UnimouseSim: A real-time mobile robot simulator with hardware-in-the-loop support for the micromouse contest

Pedro Victor Fontoura Zawadniak

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Industrial Engineering.

Work oriented by:

Professor PhD José Luis Sousa Magalhães Lima

Professor PhD Pedro Gomes Costa

Professor PhD André Luiz Regis Monteiro

Bragança

2019-2020

UnimouseSim: A real-time mobile robot simulator with hardware-in-the-loop support for the micromouse contest

Pedro Victor Fontoura Zawadniak

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Industrial Engineering.

Work oriented by:

Professor PhD José Luis Sousa Magalhães Lima

Professor PhD Pedro Gomes Costa

Professor PhD André Luiz Regis Monteiro

Bragança

2019-2020

Acknowledgement

Agradeço imensamente ao meu pai, Valdir Zawadniak, e minha mãe, Cleuslei Terezinha Fontoura, por todo suporte, educação, incentitvo e amor incondicional que tive durante o último ano de estudos e também por toda minha vida. Sem vocês, eu nunca teria chegado até aqui e por isso sou extremamente grato.

I would like to thank Professor José Lima and Pedro Costa, the supervisors of this work, for all the support and incentive in developing this work. Also to Professor André Luiz Regis Monteiro, who was willing to be the co-supervisor for the Double Degree program.

I thank Luis Piardi, Thadeu Brito and all other colleagues working in Research Centre in Digitalization and Intelligent Robotics (CeDRI) for all direct support, availability and positiveness I have received during the development of this work and other activities.

I thank all friends I have made during my life. You have helped me in innumerous ways and incentivized me to pursue my goals. You have brought me moments of happiness and experiences that helped me grow that I could not have gotten elsewhere.

I thank all members and supervisors of UTFPR's extension project Grupo de Educaçao Tutorial (GET) for introducing me to the field of robotics, which I consider my passion and has ultimately led to this work.

Abstract

Mobile robots are applied to various industrial contexts, performing repetitive and high-performance tasks. One way of generating interest in the study of robotics in this context is through robotics competitions.

The aim of this work is the development of a 3D mobile robotics simulator with hardware-in-the-loop capabilities. It includes developing models for standard components, such as time-of-flight sensors, wheel encoders, and direct current motors. The simulator interacts with development boards, programmed through Arduino-compatible libraries for communication with each robot component. By having the microcontrollers process each sensor's output and determine the appropriate motor commands, the microcontroller's limitations are present even during the simulation.

The simulator contains different environments, where users have to complete challenges that require sensor data to be interpreted and motor commands to be calculated for different purposes, namely following walls, controlling the robot speed, and developing algorithms for completing the micromouse competition.

A modification to the flood fill algorithm, commonly used in the micromouse competition, was proposed and analysed. It targets robots with a simple movement set, unable to perform turns while maintaining linear speed.

The simulator was used in the RoboSTEM hackathon, where students were presented with the challenge environments and developed their solutions. It provided insights about the problems they were asked to solve and the simulator software itself.

Keywords: mobile robotics; simulation; micromouse

Resumo

A robótica móvel é aplicada a diferentes contextos industriais, executando tarefas repetitivas e de alta performance. Uma forma de gerar interesse no estudo da robótica é por meio de competições.

O objetivo deste trabalho é o desenvolvimento de um simulador 3D de robótica com hardware-in-the-loop. Foi feito o desenvolvimento de components comumente utilizados nos robôs, como sensores time-of-flight, encoders e motores de corrente contínua. A intereção com o simulador é feita por placas de desenvolvimento programadas por bibliotecas compatíveis com o ambiente Arduino, específicas para cada componente. Sendo o microcontrolador responsável por processar as medições dos esnsores e determinar o comando apropriado para os motores, as limitações de memória e poder de processamento dos microcontroladores se fazem presentes mesmo no ambiente de simulação.

O simulador contém diferentes ambientes, em que o tulizador deve completar desafios que requerem a utilização dos sensores e atuadores para dieferentes fins, nomeadamnete o seguimento de paredes, controlo de velocidade e completar a competição do micromouse.

