1995 when the acronym is claimed to have been introduced at the International Symposium on Robotics Research, with the core ideas likely being far older [5, 6]. The key problem that SLAM presents is asking whether it is possible for a mobile robot agent, placed in an unknown environment, to incrementally create a map of this

environment while simultaneously determining its localization within said environment [6].

As mentioned in Durrant-Whyte & Bailey (2006), SLAM has been a solved problem for a while, with some early practical implementations described in the aforementioned paper. Since then, improvements have been made in terms of handling environment complexity, computation, feature representation, and data association [6]. Extraction of information from the sensor devices is one of the key stages of visual SLAM, with there being two main method types to achieve this. The first, feature-based methods, rely on extracting a sufficient amount of features and matching them between successive sets of images. The second, direct methods, involve extraction of information or parameters from pixel intensity values (e.g. brightness and illumination-based cross-correlation) [7].

An issue with early visual SLAM relying on feature-based methods was coping with non-textured areas, such as corridors and rooms with little furniture, as one of the key steps of SLAM—data association—relies heavily on the ability to distinguish between features. The result of this was reduced availability of corner points, which could cause such vision-based SLAM approaches to be unstable. Tomono suggested in 2009 an approach to cope with this by detecting edge points using stereo image pair, estimating camera motion by matching with the next image pair, and applying the Iterative Closest Point (ICP) algorithm, repeating this in order to compute the camera trajectory and 3D map [8].

Some other common SLAM challenges are dealing with non-static environments (e.g. environments containing other moving actors) [9], and loop closure where the remote mapping agent has to recognize a previously-visited location and updating accordingly, ensuring consistency of the map [10]. There is also the potential issue of losing the mobile robot agent's position in the map when localization fails, which can be prevented or otherwise mitigated by implementing a recovery algorithm, or sensor fusion with the motion model thus making calculations based on sensor data. One of the more common methods is using some form of Kalman filtering, often extended Kalman filter (EKF) SLAM class of algorithms in the case of nonlinear motion models, particle filters with Monte Carlo localization [11, 12].

The solutions for these issues are incorporated into a classic visual SLAM system which can be divided into five distinct modules or parts: the sensor module, front-end module, back-end module, loop closure module, and mapping module. The sensor module is responsible for collection of data for use in the other modules, the front-end module handles tracking of image features between adjacent frames resulting in the initial motion estimation and mapping, the

back-end module handles optimization of data from the front-end in addition to further motion estimation. The loop closure module handles elimination of accumulated errors, and the mapping module reconstructs the environment based on the results from the preceding modules [13]. The actual "architecture" for such a system varies depending on which approach one uses, with the one described above being common in Graph-SLAM approaches [14].

1.3.1 Camera types

One commonality through the various approaches is the presence of camera sensors. Common camera sensors used for visual SLAM can be divided into the following categories:

- Monocular cameras are single-lens cameras that have the advantage of being low cost and simple to deal with in terms of system complexity, but can cause issues owing to the depth of landmarks being difficult to estimate, which may cause scale ambiguity for map building. Furthermore, pixel depth may not be calculated in a static state [11, 13, 14].

- Binocular cameras are dual-lens cameras, like monocular cameras, are usually low cost, but come with the added complexity and are computationally more expensive as a result of the double images being processed. There is, however, the added benefit of pixel depth calculation being possible in static states for these cameras, and have been proven to be more robust than monocular cameras [13, 15, 14].

- RGB-D cameras differ from monocular and binocular cameras in that they provide direct pixel depth through time of flight or infrared structure-light, but suffers in environments exposed to sunlight or other high-intensity lights as they are easily disturbed by such interference (an example of such a camera with either pixel depth measurement methods can be found in Microsoft's Azure Kinect v1 and v2 kits) [13, 16, 14].

- Event cameras, also known as neuromorphic cameras, differ from previously mentioned cameras in that they capture images not with a shutter at a fixed rate, but rather operates each pixel independently and asynchronously as changes in brightness occur. This allows them to exhibit better performance in high-speed, high-dynamic conditions, but generally at an increased price-point [13].

1.3.2 Sensor calibration

In most advanced computer vision tasks it is a requirement that the aforementioned camera sensors are calibrated. The reasoning being that this is a necessary step for the computer vision approaches to extract metric information from 2D images, which in turn is required in order to mitigate the phenomenon of distortion of which some camera types are more prone to than others [17]. This kind of distortion appears in two major kinds: radial and tangential distortion. Radial distortion is a type of distortion that causes curvature in straight lines, increasing the farther lines get from the centre of the image in question [18]. An example of this type of distortion can be seen in figure 1.1.

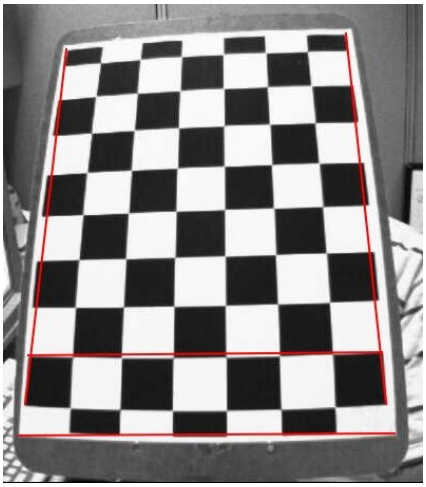

Figure 1.1: Radial distortion in chessboard image, from the OpenCV Camera Calibration documentation [18]

Tangential distortion is a type of distortion where some areas in an image may appear closer than expected, as a result of the camera lens not being perfectly aligned parallel with the imaging plane. Radial distortion can be expressed with the formula

$$\begin{aligned} x_{distorted} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{distorted} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{1.1}$$

and tangential distortion with the formula

$$\begin{aligned} x_{distorted} &= x + [2p_1 xy + p_2(r^2 + 2x^2)] \\ y_{distorted} &= y + [p_1(r^2 + 2y^2 + 2p_2 xy)] \end{aligned} \tag{1.2}$$

in the case of distortion for use in OpenCV. Of note are the distortion coefficients k_i and p_i which represent the net radial and tangential distortion respectively [18, 17].

In order to successfully mitigate the distortion, it is necessary to obtain the intrinsic and extrinsic parameters of the camera—or cameras in the case of stereo vision—where intrinsic parameters include focal length (f_x, f_y) and optical centres (c_x, c_y), which constitute the camera matrix K:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

which is unique for each specific camera. The extrinsic parameters include translation and rotation vectors which are used to translate coordinates in the image to their counterparts in the 3D real world space, and in the case of stereo cameras translating coordinates in one camera to their corresponding coordinates in the other camera [18].

Computing the intrinsic and extrinsic parameters for a set of cameras is usually achieved by using sample images containing a well-defined pattern where the geometry, points, and relative positions are already known. One commonly used pattern in computer vision applications is a chessboard. By finding specific points on the chessboard in the sample images, it is possible to use the discovered points in the image space in conjunction with their known points in real-world space to compute the distortion coefficients, intrinsic and extrinsic parameters [18, 17]. An example of the use of a chessboard pattern for the calibration process can be seen in figure 1.2

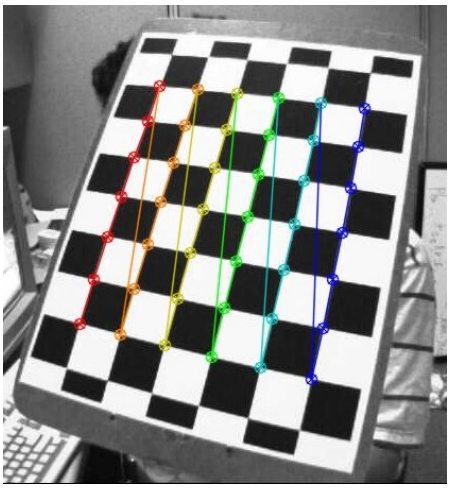

Figure 1.2: Camera calibration example with chessboard pattern drawn, from the OpenCV Camera Calibration documentation [18]

1.3.3 Algorithms

For visual SLAM there exists a sizable amount of different algorithms that have been released throughout the years, each with their own use-cases and performance. Due to the restriction of binocular cameras and real-time application, only a brief look is taken at some of the relevant algorithms for this particular use-case here, ignoring some like MonoSLAM that are limited to monocular implementations [19].

1.3.4 State-of-the-art

For the purposes of this project, there are currently three state-of-the-art algorithms for vSLAM that are of relevance: ORB-SLAM2 [3], CNN-SLAM [19], and ORB-SLAM3 which has support for both vSLAM and viSLAM approaches, given that its support for inertial sensors [20].

The ORB-SLAM2 (Oriented FAST and Rotated BRIEF) algorithm is a feature-based algorithm that is considered the state-of-the-art of this class of SLAM algorithms. It is a fairly accurate and performant algorithm, which allows for the use of monocular, binocular, as well as RGB-D approaches. One downside to this algorithm is its difficulty with recovering from a tracking failure situation unless a high similarity frame is recognized. Additionally, the method requires the collection of frames at the same rate as it processes the frames, which may result in difficulties with making real-time embedded system operation a reality. It has however been successfully implemented in embedded systems in literature [19, 3].

CNN-SLAM is a real-time SLAM algorithm using convolutional neural networks (CNN) to predict depth in its processing pipeline from collected frames, after the processing the algorithm performs a pose-graph optimization in order to obtain a globally optimized pose estimation. One benefit of this algorithm is that due to relying on depth prediction to perform scale estimation, it does not suffer from absolute scale limitation. A downside to this algorithm is the necessity of the presence of both a CPU and a GPU for real-time operation [19, 21].

Finally, the ORB-SLAM3 algorithm is an updated version of the ORB-SLAM2 algorithm with the main relevant updates over its previous version being that it is the first SLAM system able to reuse all previous information in every algorithm stage, which result in a highly robust and accurate SLAM system even compared to other state-of-the-art systems in literature, according to Campos *et al.* [20].

At least two other studies have attempted a SLAM approach using moderately

low-performance mobile agents with some success. Zhang *et al.* analysed in [22] the performance of three different SLAM algorithms with their Intel Core i5 powered agent (in conjunction with an STM32 controller), though their approach still required processing be done in the agent itself. Another approach was achieved by Karam *et al.* [23] where they examined the use of a low-cost microdrone powered by two microcontrollers streaming sensor data from the drone to a stationary host computer. This approach is possibly the closest to what this thesis is attempting, differing in the choice of sensors where Karam *et al.* [23] relied on LiDAR instead of stereoscopic cameras.

1.4 Approach

To summarize then, the main problem to be solved, or more accurately researched, was the requirement in several current SLAM approaches for computational devices with high power consumption in the mobile robot agent and the corresponding size-requirement for such an agent. This in practice involves "moving" the sensor module in a SLAM system's architecture to a remote (mobile) agent, with the rest of the system remaining stationary. As such, a research question can be formulated: *Is it possible to achieve adequate performance in a vSLAM or viSLAM approach using low-performance, low-power-consumption computational devices for sensor data collection, and offloading computing to a stationary host platform?* Additionally, the created platform's power consumption and corresponding battery life performance would be interesting to examine as well. In order to determine to what extent the results of this project answers the research question, a set of requirements for each development objective have been created.

1.4.1 Robot agent

The requirements for the robot agent can be summarized as follows:

- The agent must be capable of receiving instructions through Wi-Fi from a remote application.
- The agent must be capable of transmitting sensor data through Wi-Fi to a remote application.
- The agent must be capable of executing movement orders received through Wi-Fi from a remote application.
- The agent must be power-efficient enough to run for at least 1-2 hours

with continuous use.

1.4.2 Host platform for processing and decision-making

The requirements for the remote host platform can be summarized as follows:

- The platform must be capable of sending instructions through the network to the robot agent.
- The platform must be capable of receiving sensor data through the network from the robot agent.
- The platform must be capable of using received sensor data with a SLAM algorithm.
- The platform must be capable of using the SLAM results to compute movement orders to be transmitted to the robot agent.
- The platform must be capable of transmitting movement orders to the robot agent.
- The platform must be sufficiently computationally powerful enough to perform SLAM in real-time.

1.4.3 SLAM system

It was not expected that the created system would outperform that of contemporary state-of-the-art approaches, the only point of interest in this project was achieving a form of vSLAM or viSLAM with remote parsing. The minimum ideal milestones for the overall project/SLAM system can be summarized as follows:

- The system should be capable of yielding some results from SLAM in a small, empty room. (less than $10m^2$)
- The system should be capable of yielding some results from SLAM in a small room with sparse-density obstacles.

Capability successfully completing these two milestones to some extent—in addition to the requirements for each sub-part of the system—would signify achievement of some level of success expected for this project.

2 Method

2.1 Choice of tools and materials

As part of the preliminary thesis work, research was done to ascertain which tools, hardware, and software would likely best suit the task at hand. The research and ensuing selection of materials was based on a number of various factors, such as ease of procurement, performance, but also the author's experience. This section contains the result of this preliminary analysis, in addition to some changes made to the initial selections in the course of the project development period.

2.1.1 Experience

This subsection lists the author's experience with the relevant tools, libraries, and languages that laid the foundation for the initial choices being made, in addition to the formal educational courses with relevant curriculum to the subject.