Foi proposta e analizada uma modificação ao algorítmo flood fill, comumente usado na competição do micromouse, que visa robôs com um conjunto de movimento limitado, inaptos a fazer curvas enquanto mantêm velocidade linear.

O simulador foi utilizado no hackathon RoboSTEM, em que os diferentes desafios foram apresntados a estudantes, e as soluções elaboradas por eles continham observações imporotantes sobre os problemas apresentados e sobre o simulador em si.

Palavras-chave: Robótica móvel; simulação; micromouse

Contents

Acknowledgement **v**

Abstract **vii**

Resumo **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 3

1.3 Document outline 4

2 Related work **7**

2.1 Robotics competitions 7

2.2 Simulation software 8

2.3 Maze solving algorithms 9

3 Tools **11**

3.1 Unity development platform 11

3.1.1 Unity Editor interface 12

3.1.2 Unity's basic concepts 13

3.1.3 Scripting 13

3.1.4 PhysX physics engine 15

3.1.5 Prefabs and ScriptableObjects 16

3.2 Computational models for robot components 17
3.2.1 Direct current motor 17
3.2.2 H-bridge 20
3.2.3 Incremental encoders 23
3.2.4 Time-of-flight range sensor 25
3.3 Differential drive kinematics 30
3.4 Maze solving algorithm 32
3.4.1 Flood fill algorithm 33
3.4.2 Proposed modified flood fill algorithm 34
3.4.3 Finding the shortest path 36
3.5 Arduino library 39
3.5.1 Simulator integration 39
3.5.2 Pose estimation and modified flood fill 43

4 System Architecture **45**
4.1 Robot simulation model 45
4.2 Serial Communication protocol 48
4.3 Maze Generator 50

5 Results **53**
5.1 Simulated environments 53
5.1.1 Speed control challenge 54
5.1.2 Distance control challenge 54
5.1.3 Time trial 56
5.1.4 Micromouse contest 57
5.2 Modified flood fill case study 59
5.2.1 Steps to find the best path 61
5.2.2 Comparison between traditional and modified algorithm 64
5.3 RoboSTEAM hackathon 65

6 Conclusions and Future Work **67**

List of Tables

3.1 Parameters for the motor models. 20
3.2 Description of the possible h-bridge states. 22
3.3 Sign-magnitude drive. 22
3.4 Locked anti-phase drive. 22
3.5 Sequence of states for quadrature encoders. 24
3.6 Vl53L0X measurement metrics. 28
3.7 Comparison between the standard deviation of the model and the sampled measurements. 29
3.8 Parameters for the VL53L0X range sensor model. 30
3.9 Upper and bottom path analysis. 33
3.10 Description of the internal library elements. 41
3.11 Description of the user-facing Arduino library elements. 41
3.12 Robot properties available as variables. 44

4.1 Physical properties of the robot model. 46
4.2 Description of the joint connections. 46
4.3 Description of the Time-of-flight sensors. 47
4.4 Models used for the robot components. 47
4.5 Description of the encoders. 48
4.6 Description of the h-bridges. 48
4.7 UART transmission times. 50
4.8 Obstacle physical dimensions. 51

5.1 Trips to All Japan 2018 maze. 64
5.2 Trips to All Japan 2019 maze. 64
5.3 Algorithm comparison for All Japan 2018 maze. 65
5.4 Algorithm comparison for the All Japan 2019 maze. 65

List of Figures

3.1 Common Unity editor windows. 12
3.2 Schematic diagram for a DC motor connected to a load. Adapted from [39]. 18
3.3 H-bridge schematic diagram and its possible states. Extracted from [43]. . 21
3.4 Signal output for quadrature encoders. Extracted from [44]. 23
3.5 Components of the range finder sensor model proposed by [47]. 27
3.6 Measurements performed for the VL53L0X range sensor. 29
3.7 Differential drive kinematics. Extracted from [51]. 30
3.8 Hypothetical maze in which the modified flood fill algorithm results differ. 32
3.9 Robot movement considered in the traditional flood fill. 33
3.10 Costs assigned by the traditional flood fill algorithm 34
3.11 Robot movement considered in the modified flood fill. 35
3.12 Costs assigned by the modified flood fill algorithm to the robot configurations. 37
3.13 Flowchart to micromouse Flood fill strategy adopted. Extracted from [52] . 38
3.14 Class diagram for Arduino library for interacting with the simulator. . . . 40
3.15 The Facade design pattern. 42
3.16 Robot facade class for simplified usage. 43
3.17 The Adapter design pattern. 43
3.18 Integrating an existing library with the adapter design pattern 44