Courses

- DTE-3605 - Virtual reality, Graphics and Animation - project [24]
- DTE-3609 - Virtual reality, Graphics and Animation - theory [25]

- DTE-3606 - Artificial Intelligence and Intelligent Agents- project [26]
- DTE-3608 - Artificial Intelligence and Machine Learning - theory [27]

Programming languages

- C++(11-20)
- Rust
- Python

Libraries

- OpenGL
- Boost C++ library
- Blaze C++ library
- TensorFlow library
- Scikit library
- Qt6

Tools

- Visual Studio Code
- Unreal Engine
- Qt editor

2.1.2 Selections

Hardware

In the state-of-the-art analysis it was discovered that one of the algorithms required at minimum a quad-core ARM processor in order to run, with another requiring a processor and GPU combination. From this it was deduced that the remote control application would likely require processing power comparable or in excess of this. As such, the author elected to rely on an Intel Core i9-12900k 16-core system with 32GBs of DDR5 6000MHz RAM, 4TBs of high-speed SSD storage, and an Nvidia RTX 3090 GPU running Proxmox [28] as a hypervisor for the remote host platform. The reasoning for this choice being that it is sufficiently powerful to power any SLAM algorithm that may be reasonably

chosen to employ, removing the processing power of the remote host application from the equation, and does not require procurement for the project as it was already in the ownership of this author.

Given this, the only hardware needed to be acquired was that of the mobile robot agent itself. The robot agent's required hardware components could essentially be reduced to sensors, processing units, motors, power supply, and chassis. Given that the problem description specified the use of at least one microcontroller—and the overall goal of this project—the author was limited to relying on microcontrollers as the remote agent's processing unit. For sensors, the specification outlines at minimum two cameras.

Microcontroller

As previously mentioned, the problem description specifies that the robot agent must use a microcontroller. With this in mind, a search for some popular microcontrollers easily available for purchase in Norway as of early-February 2023 was done.

SKU	µcontr.	Processor	Memory	I/O	Wireless
Arduino Uno	ATmega328	8-bit 1-core RISC	1 KB EEPROM, 2 KB SRAM, 16 KB flash	23GPIO	No
Raspberry Pi Pico W	RP2040	32-bit 2-core RISC	264 KB SRAM, 2 MB flash	26GPIO	Yes
Arduino Micro	ATmega32u4	8-bit 1-core RISC	1 KB EEPROM, 2.5 KB SRAM, 32 KB flash	20GPIO	No
Arduino Nano	ATmega328	8-bit 1-core RISC	1 KB EEPROM, 2 KB SRAM, 32 KB flash	14GPIO	No

Table 2.1: Specifications for selection of microcontrollers in stock in Norway

Searches through two of Norway's biggest webshop aggregator sites—Prisjakt and Prisguiden [29, 30]—revealed that the great chip shortage was very much still in effect as of early 2023 [31] and did indeed impact microcontroller availability as well. In table 2.1 specifications of some of the more readily available microcontrollers present in Norwegian online retailers, as found through the aforementioned aggregator websites, are presented. From this selection it was noted that only the Raspberry Pi Pico W has wireless functionality out-of-the-box. Furthermore, for the remainder of the entries, the Pico W retails for roughly 149NOK whilst the Arduinos retails for between 330-350NOK; a great deal more than the Pico W.

It should, however, be noted that some of the Arduino chips are available with wireless functionality at around US$20 through the Arduino webshop, though availability is still lacklustre compared to their Raspberry Pi counterparts. Including shipping and import tax, the Arduino store price can be estimated to approach that of the Norwegian webshops regardless. As such, the Raspberry Pi Pico W was selected for this project due to its much lower price, generally

higher performance, and built-in wireless functionality. Another option—which does not seem to be readily available in Norwegian webshops—are ESP32-based microcontrollers offered by Espressif which offer comparable or better performance than the Pico W at a similar price-point.

Sensors

With regard to sensors, this project required at minimum two cameras (or a single stereoscopic camera). Furthermore, the cameras needed to be able to easily interface with the selected microcontroller; ideally without requiring extensive driver software. Unfortunately—as of early-February 2023—this author could not find any evidence of the Raspberry Pi foundation releasing any official camera hardware for the Pico (nor from Arduino for that matter). As such, it was necessary to search for third party alternatives.

Searches through the same aggregators as in chapter 2.1.2 revealed a single type of camera available for the Pico through a webshop in Norway, but at the price of 699NOK per camera, far in excess of what the author intended for an inexpensive stereo approach given that two of these cameras would be around 1400NOK for the cameras alone. This specific camera was a 2MP camera with an OV2640 sensor that connects to microcontrollers through the serial peripheral interface (SPI) and inter-integrated circuit (I^2C) to facilitate data transfer, camera control, and sensor calibration [32]. Some further queries revealed that while a stereoscopic camera does exist for use with Raspberry Pi products, they are seemingly limited to the full versions such as the Raspberry Pi 3 and 4 [33], both of which are full general purpose computers—not microcontrollers.

Subsequent searches for the OV2640 revealed that it was available at a cheaper price point through the same website as the aforementioned stereoscopic camera, in addition to a 5MP model—the OV5642—with a slight price increase. The bump in resolution from 2MP to 5MP was something worth considering as the option of higher resolution of the input data set could provide an opportunity to evaluate whether image resolution can be tied to performance. A possible alternative from the same manufacturer was a monochromatic camera sensor which output QVGA resolution images, that is to say a quarter of VGA resolution, at roughly half the price of the 5MP variant. However, selection of the 5MP variant would allow for experimentation with higher resolutions including QVGA, in addition to comparing the results of using greyscale images versus RGB images in vSLAM. As such, two units of the 5MP OV5642 SPI camera were selected as the visual sensors for use in the robot agent, owing to a combination of availability, price, lack of other options, as well as to facilitate the aforementioned experiments.

2.1.3 Motors, power, and chassis

The remaining parts required to construct the robot agent are motors to facilitate mobility, a motor driver module, parts to construct the robot's chassis, and a power supply of some sort. For the motors, some cheap 4.5V-9V DC motors were selected as they were already available without procurement, fit the microcontroller's rated power output, and were thought to provide the necessary mobility when connected to wheels. Likewise, for the driver module a board fitted with a L298N H-bridge motor driver chip was initially selected, but later replaced with a TB6612-based H-bridge motor driver chip due to the latter's lower power draw of ≈0.1V compared to the L298N's ≈1.4V. The robot's chassis was to be constructed with leftover Lego bricks (in combination with some drilling and glue) to simplify—and speed up—prototyping and construction. An alternative could have been to create a 3D model and 3D-print the chassis, but that was not part of the problem description and was as such not prioritized unless it proved absolutely necessary. Lastly, as the Pico only requires between 1.8V and 5.5V with a maximum power draw of 93mA [34], a series of AA batteries were initially considered to supply power to the robot, but after calculating the overall power draw of the various components, it was decided that the use of a 26800mAh USB powerbank would be the better option as it was already available and more than sufficient, being capable of delivering 5V up to 2.5A.

Software

Part of the task described in the problem description was to select programming languages, frameworks, libraries, and other tools required to create the software-side of the project, with no specific requirements having been given. As such, selections were done based on the author's previous experience with the intention to facilitate an as smooth as possible development process while ensuring that it was highly likely the selections would meet the necessary requirements of the task. The overall programming tool of choice was the Visual Studio Code (VSCode) code editor [35], as it was lightweight, multipurpose, and the author had extensive experience with it.

Robot

For the task related to the programming of the robot agent and communication with the remote application, we have chosen to rely on the Pico SDK, which is the software development kit created for use with the Pico platform by the Raspberry Pi foundation [36], and as such the programming language for this part is C++. Additionally, we decided to use the FreeRTOS kernel

as it allowed for use of LwIP's socket API for TCP communication, greatly simplifying the work required for communication between the robot and remote application [37]. Furthermore, the kernel provided a safe and easy-to-use multithreading API which was thought to likely boost the overall performance of the robot agent if implemented correctly [38]. As the robot runs the FreeRTOS kernel, an option to easily communicate with the server application for SLAM purposes could be to implement and use the open-source robot operating system (ROS) [39], specifically the microcontroller port for use with RTOS variants—micro-ROS [40]. This option was initially considered, but there was a concern regarding whether such a move would overly simplify the development process as well as introduce another potential point of failure as the port for the Raspberry Pi Pico W platform was still relatively new at the time. Given these concerns, in addition to some concerns regarding integration with the camera drivers, it was decided to forego use of micro-ROS and develop the communication system from scratch.

Remote host platform

The remote application receives sensor data from the robot through wireless, subsequently processing the data with a visual SLAM algorithm that is then used to create a 3D or 2D map with the robot's location. The application then transmits new movement orders to the robot, and the process repeats. In order to facilitate this functionality, the application requires software that allows it to communicate through either Wi-Fi or Bluetooth, apply some visual SLAM algorithm, and map the room in 3D or 2D (possibly as part of the SLAM algorithm). This could be achievable through manual implementation of SLAM algorithms for example in consort with Point Cloud Library [41] and OpenCV [42] on the remote application side, and through the use of the aforementioned ROS [39], specifically micro-ROS [40], on the microcontroller-side. The selection for this application was delayed until after the robot had been assembled, as some trial and error was needed in order to determine whether the use of ROS was possible or even desirable for this implementation. Ultimately, as use of ROS was decided against in the robot agent, this became the case for the remote host platform as well.

As for the selected SLAM algorithm, we elected to use ORB-SLAM3 library owing to its decent performance and depth of detail in documentation regarding its implementation, in addition to support for both vSLAM and viSLAm, as well as monocular and stereoscopic variants of either approach should time allow [20]. Additionally, in choosing the ORB-SLAM3 library we were able to avoid the arduous process of manually implementing a SLAM algorithm ourselves, which would be beyond the scope of this project.

As it was decided to run the host application on the hardware specified in section 2.1.2 with Proxmox as a hypervisor, we were free to pick whichever host OS we desire, provided it could run the aforementioned SLAM algorithm libraries. Supporting every OS in existence would not be possible, and given the high likelihood of having to write platform-specific socket code in order to interface with the robot agent, it was—based on previous experience—desirable to pick one OS and stick with it. As such, Linux was selected as the host OS for the remote application, due to it being lightweight, the author having extensive experience with it, and—from personal experience—the ease with which one may create virtual machines or Linux containers in Proxmox.

Client application

The client application was initially intended to communicate with the remote application, and function as a front-end for the end user by visualizing the mapping done in addition to possibly providing some rudimentary user control over the robot if possible. Due to previous experience with Qt [43], a framework and tools used for creating graphical user interfaces, it initially decided to create a desktop application in Qt with some rudimentary visualization being displayed through OpenGL [44], a graphics API, in C++. However, during the course of the project it quickly became clear that there would be no available time to do such a thing, and the user interface provided by the ORB-SLAM3 algorithm would be adequate for the purposes of this thesis. As such, the client application subtask was in large part omitted or otherwise merged with the remote host application task.

2.2 Methods

In this section, we describe in detail the implementations made during the development process for each respective subtask.

2.2.1 System architecture

The overall architecture of the system consists of a mobile robot agent, which gathers and transmits sensor data by moving through a room, and a remote host platform which receives the information, parses it, and sends new movement orders back to the robot agent. In the subsequent sections we take a look at the internal architecture of the overall SLAM system and its constituent subparts.

Picobot

The robot agent, nicknamed Picobot in development, consists of a Raspberry Pi Pico W microcontroller, two OV5642 SPI camera sensors, two DC motors, a TB6612 H-bridge motor driver, and a power supply in the form of a 26800mAh battery powerbank. Picobot runs this project's custom software powered by the Pico SDK [36] and the FreeRTOS kernel [38], which is a free and open-source operating system for microcontrollers that has been ported to the Raspberry Pi Pico W. A wiring-diagram for the Picobot can be found in Appendix B. The internal software is multithreaded and module-based, where a main function launches separate RTOS tasks for each module's functionality as needed or otherwise dictated by instructions from the remote server application. On startup, it launches a main task thread which initializes and connects to a Wi-Fi network specified in a configuration file during the build process (more details in Appendix B), after which, depending on the success of five such connection attempts, it launches an internal TCP server to which the remote server application is expected to connect.

The robot outputs its DHCP-assigned local IP address through serial (UART), which the remote server application requires in order to facilitate a connection. As the robot is not fitted with a display, the user is required to connect to it with a serial interface, unless some changes are made to output serial through USB (see Appendix B), or alternatively looking up the assigned IP in the Wi-Fi network's router interface for the Picobot's assigned hostname.

After the initial connection with the remote server application has been established, it is possible to issue orders through the server application's interface which are parsed by the robot's TCP server module, and issued to other waiting tasks in the system. Movement orders are for instance issued to the idling movement module thread, which parses and executes movement orders in a specific format (see Appendix B) before returning to idle until more orders are received and forwarded to it. The robot runs a TCP client in a separate task for transmitting sensor data back to the server application, in order to facilitate being able to receive orders simultaneously while transmitting data. The TCP client connects to an IP address and port number specified in an order received from the server application, where a successful connection is a requirement for capturing and transmitting sensor data such as photos captured with the cameras.

The cameras are handled by a module running in yet another separate task that interfaces with a driver supplied by the manufacturer [45], which was subjected to some minor modifications in order to run the cameras concurrently on separate SPI and I^2C interfaces. This module provides single-capture or multi-capture capability, meaning that it is possible to capture single-frame

images and transmit them. In the case of single-capture, it is possible to specify resolutions up to 5MP, though that incurs a time penalty in terms of both capture and transmission speed. It is not possible to specify resolution through commands from the server application, as that feature was not required for this project. Regarding multi-capture, there are a number of different options presented to the server application, though resolution is capped at 320x240 pixels. The user is able to select between capturing n-amount of pictures, with an additional internal option for how the camera captures these, or continuous capture until a stop signal is sent from the server application. Multi-capture is achieved with two different approaches: first by continuous capture of single images, or by capturing a certain number of frames continuously until either the specified amount is reached or the aforementioned stop signal is issued. The reasoning behind this design choice is discussed in detail in a later section, but the difference is a noticeable amount of delay present in the first option that is somewhat reduced when capturing several frames at a time before reading and transmitting them to the server application.

PicobotServer

PicobotServer, the development nickname for the remote host platform application, is a Linux-based multithreaded application that follows a similar design pattern as the robot agent, but with modern C++20 and POSIX [46] APIs instead of RTOS APIs. Similarly to the robot agent, it consists of a number of modules for the various functionalities being launched from a main thread as needed. TCP server and client functionality was implemented with the POSIX socket API, facilitating communication between the remote agent and server application. On launch, the user is presented with a command line interface (CLI) wherein they can launch the various components of the system and issue orders (using pre-defined functions) for the remote agent. These components run in separate threads and communicate through common C++ STL data structures in conjunction with concurrency features such as condition variables, mutexes, and semaphores to facilitate thread safety and avoid race conditions [47, 48].

In order to communicate with the remote robot agent, the user is required to provide a connection function with the target robot's IP address, as mentioned in the previous section. After this connection is established a connection order may be issued to the robot agent, ordering it to connect to the (server specified) IP where it transmits any sensor data it collects following any capture orders. When a capture order has been issued, the server threads responsible for receiving sensor data through TCP sockets collects and forwards the received data through a blocking queue to worker threads which manage writer threads responsible for writing sensor data concurrently to the filesystem for further

consumption in the application.

The written data is then available to the rectification module that undistorts and rectifies images with OpenCV features based on intrinsic and extrinsic calibration performed on each robot's unique set of cameras through the program's calibration module. These undistorted and rectified images are subsequently written to another directory for consumption by the ORB-SLAM3 system. The camera calibration process is performed as described in section 1.3.2 using OpenCV camera calibration features for monocular and stereo cameras [18]. An example of a step in this calibration with a low resolution image taken by Picobot with the chessboard pattern drawn can be seen in figure 2.1.

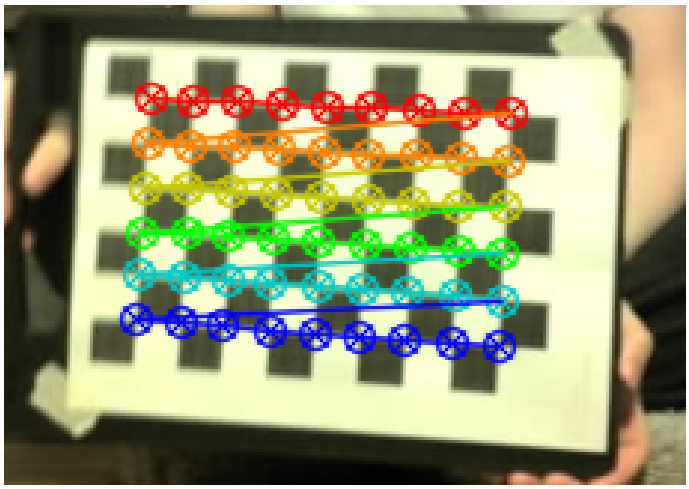

Figure 2.1: Picobot: low-resolution chessboard pattern calibration example

2.2.2 Camera data

The choice of hardware for the microcontroller presented some unique challenges when camera functionality was implemented and tested. Owing to the Pico W's SRAM size of 264kB, it became clear that storing and transmitting entire images of 5MP resolution, even with the maximum JPEG compression afforded by the camera's internal chip, would be an impossibility. Testing proved that a single such image would at minimum hover around 600kB in size, more than twice the amount of SRAM available. A workaround was devised where images would be gradually read in chunks that fit in the Pico W's SRAM, and subsequently transmitted to the server application before continuing with the read operations. This workaround allowed for capture of even 5MP still images, though for the sake of attaining the highest frame rates possible for the SLAM approach it would be necessary to downgrade the resolution significantly, both in part due to network conditions, but also due to limitations with the camera

sensors. As alluded to in a previous section, there are two possible approaches to multi-capture with these camera sensors; single-frame multi-capture, where a single frame is captured and transmitted before repeating the process, or multi-frame multi-capture, where multiple frames are captured and transmitted every capture before repeating. This is achieved by writing the number of frames to be captured in each capture operation to the camera chip's registry before capture operations begin (more details in Appendix C), reducing the time needed for capturing multiple frames consecutively. The camera then captures the specified amount of images and stores them in its framebuffer, which is subsequently read by the robot agent and transmitted, resulting in image sequences being transmitted rather than individual images; each image sequence then containing n-amount of JPEG-encoded images. For this type of transmission it is thus necessary to parse the image sequence and extract each individual JPEG before writing them to file, which is handled by a function in the remote server application. The function scans the image byte sequence for the JPEG magic bytes header [*FF D9 FF E0*], which may differ slightly between the various JPEG file types [49, 50], and as such is only confirmed to work with these particular camera's JPEG format.

2.2.3 Network communication

As mentioned in previous sections, communication between the robot and server is done through TCP with orders as strings (transmitted as bytes from the network stack), and images as byte sequences. Order messages consist of sets of bytes, with each set representing specific parts of the order message as shown in table 2.2.

0	1	2	3	
Type	Order	n-frames		} Order message

Table 2.2: Order message byte structure

For image transmissions, a metadata message is (usually) transmitted ahead of the payload sequences. An example of the metadata message structure for a multi-capture sequence can be seen in table 2.3.

0	1	2	3	
Type	CamId			} 2-character parts
Timestamp start				} Variable length segments
Timestamp end				
Sequence length				

Table 2.3: Image sequence metadata structure

The key difference between the metadata for single images and image sequences is the addition of the 'timestamp end' parameter. The timestamp parameters present in image sequence transmission metadata messages are used to interpolate timestamps for each of the n-amount of JPEG images present in the sequence, giving a rough (though not wholly accurate) timestamp for each respective image in the sequence to be used for SLAM. As this is not necessary for single frame captures, it is omitted from those metadata messages.

Following each metadata message are several data messages containing the byte segments of either individual images or image sequences, whose total length correspond to the sequence or image length parameter in the obtained metadata message.

2.2.4 Visual SLAM and Rectification

For vSLAM, we created a module to interface with the ORB-SLAM3 library [20]. This module checks for images in specific directories for either camera where the previously mentioned undistortion and rectification module has saved the processed images, and continuously provides them to the SLAM system for tracking. This module runs in a separate thread for the duration of the SLAM system's lifespan, grabbing image pairs as they arrive from the remote robot agent and remaps them based on undistortion and rectification maps created by using the intrinsic and extrinsic camera parameters previously obtained during camera calibration. These remapped images are then continuously saved to the aforementioned directory the ORB-SLAM3 module looks for images in.

As we are using pre-rectified images, the module is largely inspired by an example provided by the ORB-SLAM3 creators for the Kitti datasets [51], though with several changes made for the purposes of reading continuously received and processed images. The SLAM system will keep running until either a stop command for the system is issued in the user interface, or a certain amount of time has passed without suitable images being fed to the system, after which it will gracefully shut down all SLAM threads and save the data up to that point in a specified directory.

3 Results

This chapter describes the results from various benchmarks, tests, and experiments performed with the previously described subparts of the project.

3.1 Robot performance

The measurements of performance of the remote robot agent can be divided into four major categories representing the features and expectations of the robot as per the initial problem description: how well it captures images, how well it transmits the images, how well it receives and handles orders such as movement orders, and its general power consumption. Photos of the final design of the remote robot agent can be seen in figure 3.1.

3.1.1 Cameras

In accordance with the initial requirements, the robot is capable of capturing images for transmission to the remote application. The question is how well it performs at this task. In order to determine how well the robot captures images, it is necessary to determine some metrics by which it is measured against. Higher frame-rates, as in higher capture rates of frames per second, increase tracking of motion and lead to less motion blur, and it is as such desirable to reach a certain threshold for frames per second (fps) in order to

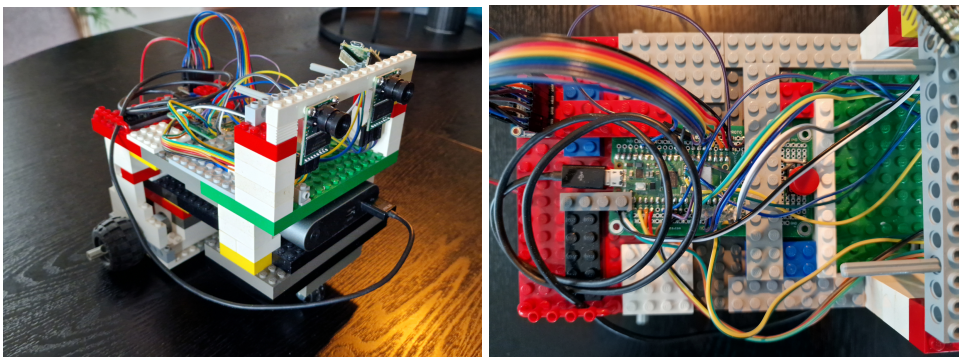

Figure 3.1: Picobot final iteration front and overhead view

facilitate higher quality data from a robot in motion in real-time [52]. The exact ideal amount for ORB-SLAM3 is not specified, but the data shown in the research paper indicate a span between 10 and 60 fps would be ideal [20]. The OV5642 camera sensors we selected for this task are advertised as capable of frame rates up to 120 fps at QVGA resolutions (320x240) [53], but no such claims were made for the shield board which the sensor is connected to. As such, we performed a number of benchmarks to determine the maximum fps the camera board was capable of at the selected resolution of 320x240. Some of the results from the QVGA resolution benchmark of the latest Picobot iteration can be seen in table 3.1, with benchmarks for the higher resolutions showing a mostly consistent doubling in average time per image with every doubling of resolution. As such, with the selected resolution we can expect around 5 frames per second on average from the cameras themselves with shorter bursts.

Resolution	N-images per capture	N-images total	Avg. seconds per image	Total time	Avg. fps
320x240	1	5	0.27s	1.3458s	≈3.715
320x240	5	5	0.19s	0.95s	≈5.263
320x240	1	15	0.2695s	4.03734s	≈3.715
320x240	5	15	0.18833s	2.85s	≈5.309
320x240	1	60	0.315s	17.711s	≈3.387
320x240	5	60	0.215s	12.921s	≈4.643

Table 3.1: ArduCAM OV5642 fps benchmark results

3.1.2 Networking

Overall, the robot agent is capable of receiving instructions through Wi-Fi from the remote application, and transmitting sensor data back to the application as it is acquired. This subsection examines how well it achieves these tasks. One metric for such performance is the rate at which captured images can be

transferred in segments from the robot agent to the server application, as that has an additional impact on the perceived fps for the SLAM system. In order to determine the transfer rate, a two-part benchmark was performed where a single large image was transferred, and the time from transfer start to when the image was fully written to file in the receiving system was measured. The benchmark executed a continuous capture and transmission of small image files to measure throughput. Excerpts from the results from the benchmark can be seen in table 3.2.

Resolution	N-images	Avg. seconds per image	Total time	Avg. image size (kB)	Est. kbps
2560x1920	1	4.753s	4.753s	853KiB	1435.724
320x240	3015	0.597s	1800s	15.1KiB	202.345

Table 3.2: Picobot network benchmark results

3.1.3 Movement

The robot is capable of receiving movement orders from the remote server application and can move reasonably well, though at relatively low speeds and with notable minor shaking during movement owing to the overall construction and relatively low-power of the DC motors.

3.1.4 Power consumption

While no extensive power consumption measurements were taken during SLAM, we can extrapolate the general power consumption from some observations made during development and during the course of testing the other functionalities of the robot. The power supply of the robot is a 26800mAh USB power bank, and during the course of a six hour testing session where cameras were used continuously while motors were used sporadically, power levels dropped from full charge to roughly 56 percent charge. Assuming that the power bank's capacity is exactly 26800mAh, that would equate to a usage of roughly 11792mAh, 1965mA per hour, which is well within the desired range of a few hours running time specified in the initial task description.

3.2 Remote host application

Similarly to the robot agent, there were a number of requirements that needed to be met in terms of functionality for the remote host application; ranging from networking functionality to SLAM functionality. In this section we examine the results of the remote host application both in terms of functionality as well

as some rudimentary performance metrics. First off, there was a requirement that the remote application must be capable of transmitting orders to and receiving sensor data from the remote robot agent. This has been successfully achieved. Furthermore, the application must be capable of using the received sensor data with a SLAM algorithm. In order to do so, it was, as mentioned in a previous section, necessary to parse, rectify, and undistort the received image data before passing it off to the SLAM system. The system successfully parses the received image data into JPEG files with timestamps, interpolated in the case of image sequences, which are then made available to the rectification subsystem.

3.2.1 Rectification

The undistort and rectification system works as intended, as can be seen in figure 3.2 where two distorted images from each camera (on top) are layered above two undistorted and rectified images with red lines through each, highlighting the effects of the rectification and undistortion.