4.1 The simulation robot model. 46
4.2 Timing diagram for one packet transmitted using the UART protocol, configured as 8N1. 48

4.3 Timing diagram for transmitting the voltage commands for the left and right motors sequentially. 49
4.4 Textual representation for the maze used in All Japan's 2018 edition. . . . 51

5.1 Screen capture of the main menu, showing the buttons for resuming execution, quitting the program, and switching between the simulated scenarios. 54
5.2 Graphical interface for establishing the serial port connection. 54
5.3 Screen capture during the speed control challenge. 55
5.4 Screen capture during the distance control challenge. 56
5.5 Screen capture during the time trial. 58
5.6 Screen capture of All Japan's 2018 maze generated environment. 58
5.7 Known maze and paths after first trip (All Japan 2018). 62
5.8 Known maze and paths after second trip (All Japan 2018). 62
5.9 Known maze and paths after third trip (All Japan 2018). 62
5.10 Known maze and paths after first trip (All Japan 2019). 63
5.11 Known maze and paths after second trip (All Japan 2019). 63
5.12 Known maze and paths after third trip (All Japan 2019). 63
5.13 Known maze and paths after fourth trip (All Japan 2019). 63
5.14 Shortest path according to the traditional Flood fill (All Japan 2018). . . . 64
5.15 Shortest path according to the traditional Flood fill (All Japan 2019). . . . 64
5.16 Activities during the RoboSTEAM hackathon. 66

Chapter 1

Introduction

This work documents the implementation of a 3D simulator with Hardware-in-the-loop (HIL) capabilities for the micromouse contest [1]. The simulator is an open-source project available in [2].

HIL is a technique used for simulating real-time control systems in which the actual processor is combined with a simulated environment representing the actual system [3].

For its development, commonly used robot components in the micromouse contest were created, namely encoders, range sensors, DC motors and their controllers. Also, HIL support was achieved by implementing a serial communication protocol.

The simulator was used to test a modified version of the flood fill algorithm [4]. The modification targets robots with a limited movement set, that can either move forward or rotate about its center of its wheel's axis.

1.1 Motivation

Robotics competitions are an excellent way to encourage research and to attract students to technological areas. The robotics competitions present problems that can be used as a benchmark to evaluate and to compare the performances of different approaches [5].

Adopting autonomous robots to explore unknown mazes can be fun and challenging, representing a unique tool to multidisciplinary and cognitive activity [6]. Simulations

with faithful models of robots, actuators and sensors are highly recommended to solve problems with mazes, reducing software or firmware development and debugging time. A robust simulator for a micromouse can reduce the difficulty of the transition between simulation and real contest.

The micromouse challenge addresses a problem in the mobile robots area. The competition is straightforward: to place the robot in the start square in the bottom left corner of the maze and let it find the central goal square [7]. The mouse must be able to navigate, self-localize and map an unknown maze or environment. When the mapping is done, the robot must calculate and go through the best path to the goal in the least amount of time [8]. Rules may differ depending on the event, but commonly, each robot gets a total of 10 minutes for the competition. A new run timer starts whenever the robot returns to the starting cell, and only the fastest run counts for ranking purposes.

The micromouse competition begins in New York City [9] and, since then, competitors have developed their approaches and adapted the new rules. Initially, purely electromechanical systems were used, without any digitized data or microprocessor. Over the years, technological improvements and the mastery of advanced sensors and actuators combined with the use of microprocessors are essential requirements for a micromouse team, showing a high level in these competitions. In this sense, 3D simulation environments with faithful models are gaining prominence.