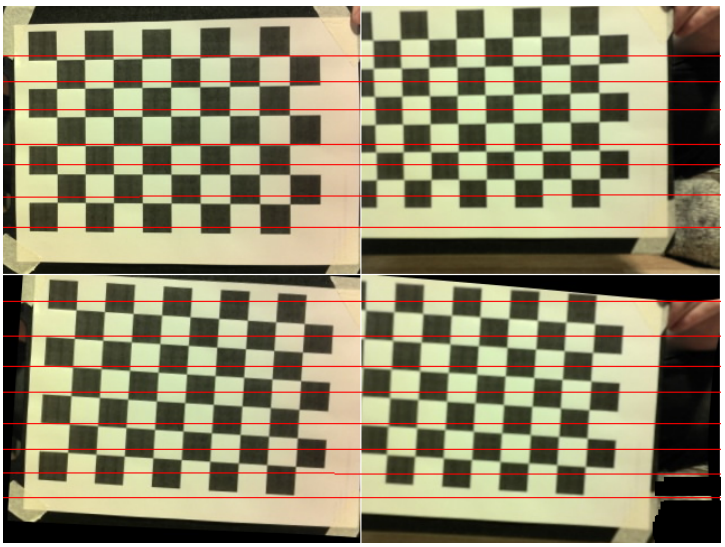

Figure 3.2: Distorted and undistorted images. Top: distorted pre-rectification images. Bottom: undistorted rectified images.

Benchmarks of the system indicates that it can undistort and rectify in excess of 3000 image files in roughly 16 seconds, which is about 187.5 frames per second and thus far in excess of both what the previously mentioned camera fps results indicate as well as what is desired.

3.2.2 vSLAM

The SLAM system is capable of using the received sensor data from the robot agent to perform some form of SLAM, with varying results, though it is not capable of using the results from the SLAM to compute new movement orders for the remote agent, as that functionality has not yet been implemented. Furthermore, SLAM performance and results are unstable and erratic as the system frequently loses tracking and creates new maps when it finds a new valid key frame. As the results in Figure 3.3 indicates, however, it is indeed capable of using the images from the robot agent to some degree, extracting features from the images. Tests of colour and greyscale images with the same environment parameters yielded no discernible differences in results.

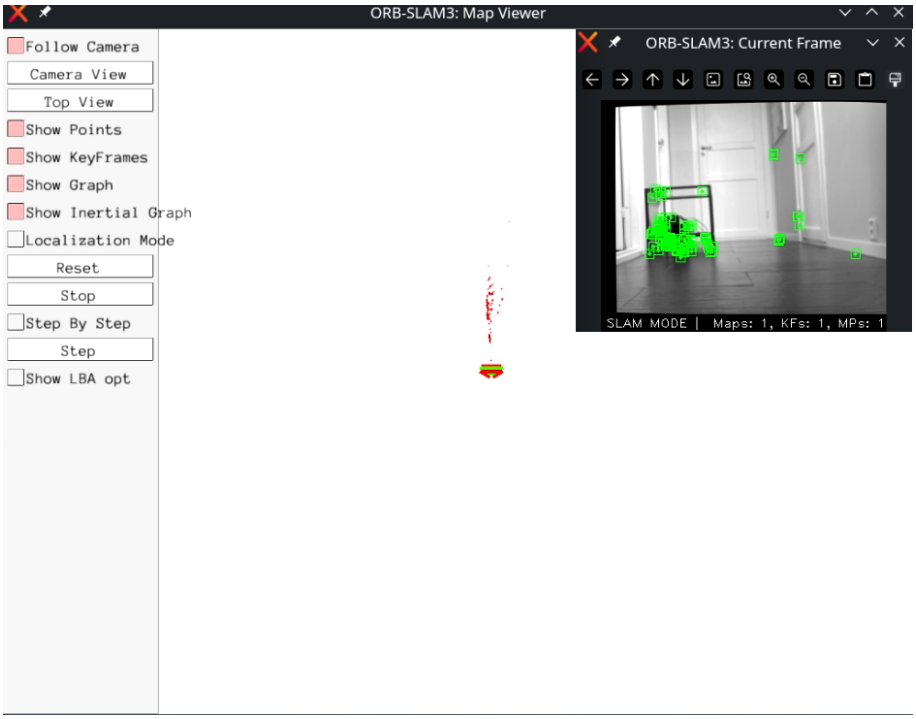

Figure 3.3: ORB-SLAM3 with rectified Picobot images initial results screenshot

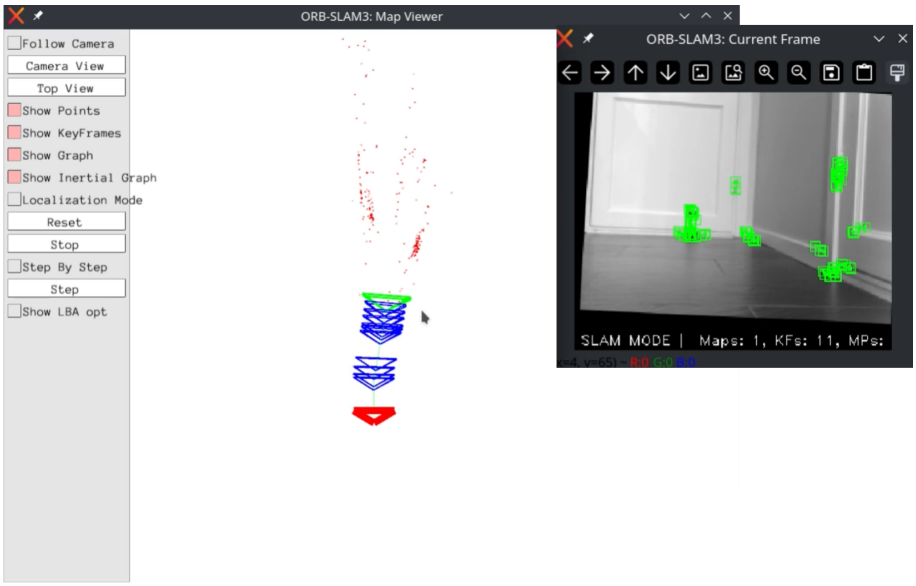

Figure 3.4: ORB-SLAM3 small corridor test results screenshot

Additional testing yielded some results for path tracking, as indicated in Figure 3.4, with the system being able to track the rudimentary shape of a small corridor as indicated by the created point cloud. The system is also capable to some extent in certain circumstances of tracking paths into adjoining rooms, as displayed in Figure 3.5.

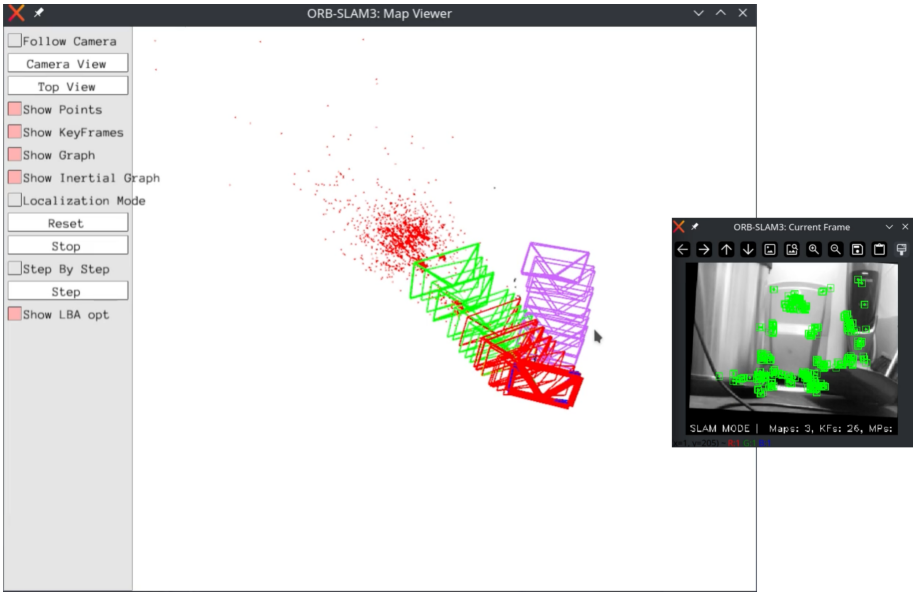

Figure 3.5: ORB-SLAM3 adjoining room test results screenshot

4 Discussion

This chapter contains an analysis and discussion pertaining to the results and methods from the previous sections, as well as the overall problem completion for the thesis project.

4.1 Picobot

Development of the Picobot went through several iterations and was as such easily the most time consuming aspect of this project, particularly due to the nature of a set of issues that were discovered roughly halfway through the development process. These issues are detailed in the respective subsequent subsections herein.

Cameras

As seen in the camera section of the results chapter, camera functionality was fully implemented with Picobot being able to capture and transmit images to the full extent of the camera hardware's capabilities; ranging from the lowest resolution to the full 5MP resolution. Limitations were however found with regard to how quickly images could both be captured, and the rate at which they could be transmitted from the camera's internal framebuffer to Picobot's microcontroller. A large portion of the development stage was spent attempting

to diagnose and optimize this performance to eke out as much performance as possible from the camera stack, with the final iteration barely managing an average 5 frames per second at the lowest possible resolution for the sensor. The reason for this limitation has been analysed and identified to be a limitation of the manufacturer's board chip. As was briefly mentioned in the results, the OV5642 camera sensor itself is capable of 15 fps at 5MP resolution, 30 fps at 1080p, 60 fps at 720p down to VGA resolutions, and finally 120 fps at QVGA resolution [53]. However, the camera board/shield from ArduCAM with the camera sensor attached does not advertise its possible frames per second for the various resolutions. The actual capabilities of the chip are unknown, but as the SPI speed is advertised to be 8MHz maximum, it is possible to calculate that even at maximum transfer speed, the interface would likely top out at a 1-2 5MP images per second in the best case scenario where 1MHz is 1mbps transfer speed. This seems to closely match the discoveries shown in the results section.

Networking

Rudimentary benchmarks for network speed and throughput were shown in the results' section to give an indication of the rough image transfer throughput capabilities of Picobot, but it should be noted that due to the nature of the manner in which images are transmitted from the robot to the remote agent these results are not entirely indicative of the actual network capabilities of the device. Specifically, in the case of the continuous capture benchmark results particularly, there is additional delay introduced by the gradual read of data from the SPI interface before transmission of each image's individual chunks. The tests could likely be further refined by measuring and removing the read start and end of each chunk from each respective image transfer time, but for the purposes of this thesis it was sufficient to demonstrate the overall image transfer rates for continuous capture in a real world scenario. Furthermore, the overall system occasionally exhibits hiccups and several second-long delays every so often, which further skews the average transfer rate for continuous capture; single capture transfer benchmarks are impacted to a lesser extent owing to only a single capture command being issued in the benchmark.

Movement

Overall the movement system of Picobot works as intended, though it was discovered during testing of the actual SLAM system that the speed and torque of the selected motors were slightly problematic due to the frame rate limitation of the camera system, where it would be desirable to move the robot at a slower speed than was possible with the original motors. An attempted solution was

the use of motors with much greater torque at lower rotations per minute, but availability and shipping delays prevented these from being put in extensive use before the thesis deadline. As such, some of the test results in the results' section were obtained through the use of human assistance, though this is unlikely to have a profound effect on the overall results beyond being able to accumulate the necessary data.

Power consumption

The power consumption of the Picobot was as expected and intended, as indicated in the results where we found through anecdotal evidence that the overall system likely averaged around 1.965A per hour with normal use, equating to several hours of running time with the intended power supply using the original motors. Some rough calculations on the estimated power consumption of each respective component added together yields a similar number with some expected variation that may be attributed to manufacturing tolerances and variation in power supply actual capacity versus expected capacity. That being said, proper power consumption measurements would have to be taken in order to with absolute certainty determine the actual consumption of the device. For the purposes of this thesis, however, it was decided that the aforementioned method was sufficient, as the performance demonstrated the capabilities of the system.

4.2 PicobotServer

Development of the server application was a mostly straight-forward process with little delay, with the most challenging aspects being the creation of the calibration and rectification systems, and the ORB-SLAM3 module which interfaces with the ORB-SLAM3 library. Overall the system achieved its intended purpose by demonstrating the possibility of using the collected sensor data from Picobot. An exception would be the missing movement plotting feature which was intended to extract information from the ORB-SLAM3 system and provide updated movement orders to the robot agent as described in previous chapters. An outline for how this feature could be implemented is included in a later chapter of this paper.

Rectification

The rectification system appears to function properly as indicated by the results in Figure 3.2 despite the relatively low resolution. There were some initial

issues with the undistortion and rectification process that were eventually diagnosed as being caused by two separate mistakes. The first being an accidental swapping of cameras on the robot itself, where the expected left camera sensor data actually was the right camera sensor data, and vice versa. This lead to the undistort and rectification process transforming the images with a basis in wrong camera poses, leading to weird and unpredictable calibration outputs where often the image was entirely black. Swapping of the camera pins on the robot completely solved this issue.