Bearing in mind the competitors' demand for a simulator that brings together all the challenges and functionality of a real robot, in a HIL approach, the Unity platform was used to develop the 3D micromouse simulator presented in this work. Unity is a real-time 3D development platform that consists of a rendering and physics engine as well as a graphical user interface called the Unity Editor [10]. The platform's focus on the development of general-purpose mechanisms, with mechanisms that enable the creation and representation of complex 3D models. That is helpful to simulate robot applications with sensors and actuators, dynamic and physical characteristics, with easy distribution and flexible control and communication systems. The framework can run on different platforms such as Windows, Linux, or macOS. Regarding all aspects, the software Unity

may be ideal for the development of micromouse 3D simulator.

This work aims to develop a micromouse simulator in Unity, to enhance the experience of competing teams in the micromouse. The simulator has features necessary for a quick transition to the real robot. The proposed simulator is based on HIL approach through serial communication with a real microcontroller, reflecting the performance of the programmed strategies given the constraints of the microcontroller utilized. Also, a typical robot model and the flood fill algorithm are available as an example to help beginners. The template robot is inspired by competition, with DC motors and encoders for odometry, distance sensors for wall detection, and a library that provides transparent and compatible access for development on the Arduino platform. The simulator and the library are available on the repository [2].

1.2 Objectives

The main objective is the development of a simulator software with HIL support for testing algorithms for the micromouse contest. To achieve it, the following sub-objectives are required:

- Develop simulation models for commonly utilized robot components, namely range sensors, encoders, DC motors and h-bridge motor controllers.
- Develop a communication protocol for interfacing a microcontroller with the simulator.
- Develop an Arduino-compatible for interacting with the robot components.
- Create environments that present challenges that require sensor data processing and calculating actuator commands
- Develop and analyse a search algorithm for the micromouse contest

1.3 Document outline

This document is divided in 6 chapters, which presents the micromouse contest and the type of simulators available, the tools that have been developed and used, and the results obtained from their application. The following is a brief description of the content of each chapter.

Chapter 1 is the introduction, which presents the micromouse contest, the idea of maze-solving algorithms and the purpose of simulation software.

Chapter 2 talks about different robotics competitions and their education potential for science, technology, engineering and mathematical areas, the types of simulators available for the micromouse contest, and an overview of maze-solving algorithms.

Chapter 3 presents the tools that have been used and developed to accomplish the goal of this work. It introduces Unity, a 3D development platform, created by Unity Technologies, a description of the developed computational model for the considered robot components, the kinematic model for the simulated robot and the proposed modification to the flood fill exploration algorithm. Finally, the Arduino compatible library that handles communication with the simulator and implements the modified algorithm is described.

Chapter 4 describes the architecture of the assembled system. It contains a description of the simulated robot model, the specification of the underlying serial communication protocol, and an overview of the expected text format for importing mazes into the simulation.

Chapter 5 presents the simulated environments that have been created. Three environments present challenges for the user to complete by interpreting sensor data and controlling the motors accordingly. The fourth environment is a representation of a micromouse contest. Next, this chapter presents a case-study for the modified flood fill algorithm in two mazes from official micromouse contests. Finally, a section describes the use of the developed simulator in the RoboSTEAM hackathon event.

Chapter 6 presents publications resulting from this work, the conclusions and future

works that further validate the proposed exploration algorithm and extend the simulator functionality.

Chapter 2

Related work

This chapter contextualizes the development of this project by giving an overview of different robotics competitions and their organizations. Regarding the micromouse contest, different types of simulators are presented, as well as some algorithms used for exploration and deicision making during the contest.

2.1 Robotics competitions

Mobile Robotics is an area of science that has a lot of educational potentials that is growing over the last few years and it can be seen as a tool to captivate students to explore STEM (Science, Technology, Engineering and Mathematics) areas.

Robotics competitions are a perfect way to motivate students to develop and apply knowledge gathered on lessons. There is a huge number of robotics competitions around the world organized by well known groups such as AAAI (Association for the Advancement of Artificial Intelligence) and IEEE (Institute for Electrical and Electronics Engineers) between others. Portuguese competitions have increased over the last years with events such as RoboParty (a robotic three-day camp organized at Universidade do Minho in Guimarães where school-age children learn electronics, mechanical engineering, and programming), the Portuguese Robotics Open (aims to promote science and technology to youngsters, teachers, researchers, and the general public), Bot Olympics (organized

at the University of Coimbra), Firefigher robot (organized at the Polytechnic of Leiria), the Micromouse (organized at the University of Trás-os-Montes e Alto Douro), a world well-known competition that exists for several years and still remains problems to solve, among the others. On the other hand, simulation is a powerful tool that allows students to create and test solutions without hardware and thus much cheaper solution. Simulation speeds up the development of algorithms since they can be validated before installed at the real robot.

2.2 Simulation software

Regarding the micromouse, there are two types of simulators: 2D simulators and 3D simulators. 2D simulators are used to test search algorithms and path planning procedures. These simulators are not concerned with the sensors or with the control of the robots. Typically, the simulators show the labyrinth in 2D, with the walls and the robot path performed. Algorithm verification works are more suitable for simulators rather than an actual maze, thus a lot of time could be saved [11]. They have great advantages; faster and easier to set up, allowing to quickly use multiple mazes. How the effectiveness of the search algorithms varies between different types of mazes simulators should be studied. As an example of these simulators, there is mms [12], that has the possibility of being used with any coding language. Micromouse Maze Simulator [13] acts as a server, so any client can connect to it and send requests to read walls and log the current exploration state.

3D simulators are already able to offer other advantages, such as robot control, the inclusion of sensors, and images more similar to the real ones. The simulators that simulate all the physical and electrical characteristics can simulate, for example, the skidding of the wheels, something that is very important in this competition. Within these categories, the Virtual 3D micromouse [14] has real mechanical and electrical characteristics similar to the real micromouse, and users can modify the size of 3D micromouse, infrared properties, and motor speed. In other words, it is possible to configure the properties of the infrared

sensor and also to change engine characteristics. In [15], a HIL simulator tool is presented where the simulated robot is controlled by the same microcontroller used by the robot. In this way, the developed algorithms are tested and validated with the limitations and constraints presented in the real hardware, such as memory and processing capabilities. The robot dynamics, the slippage of the wheels, the friction, and the 3D visualization are present in the simulator.

The simulators have reached a level that is almost not necessary to have a real maze to prepare a team.

2.3 Maze solving algorithms

Since the beginning of the micromouse contest, in the mid-70s, researchers have investigated the best method of finding the fastest path through the maze. Several algorithms were implemented over time, and initially, the most used and one of the most popular algorithms, especially in beginners, is "Wall Follower" [16], which has two variants, "Left Wall Followe" ("LWF") and "Right Wall Follower" ("RWF"). The main idea of this algorithm is to follow the left/right wall until it finds the destination. The great advantage of this algorithm is that it is easy to implement, but the problem is that in many cases could not reach its destination, entering in a loop.

In order to solve the problem of not always find the destination, the "Pledge Algorithm" was implemented, which is a modified version of the "Left Wall Follower" algorithm [17]. This algorithm, when it enters a loop, can jump to a new wall. It may take a while to find the destination, but it always does.

Another method introduced in order to reduce the time to explore the maze was the "Partition-central Algorithm" [18]. This method divides the maze into 12 partitions, and depending on the direction of the robot and the location in each partition, the exploration rules are changed in order to optimize the process of exploring the maze.

Later, in 2016 Bienias [19] joined two labyrinth exploration algorithms; "Wall follower" and "Trémaux's algorithm." The algorithm involves two phases: first, the whole maze is

explored in an ordered way, and then, the shortest possible way out is determined. Thus, this algorithm was able to find better solutions more quickly.

The most popular method today is the Flood fill algorithm. The Flood fill algorithm uses the concept of water flows from a high point to a lower point [20] [16]. The Flood fill algorithm [4] assumes at the start that there are no walls in the maze and then uses Lee's algorithm[21] to find its way to the target. This concept is used, giving each cell a value that represents the distance to the destination cell. The destination cell is assigned a value of 0.