The second issue was an issue stemming from the checkerboard pattern on the first chessboard used for calibration where the algorithm was seemingly incapable of detecting the pattern's corners. The exact reason for this error is unclear, but may be due to the internal workings of the OpenCV corner detection algorithm expecting each respective checker row to end with the opposite colour that it started with, which the initial chessboard did not. While not completely conclusive evidence, this seems the most likely explanation owing to the aforementioned pattern difference being the primary discernible difference between the two chessboard patterns as they were of similar dimensions and square sizes. As such, the issue was solved by swapping the chessboard in favour of one where this pattern was followed. Overall, the system yielded the expected results which allowed for use of the stereo cameras with the ORB-SLAM3 algorithm as intended.

vSLAM

As seen in the vSLAM results, the system is capable of using the collected and processed camera data to perform SLAM with the ORB-SLAM3 library to some extent. While the system is capable of tracking the movement path—and extracting features in order to map the real world by creating point clouds—it is in its current state not capable of fully mapping a room and the robot's path through said room in most circumstances. As mentioned, it frequently loses tracking and struggles to extract features in particularly bare environments, often resetting and creating a new local map or outright crashing when this happens. The reasons for this behaviour likely stem from a combination of the shortcomings mentioned in previous sections, in addition to a need for further fine-tuning of parameters specific for the ORB algorithm.

The combination of low resolution images, the rate at which they are captured, their transmission rate, and further delays, are thought to be some of the main factors impacting the SLAM performance of the system in its current iteration. It is known that ORB systems are particularly sensitive to variations in time for sensor data, such as the time between captured frames, which impact performance in a real-time application of this form of SLAM given that

it prefers a collection of frames at the same rate at which it processes said frames [3, 19]. Inspection of timestamps for images being processed at the time of tracking failure give an indication of a correlation between greater than usual delays in capture and processing, and some of the tracking failures. Capture and transmission of images from Picobot occasionally exhibit several second long stutters which may likely be attributed to variations in network latency, internal system lag in the robot's microcontroller, the camera board(s), or a combination of the aforementioned. The stereo baseline (i.e. distance between each sensor in a stereo system) is also another potential source of inaccuracy in the system, as even tiny variations of around a thousandth of a meter have yielded noticeably different results in testing. Due to the limited precision with which the baseline was measured, further improvements in this regard may be possible with appropriate measuring devices.

4.3 Overall results

Overall, the system components have exhibited varying degrees of satisfactory performance. Considering the original requirements bullet lists in subsections 1.4.1, 1.4.2, and 1.4.3 we can attempt to measure some of the overall results of the thesis project. For the requirements of the robot agent, it successfully meets every requirement specified in its requirements list albeit with the caveats discussed in previous sections. The host platform successfully meets every requirement as well barring the one requirement regarding use of SLAM results for further movement order computation. As for the SLAM system itself, while it does yield some results in both scenarios described in its requirements, it is hardly satisfactory due to the limitations mentioned in the previous section. It is highly likely—probable even—that were the aforementioned limitations with the robot agent successfully mitigated or non-present, different results could have been obtained that would be more in line with the intended use-case of the designed system.

5

Suggestions and Conclusion

5.1 Suggestions for future work

The final results of the project were the results of both the work done during the course of the project period, but also a result of the initial choices made during the pre-project analysis. With the wealth of experience and benefit of hindsight—which is always 20/20—it is possible to suggest improvements in some areas which may have produced overall different results.

For example, an alternative selection of materials for the robot agent such as cameras with higher capture rate and greater transmission interface speeds, a computational unit with more on-board memory as well as a wireless chip with greater network speed and throughput, could likely provide the desired the results. Based on the parameters of the Kitti dataset [51]—which successfully operates at a frame rate of 15 fps for its ORB-SLAM3 example [54]—we can set the minimum frame rate at 15 fps. With VGA resolution of 620x480, double that of QVGA, and JPEG compression of the same rates as with the current camera setup, it is likely each individual image would average roughly 30-40 KiB per image. With a frame rate of 15 fps—in a perfect world—would require roughly 450-600 KiB transfer rate per second, or roughly 3686-4916kbps, in order to accommodate this frame rate. It is as such possible to deduce that the communication's interface between the camera module and the wireless chip of the microcontroller would need to be capable of speeds in excess of this in order to facilitate 15 fps in the ORB-SLAM3 side of the system. The RP2040 chip which powers the currently selected microcontroller is capable of

roughly 62.5mbps SPI speed at the default clock at 133MHz [55], with greater speeds being achievable with some light overclocking. Given this, the main bottlenecks that need addressing are the camera capture speeds, SPI speeds, as well as the network speed of the wireless chip. The presence of more on-board memory would also help alleviate these issues, as it would be possible to concurrently capturing and transmission images with enough memory, rather than the current sequential approach of capture and fully transferring the image parts before subsequent capture orders. The minimum desirable amount of on-board memory can be calculated to be $image\ size * fps$, consequently $40\text{KiB} * 15 = 600\text{KiB}$ in terms of space for purely image data, with more needing to be added on top for other operations. The result being that it is likely desirable to have minimum 1 MiB of on-board memory.

The aforementioned specifications are not guaranteed to mitigate the performance issues experienced with the current hardware configuration, but it is thought to be likely. One primary motivation for using a microcontroller for this project was the power consumption savings, and perceived challenge this approach would yield. If the use of a microcontroller could be foregone, in terms of price and performance the Raspberry Pi Zero W would likely be a decent candidate for consideration. In terms of specifications, the Zero W provides 512 MiB of on-board memory, a BCM43438 single-band 802.11 b/g/n 2.4GHz-capable network chip—yielding theoretical network speeds of up to 72mbps, though in reality likely much lower [56]—and the possibility of using native Raspberry Pi cameras foregoing the need of custom driver software [57, 58]. In terms of power consumption, the Zero W is expected to have a typical bare-board consumption of 150mA [59]; slightly more than the Pico's ≈93mA [34]. Documentation for the expected Pico W power consumption has not been found, but is likely to be somewhat higher than the regular Pico as a result of the added wireless chip and other minor differences. That said, as the power consumption results showed it is unlikely that such an increase in power consumption would significantly impact the overall robot's possible battery runtime, though the addition of more powerful cameras likely would. An additional possibility to boost network speed with the Zero W could be the addition of a USB wireless adapter, providing greater speeds than the single-band BCM43438.

Regarding design of the movement system such that it can extract information from the SLAM system with which it can plan movement orders for the robot agent, there are a number of different approaches that are possible. One possible approach might be to process the point cloud of each map and classify points belonging to various aspects of the room such as walls, floor, and ceiling, subsequently processing these points together into positional data for objects to avoid [60]. This information can then in turn be used to for example utilize a chain-based path planning approach to navigate the room and obtain sensor data from which the overall room layout can be constructed.

5.2 Conclusion

In the final section of chapter one, a research question was posed: *Is it possible to achieve adequate performance in a vSLAM or viSLAM approach using low-performance, low-power- consumption computational devices for sensor data collection, and offloading computing to a stationary host platform?* This thesis has examined the theory, methods, portrayed and discussed the results of the project attempting to answer the research question.

In order to facilitate low-performance, low-power-consumption sensor data collection for use with vSLAM, components for a robot agent meeting these specifications were selected, put together, and software was created with which to power and run the agent. The created robot agent can collect the necessary data, transmit it to a stationary computation host platform, as well as receive and execute movement orders sent to it from the platform. The quality of and rate at which it collects the sensor data was not within the ideal range, but still demonstrates in principle the viability of this approach.

A remote host application was created in order to test and demonstrate the capabilities of a system using the collected sensor data from the aforementioned robot agent. The application is capable of issuing orders to the robot, receiving sensor data as it is transmitted, process the sensor data to make it suitable for use with the vSLAM system, performing vSLAM with the processed sensor data, and transmitting movement orders to the robot agent. The movement orders are however not, as initially intended, computed from the results of the SLAM process, but rather manually issued due to time constraints. Additionally, the SLAM system does not run perfectly, but is regardless capable of demonstrating in principle that the collected and processed sensor data is indeed viable for use in a SLAM system, with some previously discussed caveats.

We can conclude that the thesis is partially successful in achieving its intended goal of answering the research question, indicating that it is indeed possible to achieve this form of SLAM with the selected parameters. It also highlights issues with the current approach, and suggests ways to mitigate these issues in order to achieve improved overall results for future attempts at this form of remote vSLAM.

Bibliography

[1] F. A. Haq, B. S. B. Dewantara, and B. S. Marta, "Room mapping using ultrasonic range sensor on the atracbot (autonomous trash can robot): A simulation approach," in *2020 International Electronics Symposium (IES)*, pp. 265–270, 2020.

[2] M. Rivai, D. Hutabarat, and Z. M. J. Nafis, "2d mapping using omnidirectional mobile robot equipped with lidar," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 18, no. 3, pp. 1467–1474, 2020.

[3] R. Mur-Artal and J. D. Tardós, "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[4] C. Toft, D. Turmukhambetov, T. Sattler, F. Kahl, and G. J. Brostow, "Single-image depth prediction makes feature matching easier," in *Computer Vision – ECCV 2020* (A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, eds.), (Cham), pp. 473–492, Springer International Publishing, 2020.

[5] M. Servières, V. Renaudin, A. Dupuis, and N. Antigny, "Visual and visual-inertial slam: State of the art, classification, and experimental benchmarking," *Journal of Sensors*, vol. 2021, pp. 1–26, 2021.

[6] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics & Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[7] M. Magnabosco and T. P. Breckon, "Cross-spectral visual simultaneous localization and mapping (slam) with sensor handover," *Robotics and Autonomous Systems*, vol. 61, no. 2, pp. 195–208, 2013.

[8] M. Tomono, "Robust 3d slam with a stereo camera based on an edge-point icp algorithm," in *2009 IEEE International Conference on Robotics and Automation*, pp. 4306–4311, 2009.

[9] S. Perera and A. Pasqual, "Towards realtime handheld monoslam in dynamic environments," in *Advances in Visual Computing* (G. Bebis, R. Boyle, B. Parvin, D. Koracin, S. Wang, K. Kyungnam, B. Benes, K. Moreland, C. Borst, S. DiVerdi, C. Yi-Jen, and J. Ming, eds.), (Berlin, Heidelberg), pp. 313–324, Springer Berlin Heidelberg, 2011.

[10] S. Perera, D. Barnes, and D. Zelinsky, *Exploration: Simultaneous Localization and Mapping (SLAM)*, pp. 268–275. Boston, MA: Springer US, 2014.

[11] E. Eade and T. Drummond, "Unified loop closing and recovery for real time monocular slam.," in *BMVC*, vol. 13, p. 136, 2008.

[12] A. Thallas, E. Tsardoulias, and L. Petrou, "Particle filter — scan matching slam recovery under kinematic model failures," in *2016 24th Mediterranean Conference on Control and Automation (MED)*, pp. 232–237, 2016.

[13] J. Cheng, L. Zhang, Q. Chen, X. Hu, and J. Cai, "A review of visual slam methods for autonomous driving vehicles," *Engineering Applications of Artificial Intelligence*, vol. 114, p. 104992, 2022.

[14] I. Abaspur Kazerouni, L. Fitzgerald, G. Dooly, and D. Toal, "A survey of state-of-the-art on visual slam," *Expert Systems with Applications*, vol. 205, p. 117734, 2022.

[15] H. Yin, Z. Ma, M. Zhong, K. Wu, Y. Wei, J. Guo, and B. Huang, "Slam-based self-calibration of a binocular stereo vision rig in real-time," *Sensors*, vol. 20, no. 3, 2020.

[16] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard, "3-d mapping with an rgb-d camera," *IEEE Transactions on Robotics*, vol. 30, no. 1, pp. 177–187, 2014.

[17] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[18] OpenCV, "Camera calibration - opencv 4.7.0 documentation." `https://docs.opencv.org/4.7.0/dc/dbb/tutorial_py_calibration.html`, Accessed: 2023-04-10, 2022.

[19] A. Macario Barros, M. Michel, Y. Moline, G. Corre, and F. Carrel, "A comprehensive survey of visual slam algorithms," *Robotics*, vol. 11, no. 1, 2022.

[20] C. Campos, R. Elvira, J. J. G. Rodriguez, J. M. M. Montiel, and J. D. Tardos, "ORB-SLAM3: An accurate open-source library for visual, visual–inertial, and multimap SLAM," *IEEE Transactions on Robotics*, vol. 37, pp. 1874–1890, dec 2021.

[21] K. Tateno, F. Tombari, I. Laina, and N. Navab, "Cnn-slam: Real-time dense monocular slam with learned depth prediction," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6565–6574, 2017.

[22] X. Zhang, J. Lai, D. Xu, H. Li, and M. Fu, "2d lidar-based slam and path planning for indoor rescue using mobile robots," *Journal of Advanced Transportation*, vol. 2020, pp. 1–14, 2020.

[23] S. Karam, F. Nex, B. T. Chidura, and N. Kerle, "Microdrone-based indoor mapping with graph slam," *Drones*, vol. 6, no. 11, 2022.

[24] UiT The Arctic University of Norway, "Dte-3605 - virtual reality, graphics and animation - project." `https://uit.no/utdanning/emner/emne?p_document_id=765882`, Accessed: 2023-01-28, 2022.

[25] UiT The Arctic University of Norway, "Dte-3609 - virtual reality, graphics and animation - theory." `https://uit.no/utdanning/emner/emne?p_document_id=743948`, Accessed: 2023-01-28, 2022.

[26] UiT The Arctic University of Norway, "Dte-3606 - artificial intelligence and intelligent agents - project." `https://uit.no/utdanning/emner/emne?p_document_id=765881`, Accessed: 2023-01-28, 2022.

[27] UiT The Arctic University of Norway, "Dte-3608 - artificial intelligence and intelligent agents - theory." `https://uit.no/utdanning/emner/emne?p_document_id=743949`, Accessed: 2023-01-28, 2022.

[28] Proxmox Server Solutions GmbH, "Proxmox virtual environment." `https://www.proxmox.com/en/`, Accessed: 2023-02-24, 2023.

[29] Schibsted Media Group, "Prisjakt." `https://www.prisjakt.no`, Accessed: 2023-01-28, 2023.

[30] Prisguiden AS, "Prisguiden." `https://www.prisguiden.no`, Accessed: 2023-01-28, 2023.

[31] S. Ashcroft, "Timeline: causes of the global semiconductor shortage." `https://supplychaindigital.com/top10/timeline-causes-of-the-global-semiconductor-shortage`, 2023.

[32] Digital Impuls, "Mini 2mp spi kamera modul." https://www.digitalimpuls.no/the-pi-hut/148459/mini-2mp-spi-kamera-modul-for-pi-pico-med-gpio, Accessed: 2023-01-26, 2023.

[33] Uctronics, "Arducam 2mp stereo camera for raspberry pi." https://www.uctronics.com/arducam-2mp-stereo-camera-for-raspberry-pi-nvidia-jetson-nano-xavier-nx-dual-ov2311-monochrome-global-shutter-camera-module.html Accessed: 2023-01-26, 2023.

[34] Raspberry Pi foundation, "Raspberry pi pico w datasheet." https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf, Accessed: 2023-01-26, 2022.

[35] Microsoft, "Visual studio code." https://code.visualstudio.com/, Accessed: 2023-01-28, 2023.

[36] Raspberry Pi foundation, "Raspberry pi pico sdk." https://github.com/raspberrypi/pico-sdk, Accessed: 2023-01-28, 2022.

[37] lwIP developers group, "lwip - a lightweight tcp/ip stack." https://savannah.nongnu.org/projects/lwip/, Accessed: 2023-02-16, 2023.

[38] Amazon Web Services, "Freertos - real-time operating system for microcontrollers." https://www.freertos.org/, Accessed:2023-02-14, 2023.

[39] Open Robotics, "Robot operating system." https://www.ros.org/, Accessed: 2023-01-29, 2023.

[40] micro-ROS Community, "micro-ros - a lightweight and flexible ros client library for microcontrollers." https://micro.ros.org/, Accessed: 2023-02-05, 2023.

[41] The Point Cloud Library, "Point cloud library." https://pointclouds.org/, Accessed: 2023-01-29, 2023.

[42] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[43] The Qt Company, "Qt - cross-platform software design and development tools." https://www.qt.io/, Accessed: 2023-01-28, 2023.

[44] Khronos Group, "Open graphics library (opengl) - a cross-language, cross-platform graphics application programming interface." https://www.opengl.org/, Accessed: 2023-01-28, 2023.

[45] ArduCAM, "Arducam pico spi cam driver." https://github.com/ArduCAM/PICO_SPI_CAM, Accessed: 2023-02-16, 2021.

[46] The Open Group, "Posix™ 1003.1 frequently asked questions (faq version 1.18)." https://www.opengroup.org/austin/papers/posix_faq.html, Accessed: 2023-04-11, 2020.

[47] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov, *C++ Standard Template Library*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2000.

[48] cppreference, "C++ concurrency support library - cppreference.com." https://en.cppreference.com/w/cpp/thread, Accessed: 2023-03-22, 2022.

[49] A. Albertini, "This pdf is a jpeg; or, this proof of concept is a picture of cats," *PoC or GTFO 0x03*, 2014.

[50] G. Kessler, "Gck's file signatures table." https://www.garykessler.net/library/file_sigs.html, Accessed: 2023-04-26, 2023.

[51] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[52] A. Handa, R. A. Newcombe, A. Angeli, and A. J. Davison, "Real-time camera tracking: When is high frame-rate best?," in *Computer Vision – ECCV 2012* (A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, eds.), (Berlin, Heidelberg), pp. 222–235, Springer Berlin Heidelberg, 2012.

[53] Uctronics, "Ov5642 sensor datasheet." https://www.uctronics.com/download/cam_module/OV5642DS.pdf, Accessed: 2023-01-28, 2009.

[54] C. Campos, R. Elvira, J. J. G. Rodriguez, J. M. M. Montiel, and J. D. Tardos, "ORB-SLAM3 source code." https://github.com/UZ-SLAMLab/ORB_SLAM3, Accessed: 2023-03-25, 2021.

[55] Raspberry Pi foundation, "Rp2040 datasheet." https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf, Accessed: 2023-05-04, 2023.

[56] M. Zolnierczyk, "Raspberry pi network speed test: Rpi2, rpi3, zero, zerow (lan&wifi)." https://notenoughtech.com/raspberry-pi/raspberry-pi-internet-speed/, Accessed: 2023-05-04, 2017.

[57] Raspberry Pi foundation, "Raspberry pi zero w product listing." https://

www.raspberrypi.com/products/raspberry-pi-zero-w/, Accessed: 2023-05-04, 2023.

[58] Cypress, "Bcm43438 datasheet (pdf) - cypress semiconductor." https://www.alldatasheet.com/datasheet-pdf/pdf/1018493/CYPRESS/BCM43438.html, Accessed: 2023-05-04, 2017.

[59] Raspberry Pi foundation, "Raspberry pi hardware documentation." https://www.raspberrypi.com/documentation/computers/raspberry-pi.html, 2023.

[60] I. Anagnostopoulos, V. Pătrăucean, I. Brilakis, and P. Vela, "Detection of walls, floors, and ceilings in point cloud data," in *Construction Research Congress 2016*, pp. 2302–2311, 2016.

[61] ArduCAM, "Spi camera for raspberry pi pico (docs at bottom of page)." https://docs.arducam.com/Arduino-SPI-camera/Legacy-SPI-camera/Pico/Camera-Module/SPI-Camera/, Accessed: 2023-02-14, 2021.

A

Task Description

This appendix contains the initial task description from which this thesis created as originally received at the beginning of the project period.

Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering
UiT - The Arctic University of Norway

3D room mapping and modelling application utilizing a microcontroller agent with stereoscopic cameras

Stian Jakobsen

Thesis for Master of Science in Technology / Sivilingeniør

Problem description

The aim is to create an application capable of mapping a room in 3D from sensor/camera data provided by two cameras from a microcontroller agent/robot through wireless communication (likely Wi-Fi), with parsing and mapping done by a separate AI-fueled (TensorFlow, OpenCV etc.) application which then gives the agent new orders based on the mapping done in order to fully map the room. One of the aims is to keep the robot agent as minimalized as possible, with as little processing power and power consumption as possible (also keeping cost down for each respective agent). Finally, if time allows, modelling the mapped room in real time with a graphical modelling interface program or game engine (such as Unreal Engine or Godot) might be desirable to provide a client user interface to the system.

Objectives

The project may be divided into four parts:

1. Reviewing material/research on the subject(s), choosing hardware, programming languages, frameworks, and other tools based on this review.
2. Creating the robot and programming the microcontroller with its cameras and servos, making it communicate wirelessly with the desired application host.
3. Creating the AI application/container with chosen image depth processing/AI techniques and frameworks with photo data from robot as basis for analysis, providing translated movement orders to robot based on analysis. Additionally, providing the processed data in a manner accessible by the client application.
4. Creating a client application visualizing the room as it is being created in real time from the processed data provided by the AI application, providing capability of directly controlling the robot with user input if so desired (thus temporarily disabling AI application).

Suggested hardware

Only hardware for the robot is necessary, as the required hardware for the other development is already in-place. Suggestions for robot microcontroller is the RP2040 in the form of the Raspberry Pi Pico W, as it is cheap, ubiquitous (which is nice considering the ongoing silicon shortage) and provides (likely) adequate performance in addition to Wi-Fi connectivity capabilities. Additionally, the maximum power consumption of the Pico W is less than 93mA and can as such be powered for extensive periods with simple AA-or AAA-batteries (as the Pico W requires between 1.8 and 5.5V DC). Servo motors should be easily and cheaply available, as well as wheels or belts for movement. Lastly, it is suggested to use two OV2640 (2MP) or OV5642 (5MP) cameras in tandem in place of a single stereoscopic camera. These cameras use the serial peripheral interface and/or i2c to communicate with microcontrollers, of which the Pico has two, and is as such ideal for use with microcontrollers.

Dates

Date of distributing the task:	<09.01.2023>
Date for submission (deadline):	<15.05.2023>

Contact information

Candidate	Stian Endrè Jakobsen stian.e.jakobsen@uit.no
Supervisors at UiT-IVT	Rune Dalmo rune.dalmo@uit.no Børre Bang borre.bang@uit.no

General information

This master thesis should include:

- ❄ Preliminary work/literature study related to actual topic
  - A state-of-the-art investigation
  - An analysis of requirement specifications, definitions, design requirements, given standards or norms, guidelines and practical experience etc.
  - Description concerning limitations and size of the task/project
  - Estimated time schedule for the project/ thesis
- ❄ Selection & investigation of actual materials
- ❄ Development (creating a model or model concept)
- ❄ Experimental work (planned in the preliminary work/literature study part)
- ❄ Suggestion for future work/development

Preliminary work/literature study

After the task description has been distributed to the candidate a preliminary study should be completed within 3 weeks. It should include bullet points 1 and 2 in "The work shall include", and a plan of the progress. The preliminary study may be submitted as a separate report or "natural" incorporated in the main thesis report. A plan of progress and a deviation report (gap report) can be added as an appendix to the thesis.

In any case the preliminary study report/part must be accepted by the supervisor before the student can continue with the rest of the master thesis. In the evaluation of this thesis, emphasis will be placed on the thorough documentation of the work performed.

Reporting requirements

The thesis should be submitted as a research report and could include the following parts; Abstract, Introduction, Material & Methods, Results & Discussion, Conclusions, Acknowledgements, Bibliography, References and Appendices. Choices should be well documented with evidence, references, or logical arguments.

The candidate should in this thesis strive to make the report survey-able, testable, accessible, well written, and documented.

Materials which are developed during the project (thesis) such as software / source code or physical equipment are considered to be a part of this paper (thesis). Documentation for correct use of such information should be added, as far as possible, to this paper (thesis).

The text for this task should be added as an appendix to the report (thesis).

General project requirements

If the tasks or the problems are performed in close cooperation with an external company, the candidate should follow the guidelines or other directives given by the management of the company.

The candidate does not have the authority to enter or access external companies' information system, production equipment or likewise. If such should be necessary for solving the task in a satisfactory way a detailed permission should be given by the management in the company before any action are made.

Any travel cost, printing and phone cost must be covered by the candidate themselves, if and only if, this is not covered by an agreement between the candidate and the management in the enterprises.

If the candidate enters some unexpected problems or challenges during the work with the tasks and these will cause changes to the work plan, it should be addressed to the supervisor at the UiT or the person which is responsible, without any delay in time.

Submission requirements

This thesis should result in a final report with an electronic copy of the report including appendices and necessary software, source code, simulations and calculations. The final report with its appendices will be the basis for the evaluation and grading of the thesis. The report with all materials should be delivered according to the current faculty regulation. If there is an external company that needs a copy of the thesis, the candidate must arrange this. A standard front page, which can be found on the UiT internet site, should be used. Otherwise, refer to the "General guidelines for thesis" and the subject description for master thesis.

The supervisor(s) should receive a copy of the the thesis prior to submission of the final report. The final report with its appendices should be submitted no later than the decided final date.

/B

Project setup and use instructions

This appendix contains the various setup instructions for building and using the created code for each respective sub-part of the project, in addition to the required default hardware configurations needed.

Picobot

In order to run the Picobot-side of the project, a number of dependencies need to be met:

Dependencies

- Raspberry Pi Pico W (or other RP2040-based boards with the same minimum capabilities, similar network stack, and GPIO. Only the Pico W has been tested).

- ArduCAM OV5642 cameras (other SPI cameras can be added, but will require modification to the camera handler class to accommodate the changes in API).

- A C++17 compatible compiler (only GCC 12.2.0 `arm-none-eabi`, as provided by the `arm-none-eabi-gcc` Arch Linux package, has been verified to work. Newer versions have been confirmed to crash as per Pico-SDK version 1.5 and FreeRTOS-Kernel V202110.00-SMP. This may change as these are updated).

- CMake (version 3.14 or newer).

- Raspberry Pi Pico SDK (version 1.4 and 1.5 tested, v1.5 or newer preferred).

- FreeRTOS-Kernel (SMP branch required, only tested with the V202110.00-SMP release).

- ArduCAM OV5642 Raspberry Pi Pico/SPI driver (provided in the "third-party" directory of the project, as we rely on a modified version).

- Optional: Second Pico (W-version optional) with Picoprobe and OpenOCD for easy debug/development work with Pico.

Build instructions

1. Navigate to the Picobot repository root directory.

2. Ensure dependencies are satisfied.

3. Ensure FreeRTOS-Kernel and Pico-SDK paths are available as environment variables `$FREERTOS_KERNEL_PATH` and `$PICO_SDK_PATH` respectively (see CMake file for more information).

4. Read and follow the instructions in subsection *Connecting to Wi-Fi*.

5. Create a build directory and navigate into it.

6. Create CMake configuration files (e.g. with command `"cmake ../"` optionally specifying a generator such as Ninja if so desired).