Several modifications have been made to this algorithm, and one of the most popular is the modified Flood fill algorithm. This algorithm has the great advantage of not having to redo the algorithm from the beginning each time a new cell is reached. It only updates the neighboring cells through a recursive process [22]. Aggarwal, in 2013[23], integrated "Look Ahead technique" and "Directional algorithm" with the Flood fill algorithm in order to optimize the distance traveled and the time taken. This method is called "Iterative Flood-Ahead Algorithm".

In order to have some extra information to the Flood fill Algorithm, Cai, in 2014 [24], introduced the "ET-Floodfill" algorithm. This algorithm assumes that unknown walls have an extra penalty. That is, there is a value for the uncertainty of the existence of walls. With this modification, this algorithm behaves better than Flood fill by making fewer turns.

In 2017, Tjiharjadi [25] joined the A* algorithm with the Flood fill algorithm. It uses two algorithms at the same time compares and optimizes the solution in order to remove the best of each of the methods.

Chapter 3

Tools

This chapter presents the tools used to develop this project, some of which were developed for the purposes of this work.

It contains an overview of the development platform that the simulator is based upon and the kinematic model considered for the robot.

Regarding the tools developed for this work, there are sections defining the computational models for the components, the proposed maze exploration algorithm and, finally, the Arduino compatible library responsible for interacting with the simulator and implementing the exploration algorithm.

3.1 Unity development platform

Unity is a cross-platform, real-time 3D development platform developed by Unity Technologies. The platform supports three-dimensional and two-dimensional environments, including virtual reality and augmented reality projects. Unity's development platform is available for Windows, Mac OS and Linux and deploys projects for over 20 other platforms, including Android, Playstation, Xbox, Nintendo Switch, augmented reality (AR) and virtual reality (VR) platforms [26].

Unity provides built-in physics engines to help simulate physics interactions, such as movement and collisions. For object-oriented projects, Nvidia PhysX is used for 3D

projects, and Box2D is used for 2D projects. Alternatively, projects that use Unity's Data-Oriented Technology Stack (DOTS) have access to Unity Physics and an implementation of Havok Physics [27].

This work makes use of Unity version 2020.1.0f1 to create a 3D simulated environment for mobile robotics, using the built-in physics engine PhysX 4.1. The following sections explain Unity's functionality relevant to this work.

3.1.1 Unity Editor interface

The development of a project using Unity happens within Unity's Editor. The most common editor windows are shown in Figure 3.1.

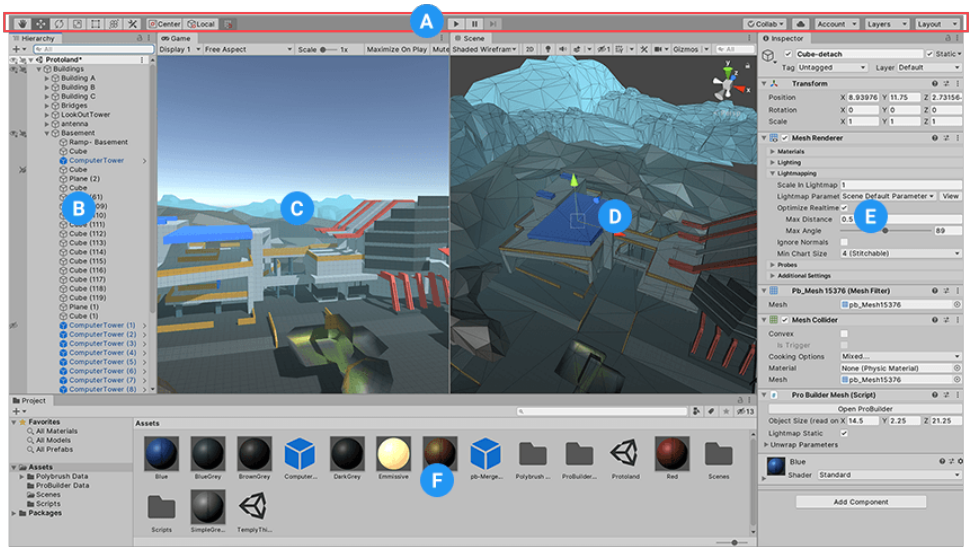


Figure 3.1: Common Unity editor windows, where the labeled windows have the following role: Ⓐ Toolbar. Provides tools for manipulating `Scenes` and `GameObjects`. Controls program execution with the play, pause and step buttons; Ⓑ Hierarchy Window: Textual representation of the `GameObjects` hierarchy in the `Scene`; Ⓒ Game View: Visualization of what the game will look like when executed; Ⓓ Scene View: Allows visual navigation and editing of the `Scenes`; Ⓔ Inspector Window: Allows editing and visualization of the selected `GameObjects`'s properties. Ⓕ Project Window: Displays assets available in the project. Extracted from [28].