7. Build with `make` or `ninja` if using the Ninja generator (make flag `"-$(nproc)"` recommended for faster build times when using `make`).

8. This should produce a `picobot.uf2` binary which can be put on the Pico W for execution (see Pico documentation for how to do this). An alternative is using Picoprobe with the created `picobot.elf`.

Usage instructions

After completing the build instructions and putting the binary on the Pico through any of the officially supported methods, follow the instructions in *Wiring up the Picobot*. Following completed wiring of the minimum-required components (Picobot will *not* boot properly without both cameras connected properly), Picobot is ready for boot up and use. On power-on, Picobot's LED will start blinking, indicating that it is attempting to connect to the specified Wi-Fi network. If the connection attempt should fail, the blinking will continue perpetually (or until the OS crashes/some exception happens which causes a kernel panic). By default, Picobot makes 5 attempts at connecting to the Wi-Fi before aborting. Should the connection succeed, the blinking will stop, and the LED will be fully solid; indicating that the robot is ready for connections from the server application. Picobot outputs its DHCP-assigned IP address through serial output, either UART or USB depending on selections made in the CMake file (see the Raspberry Pi Pico documentation for swapping UART output to USB). An alternative and likely easier approach to obtaining the IP address would be to look up the IP for the Picobot hostname in your router interface or polling the local DNS. For actual use after this stage, follow the instructions for the server application in the Picobotserver *usage instructions*.

Connecting to Wi-Fi

Picobot requires a Wi-Fi connection in order to function. Requirements for the Wi-Fi network are the same as all Raspberry Pi Pico W boards, such as only supporting 2.4GHz bands. Additionally, Picobot does not support enterprise security protocols and has only been tested with the WPA2-Personal protocol. Any other protocol is likely to either not work or result in undefined behaviour.

In order to connect Picobot to your Wi-Fi network, it is necessary to create the file `include/wifi_settings.hpp` (next to `main.hpp`) and adding the code seen in listing B.1 to the created file, substituting the string values with your network's SSID and WPA.

```
#pragma once

#define my_ssid "my_ssid"
#define my_wpa "my_wpa"
```

Listing B.1: wifi_settings.hpp example

Assuming the SSID and WPA are valid and correct, Picobot should now connect to the Wi-Fi network on bootup.

Wiring up the Picobot

In order to connect Picobot to its various peripherals such as camera sensors, it is necessary to wire the various I/O pins of the peripherals to the assigned GPIO pinouts on the Pico W. The pinout is set in the modules for each respective module's include file(s) (e.g. camera pinouts are set in the camera module's *cam_pins.hpp* file), and can be changed as desired with some caveats (though alternative configurations have not been tested and as such should not be considered "safe"). For example, it is possible to change the camera pinouts provided each respective pin type is appropriately handled (i.e. SPI pins such as the CS pin *need* to be wired to an SPI CS-capable GPIO pin, and similarly PWM requires a PWM-capable GPIO pin). An example of the verified (and thus recommended) pinout for Picobot can be seen in figure B.1, with the exception of wiring for the motor power pins on the TB6612 H-bridge which have not been included.

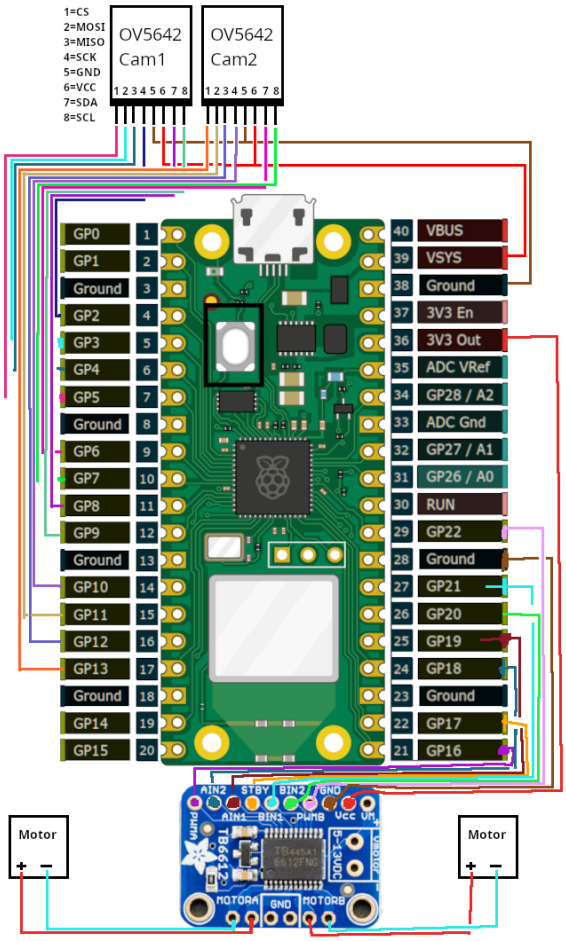

Figure B.1: Picobot wiring diagram example

Motor power wiring for the TB6612 depends on the motor configuration and DC motor capabilities, and as such need to be considered appropriately on a per-case basis. For example, motors capable of running in the span of 4.5-9v at 0.2A max draw were used during testing, and performed sufficiently with motor power directly from the Pico W. In this case, the VM pin on the TB6612 was wired to V_{sys} on the Pico W as it could make do with the current and voltage supplied from the Pico W running on USB power (through a 5V 2.5A powerbank). Some additional testing was done with some other motors that required 12V 0.5A per motor, and in this case it was necessary to connect a standalone 12V (portable) power supply to the 5V-13V Vmotor pins on the TB6612, omitting the VM pin altogether to avoid accidentally feeding the Pico W 12V back through the TB6612.

PicobotServer

Like Picobot, the PicobotServer-side of the project has a number of dependencies that need to be met in order to build properly:

Dependencies

- A C++20 compatible compiler (tested with GCC version 12.2.1 and Clang 14.0.6 for the x86-64 platform, any other versions or platforms are unverified and unsupported).
- CMake (version 3.14 or higher).
- Linux (tested with kernel 6.2.12 and libstdc++ version 6. Other kernel versions may work, but are untested).
- Python (tested with version 3.10, errors with version 3.11 or higher) for ORB-SLAM3.
- spdlog (logging library)
- OpenCV (version 4.4 minimum, tested with version 4.7.0)
- Pangolin (OpenGL library)
- Eigen (version 3 minimum, tested with 3.4.0-1)
- ORB-SLAM3 and all its additional dependencies beyond the above (only

tested with release version 1.0)

- fmtlib (FOSS formatting library, until libstdc++ supports C++20 format)

Build instructions

1. Ensure the dependencies are met

2. Navigate to the project repository's root directory (clone recursively if using the git version)

3. If not using the bundled ORB-SLAM3 version, follow the instructions in *ORB-SLAM3 bugfixes* as there are a number of mandatory fixes that need to be manually implemented in this case.

4. Navigate to the ORB-SLAM3 directory (if using the bundled version, it is in the `thirdparty` subdirectory)and build its thirdparty dependencies (`g2o`, `DBoW2`, `Sophus` etc.)

5. Create a build directory for ORB-SLAM3, navigate into it, and create its CMake configuration files with the `cmake <path_to_orb_slam3_root>` command, optionally specifying a generator such as Ninja.

6. Build ORB-SLAM3 with `make` or `ninja` (using the build flag `-j$(nproc)` is *highly* recommended for faster build speeds when using `make`).

7. Navigate back to the project root repository, create a build directory for the server project, navigate into the build directory, and create its CMake configuration files as with ORB-SLAM3.

8. Build the server application as before with ORB-SLAM3, using `make` or `ninja`.

Usage instructions

Following completion of the build instructions, the server application is now ready for use. It is a CLI application, and as such requires launching from the terminal. Upon launch, the user may hit 'o' to display the help menu, showing the various options for each respective functionality of the application. In order to start the SLAM process, a number of modules need to be started and configured. Assuming that the remote agent used with the application is a Picobot variant, follow the instructions below:

1. Calibration of cameras using the option in the application. See further instructions in *camera calibration*.

2. Obtain the remote agent's IP address and use it with the "connect to Picobot" option in the CLI.

3. Start the server's TCP server module.

4. Start the SLAM system using the CLI.

Assuming every step is followed, and no error appears, the ORB-SLAM system should spawn a graphical user interface displaying both the camera feed (i.e. the latest image processed) and the map with point clouds generated by the SLAM library. The application will continue to run continuously unless an error occurs (such as no new images being fed to the SLAM system for a certain amount of time) or the user decides to stop the SLAM system. This can be achieved through a couple ways, but the recommended approach is to press the shutdown button in the SLAM interface, or by selecting the SLAM system shutdown option in the application's CLI. While the SLAM system is running, it is possible to send movement commands through the CLI to the remote agent, in order to facilitate movement (as it currently does not support independent movement decision-making). Once the system has shut down, a file is generated containing data for the robot's trajectory during the SLAM process. This can be easily plotted using Python (the process for achieving this is not covered in this paper).

Note that for ease of use and rapid testing purposes, the source code currently foregoes a lot of the user input by manually declaring IP addresses, port numbers, and directories in the source code of both the server application and robot agent. Change these to match the user's server and robot IP addresses for easy testing.

The minimum requirements for this application are currently unknown, as it has only been tested on a relatively high-end system, and as such any performance impacts from running it on a lower-powered system are unknown.

ORB-SLAM3 bugfixes

A number of bugs and outright errors require fixing in order to use ORB-SLAM3 with this project's sensor configuration and dependencies in the case of not using the provided ORB-SLAM3 fork. Version 1.0 is still assumed. The following changes need to be made in order to circumvent the aforementioned errors:

- The compiler flag `-march=native` needs to be removed from both the ORB-SLAM3 repository's CMakeLists file and the ORB-SLAM3 dependency g2o's CMakeLists files (found in the `thirdparty/g2o` subdirectory in the ORB-SLAM3 directory), as this prevents a segmentation fault during real-time use of rectified images. This fix has not been extensively tested and may carry with it unknown consequences, but allowed for the collection of data.

- An additional segmentation fault was found when using pre-rectified images instead of having ORB SLAM3 rectify the images for us. This error happens because the system attempts to read `originalCalib2_` object for sensor type rectified (leading to a segmentation error as that object's size will be 0 in the case of rectified sensors). This can be mitigated by modifying the file `src/Settings.cc` (found in the ORB-SLAM3 directory) changing the lines shown in listing B.2 to those shown in listing B.3.

```
output << "" << ": [";
for(size_t i = 0; i < settings.originalCalib2_->size(); i++){
    output << " " << settings.originalCalib2_->getParameter(i);
}
output << " ]" << endl;
```

Listing B.2: Bugfix code before fix

```
output << "" << ": [";
if (settings.cameraType_ != Settings::Rectified) {
    for (size_t i = 0; i < settings.originalCalib2_->size(); i
        ++) {
        output << " " << settings.originalCalib2_->getParameter
            (i);
    }
}
output << " ]" << endl;
```

Listing B.3: Bugfix code after fix

Camera calibration

Calibration of cameras for use with the Picobot project assumes stereo cameras of the same make and type, and does as such not cover use-cases where dissimilar cameras are used (in terms of focal length, pixel size, resolutions, and so on). Additionally, calibration requires connecting the Picobot to the server application (or obtaining images from it in some other manner), in order to obtain the calibration images. Following this connection having been

established, and images being confirmed transmitted to the server, print a chessboard photo (A4 size was used in this project) and attach it to something flat such that no creasing happens.

Hold the chessboard in front of both cameras such that it is clearly visible in photos taken by both cameras (it is recommended to hold it close enough such that most if not the entire fame is filled by the chessboard), ensure that the corners of the chessboard are visible in either camera. It is recommended to capture minimum 30 photos for each camera, more photos may yield better results, but may also reach a point of diminishing returns. After capturing enough photos, move the photos from the capture directory into the respective "cam1_calib" and "cam2_calib" subdirectories in the input directory (create them if they don't exist). Finally, select the camera calibration in the CLI and follow the on-screen instructions. If no errors occur, the application should indicate the projection error rate from the calibration of both cameras, and output undistorted as well as rectified versions of the calibration images in the output directory. Additionally, confirm the existence of the intrinsic and extrinsic calibration files in the config directory.

Assuming these have been created, it is now necessary to create and fill in the acquired parameters into an ORB-SLAM3 configuration file for the camera, which will be used in the ORB-SLAM process. This configuration file only requires the camera matrix for camera 1 as provided in the extrinsic configuration file, as well as parameters for the resolution (camera width and height), frames per second, whether the cameras are RGB or BGR, the stereo baseline (i.e. the distance between the two camera sensors in meters), as well as the depth threshold (manually adjusted from the recommended selected start value of 50.0). See the ORB-SLAM3 documentation for more information regarding the specifics. Additionally, there are some ORB-SLAM-related parameters that may be set and fine-tuned. An example file has been provided in the examples subdirectory in the project root directory.

/C

ArduCAM OV5642 documentation

This appendix contains an excerpt from the documentation for the ArduCAM OV5642 5MP camera as obtained from ArduCAM's website [61]. All rights belong to ArduCAM.

5 ArduChip Functions

ArduChip is ArduCAM property technology which handles all the timing control over camera interface, LCD interface, frame buffer and SPI interface timings with a set of registers. The ArduChip register address is also called Command Code, user can use low level APIs with these command codes to achieve customized combination of actions that off the shelf APIs don't have.