3.1.2 Unity's basic concepts

`Scenes` contain the menus and the environment of the application. The menus are the labels, buttons, dropdown menus, text fields and other similar user-interface elements. The environment is composed of visual elements and non-visual elements. Visual elements, light sources, decorations and rigid bodies. Non-visual elements include the application camera position and user-defined components, such as the actuator and sensor models described in Section 3.2. Multiple `Scenes` can be created, defining different environments, but they need not be active at the same time.

Each element contained in a `Scenes` is a `GameObjects`, the most important concept in Unity [29]. `GameObjects` don't do anything on their own; instead, they act as a collection of Components, the units that define state and behaviour for a `GameObjects`. At any given time, they can be active or inactive. Their state influence the execution of methods for it's component's scripts defined in Section 3.1.3.

Some Components that provide common functionality for `GameObjects` and are predefined by Unity, for example, `Transform`, `Mesh Filter`, `Mesh Renderer` and others, related to the physics simulation defined in Section 3.1.4 [30]. `Mesh Filter` and `Mesh Renderer` are responsible for visual representation of `GameObjects`. Application-specific components can be defined by the user by means of Scripting

The `Transform` stores a `GameObjects`'s position, orientation, scale and parent. `GameObjects` can be placed in a hierarchy by assigning them a parent. By parenting, all of it's movement, rotation and scale will change accordingly to it's parent. If the `Transform` has no parent, it is called the root of the hierarchy.

3.1.3 Scripting

Scripting is done with `C#`, a general-purpose, multi-paradigm, programming language. `Scripts` are connected to the internal workings of Unity by defining a class which inherits from the built-in `MonoBehaviour` class [31]. Then, by attaching a Script to a `GameObjects`, an instance of a Component is created. The state of the `GameObjects` is defined by the

classes' fields and the behaviour is defined by the classes' methods.

By implementing certain specially-named methods, the user has control over the life cycle and behaviour of the Component defined by a Script. The life cycle of `Scripts` can be divided into separate phases. The initialization phase is executed once, then the physics, logic and render phases are looped over during normal execution, and the decommissioning phase takes place right before the application exits. The code for all `Scripts` present in the active `Scenes` are executed [32].

The Initialization phase is responsible for initialization of fields and the assignment of references between objects. This phase is executed when the application starts running or when the object becomes active. The method names that must be defined to set the behaviour in this phase are `Awake`, `OnEnable` and `Start`.

The Physics Update phase is controlled by the `FixedUpdate` method. Physics calculations and updates occur immediately after this phase. Generally, this method is called multiple times before the the Update and Render phases occur. The number of times it is executed depends on the `Physics Timestep` setting, and the frame-rate the application is running.

In the Update phase, the `Update` method is executed. It is used to process user input, such as keyboard key presses and mouse clicks. Calculations unrelated to the physics engine also happen in this phase. It is executed once per frame.

The Render phase is responsible for drawing the content of the `Scenes`. Unity handles drawing the `GameObjects` on it's own. Custom geometry can be draw with GL, a low-level graphics library similar to OpenGL, in the `OnRenderObject` method.

The decommissioning phase is executed once, before the application quits, or when the object becomes inactive. The methods of interest are named `OnApplicationQuit` and `OnDisable`.

In both the Update and `FixedUpdate` methods, the scripts may access the Time.deltaTime variable, defined in the `UnityEngine` namespace. Within the `Update` method, the variable resolves to the actual time passed since the last frame, whereas within the `FixedUpdate`, it resolves to the physics timestep.