Different ArcuCAM platform uses different ArduChip and has different functionalities. Here is a list of possible hardware platforms:

Hardware Platform	Functions					
	Single Capture/ Read	Burst Read	Multiple Capture	Rewind	Low Power Mode	Short Video Capture
ArduCAM-Mini-2MP	√	√		√	√	
ArduCAM-Mini-5MP-Plus (OV5642)	√	√	√	√	√	√

5.1 Single Capture Mode

It is a basic capture function of the ArduChip. The capture command code is 0x84, and write '1' to bit[1] to start a capture sequence. And then polling bit[3] which is the capture done flag by sending command code 0x41. After capture is done, user have to clear the capture done flag by sending command code 0x41 and write '1' into bit[0] before next capture command.

5.2 Multiple Capture Mode

By sending the command code 0x81 and with writing the number of images to be capture into bit[2:0], before starting the capture command as the single capture sequence does. Please note that user should trade off between the resolution and number of images to be captured and do not make the frame buffer overflow.

5.3 Short Video Capture Mode

Use the same command as the Multiple Capture Mode. When the value bit[2:0] equals to 7, the ArduCAM will continuously capture the images until the entire frame buffer is full. User can save the captured MJPEG to AVI files to create short movie clips.

5.4 Single Read Operation

It is basic memory read function which start a single read operation and read a single byte each time. By sending command code 0x3D to start a single read operation, a single byte is read out from the frame buffer.

5.5 Burst Read Operation

It is advance capture function which can read multiple bytes out of the frame buffer by just sending a single command code 0x3C.

Please note that for these hardware platforms (ArduCAM-Mini-2MP, ArduCAM-Mini-5MP) the first read byte should be ignored in the first read transaction, because it is a dummy byte. In the following read transaction, the first byte read is the last read byte in the last read transaction, it is very important. And do not use other SPI command between burst read transaction. Detail timing can be found from Figure 5.

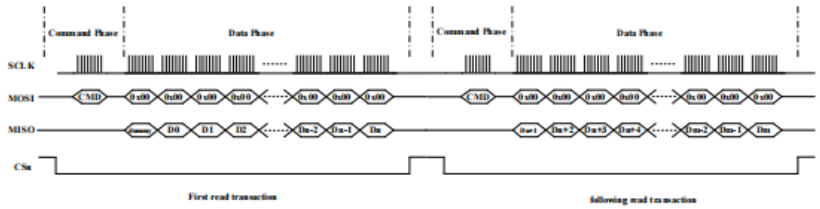

Figure 5 Burst read timing diagram 1

For hardware platforms (ArduCAM-Shield-V2, ArduCAM-Mini-5MP-Plus), you don't need to worry about the first byte. Detail timing can be found from Figure 6.

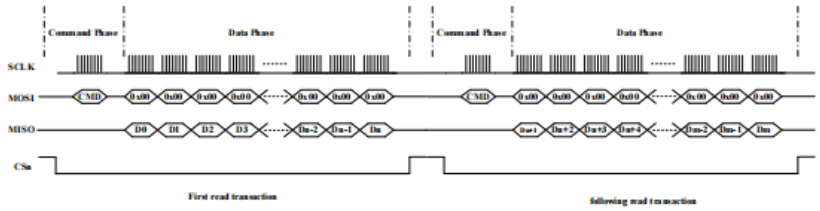

Figure 6 Burst read timing diagram 2

5.6 Rewind Read Operation

Rewind read is useful for some application that need access the same pixel data multiple times. By sending the command code 0x84 and write '1' to bit[5] in the data phase, it will reset the memory read pointer to ZERO. Then user can read the image data from the start of the memory.

5.7 Low Power Mode

For some battery powered device power consumption is very important. There are two levels to achieve low power mode, user have to combine these modes according to their own power strategy.

5.7.1 Power down the sensor circuit

It is achieved by controlling the power enable pin of the onboard LDOs. The power enable pin is controlled by the GPIO[2] of ArduChip. By sending the command code 0x86 and write '1' to bit[2] to enable the LDOs, or write '0' to bit[2] to disable the LDOs to save power. Note that power down the sensor circuit, the camera settings are lost. User should reinitialize the sensor when power up the sensor circuit again.

5.7.2 Sensor standby

It is achieved by controlling the power enable pin of the onboard LDOs. The power enable pin is controlled by the GPIO[1] of ArduChip. By sending the command code 0x86 and write '1' to bit[1] to set the sensor into standby mode, or write '0' to bit[1] to set the sensor out of standby mode. Note that the sensor settings are not lost when in standby mode, and reinitialize is not needed.

6 ArduCAM APIs

There are a set of API functions that issue different commands to ArduCAM shield.

6.1 void InitCAM (void)

InitCAM function initializes the hardware information of the user system, such as the SPI chip select port initialization and image sensor slave address initialization.

6.2 void flush_fifo (void)

flash fifo function is used to reset the fifo read pointer to ZERO.

6.3 void start_capture (void)

start_capture function is used to issue a capture command. After this command the ArduCAM hardware will wait for a start of a new frame then store the entire frame data to onboard frame buffer.

6.4 void clear_fifo_flag (void)

Once a frame image is buffed to onboard memory, the capture completion flag is asserted automatically. The clear_fifo_flag function is used to clear this flag before issuing next capture command.

6.5 void write_reg(uint8_t addr, uint8_t data)

Param1: ArduChip register address (or command code)

Param2: data to be written into the register

ite_reg is a basic function to write the ArduChip internal registers.

6.6 uint8_t read_reg(uint8_t addr)

Param1: ArduChip register address (or command code)

Return value: register value

read_reg is a basic function to read ArduChip internal register value.

6.7 uint32_t read_fifo_length(void)

Return value: 32 bit length of captured image

read_fifo_length function is used to determine the length of current captured image. Note the Rev.C shield doesn't support this feature.

6.8 void set_fifo_burst(void)

set_fifo_burst function is used to set the read memory into burst read mode. It should be called before burst memory read operation. Note the Rev.C shield doesn't support this feature.

6.9 int wrSensorRegs8_8(const struct sensor_reg*)

Param1: sensor setting data array

Return value: error status

wrSensorRegs8_8 function is used to write array of settings into sensor's internal register over I2C interface and sensor's register is accessed with 8bit address and 8bit data.

6.10 int wrSensorRegs8_16(const struct sensor_reg*)

Param1: sensor setting data array

Return value: error status

wrSensorRegs8_16 function is used to write array of settings into sensor's internal register over I2C interface and sensor's register is accessed with 8bit address and 16bit data.

6.11 int wrSensorRegs16_8(const struct sensor_reg*)

Param1: sensor setting data array

Return value: error status

wrSensorRegs16_8 function is used to write array of settings into sensor's internal register over I2C interface and sensor's register is accessed with 16bit address and 8bit data.

6.12 int wrSensorRegs16_16(const struct sensor_reg*)

Param1: sensor setting data array

Return value: error status

wrSensorRegs16_16 function is used to write array of settings into sensor's internal register over I2C interface and sensor's register is accessed with 16bit address and 16bit data.

6.13 byte wrSensorReg8_8(int regID, int regDat)

Param1: sensor internal register address

Param2: value to be written into the register

Return value: error status

wrSensorReg8_8 function is used to write a single sensor's internal register over I2C interface and sensor's register is accessed with 8bit address and 8bit data.

6.14 byte wrSensorReg8_16(int regID, int regDat)

Param1: sensor internal register address

Param2: value to be written into the register

Return value: error status

wrSensorReg8_16 function is used to write a single sensor's internal register over I2C interface and sensor's register is accessed with 8bit address and 16bit data.

6.15 byte wrSensorReg16_8(int regID, int regDat)

Param1: sensor internal register address

Param2: value to be written into the register

Return value: error status

wrSensorReg16_8 function is used to write a single sensor's internal register over I2C interface and sensor's register is accessed with 16bit address and 8bit data.

6.16 byte wrSensorReg16_16(int regID, int regDat)

Param1: sensor internal register address

Param2: value to be written into the register

Return value: error status

wrSensorReg16_16 function is used to write a single sensor's internal register over I2C interface and sensor's register is accessed with 16bit address and 16bit data.

6.17 byte rdSensorReg8_8(uint8_t regID, uint8_t* regDat)

Param1: sensor internal register address

Param2: value read from the register

Return value: error status

rdSensorReg8_8 function is used to read a single sensor's internal register value over I2C interface and sensor's register is accessed with 8bit address and 8bit data.

6.18 byte rdSensorReg16_8(uint16_t regID, uint8_t* regDat)

Param1: sensor internal register address

Param2: value read from the register

Return value: error status

rdSensorReg16_8 function is used to read a single sensor's internal register value over I2C interface and sensor's register is accessed with 16bit address and 8bit data.

6.19 byte rdSensorReg8_16(uint8_t regID, uint16_t* regDat)

Param1: sensor internal register address

Param2: value read from the register

Return value: error status

rdSensorReg8_16 function is used to read a single sensor's internal register value over I2C interface and sensor's register is accessed with 8bit address and 8bit data.

6.20 byte rdSensorReg16_16(uint16_t regID, uint16_t* regDat)

Param1: sensor internal register address

Param2: value read from the register

Return value: error status

rdSensorReg16_16 function is used to read a single sensor's internal register value over I2C interface and sensor's register is accessed with 16bit address and 16bit data.

6.21 void OV2640_set_JPEG_size(uint8_t size)

Param1: resolution code

OV2640_set_JPEG_size function is used to set the desired resolution with JPEG format for OV2640. Current support resolution is shown as follows:

#define OV2640_160x120	0	//160x120
#define OV2640_176x144	1	//176x144
#define OV2640_320x240	2	//320x240
#define OV2640_352x288	3	//352x288
#define OV2640_640x480	4	//640x480
#define OV2640_800x600	5	//800x600
#define OV2640_1024x768	6	//1024x768
#define OV2640_1280x1024	7	//1280x1024
#define OV2640_1600x1200	8	//1600x1200

6.22 void OV5642_set_JPEG_size(uint8_t size)

Param1: resolution code

OV5642_set_JPEG_size function is used to set the desired resolution with JPEG format for OV5642. Current support resolution is shown as follows:

#define OV5642_320x240	0	//320x240
#define OV5642_640x480	1	//640x480
#define OV5642_1024x768	2	//1024x768
#define OV5642_1280x960	3	//1280x960
#define OV5642_1600x1200	4	//1600x1200
#define OV5642_2048x1536	5	//2048x1536
#define OV5642_2592x1944	6	//2592x1944

6.23 void set_format(byte fmt)

set_format function is used to set the sensor between RGB mode and JPEG mode. The InitCAM function should be called after set_format function.

7 Registers Table

Sensor and FIFO timing is controlled with a set of registers which is implemented in the ArduChip. User can send capture commands and read image data with a simple SPI slave interface. The detail description of registers' bits can be found in the software section in this document. Not all the registers are implemented in a given hardware platform, please check the hardware develop guide for detail register description for certain hardware you've got.

As mentioned earlier the first bit[7] of the command phase is read/write byte, '0' is for read and '1' is for write, and the bit[6:0] is the address to be read or write in the data phase. So user has to combine the 8 bits address according to the read or write commands they want to issue.

Table 1 ArduChip Register Table

Register Address bit[6:0]	Register Type	Description
0x00	RW	Test Register
0x01	RW	Capture Control Register Bit[2:0]: Number of frames to be captured The value in this register + 1 equal to the number of frames to be captured. The value=7 means capture continuous frames until the frame buffer is full, it is used for short video clip recording.
0x02	RW	Bus Mode Determine who is owner of the data bus, only one owner is allowed. Bit[7:2]: Reserved Bit[1]: Camera write LCD bus Bit[0]: MCU write LCD bus
0x03	RW	Sensor Interface Timing Register Bit[0]: Sensor Hsync Polarity, 0 = active high, 1 = active low Bit[1]: Sensor Vsync Polarity 0 = active high, 1 = active low Bit[2]: LCD backlight enable 0 = enable, 1 = disable Bit[3]: Sensor PCLK reverse 0 = normal, 1= reversed PCLK
0x04	RW	FIFO control Register Bit[0]: write '1' to clear FIFO write done flag Bit[1]: write '1' to start capture Bit[4]: write '1' to reset FIFO write pointer Bit[5]: write '1' to reset FIFO read pointer
0x05	RW	GPIO Direction Register Bit[0]: Sensor reset IO direction

		Bit[1]: Sensor power down IO direction Bit[2]: Sensor power enable IO direction 0 = input, 1 = output
0x06	RW	GPIO Write Register Bit[0]: Sensor reset IO value Bit[1]: Sensor power down IO value Bit[2]: Sensor power enable IO value
0x3B	RO	Reserved
0x3C	RO	Burst FIFO read operation
0x3D	RO	Single FIFO read operation
0x3E	WO	LCD control register with RS=0
0x3F	WO	LCD control register with RS=1
0x40	RO	ArduChip version Bit[7:4]: integer part of the revision number Bit[3:0]: decimal part of the revision number
0x41	RO	Bit[0]: camera vsync pin status Bit[3]: camera write FIFO done flag
0x42	RO	Camera write FIFO size[7:0]
0x43	RO	Camera write FIFO size[15:8]
0x44	RO	Camera write FIFO size[22:16]
0x45	RO	GPIO Read Register Bit[0]: Sensor reset IO value Bit[1]: Sensor power down IO value Bit[2]: Sensor power enable IO value