3.1.4 PhysX physics engine

PhysX is an open-source physics engine developed by Nvidia. The role of the physics engine is to simulate a physics system, by calculating position and velocity of objects, how they accelerate and respond to interactions such as collisions and forces.

Unity integrates this engine into it's environment by defining dedicated Components, namely `Rigidbodys` [33], `Colliders` [34] and `Joints` [35].

The `Rigidbody Component` defines the mass, drag and angular drag of the rigid body it is attached to. It also defines whether the `GameObjects` should be affected by gravity and whether it is currently set as kinematic, meaning it should remain static. It also defines settings related to the physics engine calculation methods, namely as interpolation and collision detection methods.

Assets of the type `Physic Material` define coefficients for dynamic friction, static friction and bounciness for each object. Two other parameters, Friction Combine and Bounce Combine, define how the coefficients of two objects are combined: Maximum, Minimum, Average or Multiply. For every collision between two objects, a resulting coefficient is calculated for the interaction. In case they define different combination settings, they take priority as follows: Average $<$ Minimum $<$ Multiply $<$ Maximum. The coefficient values may not present behaviour close to the real world, as they are tuned for performance and simulation stability.

`Colliders` define the shape and `Physic Material` properties of objects for the purposes of the physics simulation. The shape defined by the `Collider` is usually simpler than the visual representation of the object. The rationale is that simulating complex meshes can be computationally expensive, and so `Colliders` provide approximated, simpler shapes. `Box Colliders`, `Capsule Colliders` and `Sphere Colliders` computationally effective definitions of simple shape's boundaries, whilst `Mesh Colliders` reflect the exact shape of the visible mesh, defined by the `Mesh Renderer Component`. Naturally, it is more computationally expensive.

`Box Colliders`' shapes are defined by their center position and and the size of each

axis. `Capsule Colliders`' shapes are defined by their center, radius, height and direction. `Sphere Colliders`' shapes are defined by their center and radius. The size and position of all `Colliders` are relative to the `GameObjects`'s `Transform`.

`Colliders` can be attached to `GameObjects`'s without `Rigidbodys`, in which case they are called static `Colliders`. These are useful for representing floors or walls. They may also be configured as triggers, so they don't collide with `Rigidbodys`, and instead send `OnTriggerEnter`, `OnTriggerExit` and `OnTriggerStay` messages when a `Rigidbody` enters or exists its boundaries, in the form of method calls. In order to respond to these messages, a Script must define a method with their respective name.

The `Joint Components` define connections between two `Rigidbodys` or between a `Rigidbody` and a fixed point in space. There are multiple joint types, namely `Fixed Joint`, `Hinge Joint`, `Spring Joint`, `Character Joint` and `Configurable Joint`. `Hinge Joints` restricts the movement of a `Rigidbody`, allowing it to rotate about one defined axis, exclusively. The `Configurable Joint` allows the user to specify which degrees of freedom are restricted.

Besides simulating physics systems, PhysX allows the user to query the state of the simulation [36]. For example, it is possible to retrieve information about the velocity of a `Rigidbody`, and to perform ray cast operations, which calculates which objects are intercepted by a ray starting at a given position, orientation and length.

3.1.5 Prefabs and ScriptableObjects

The prefab system allow a `GameObjects` template to be created, modified and stored as a reusable asset, complete with all Components and property values [37].

Likewise, `Scriptable Objects`, act as data-containers, independent of class instances. `Scriptable Objects` are by inheriting from the `ScriptableObject` built-in class [38]. Then, they can be created through the unity Project Window by the right click context menu, accessible by selecting `Create` after right-clicking, and edited in the Inspector Window.

3.2 Computational models for robot components

This section presents details about the mathematical models defined for the robot components. Each component type has a set of parameters that represent an specific component of that type. For example, there are values defined for two different Direct current (DC) motors and one range sensor. The models defined for the batteries, h-bridges and encoders are considered "ideal" models and require further refinement.

The battery component is simply a fixed voltage source. The other components are detailed in the following subsections.

3.2.1 Direct current motor

The motor-wheel electromechanical system converts electrical input into mechanical output that results in a shaft being rotated [39]. It is modelled by the diagram in Figure 3.2. In the mechanical part, the wheel is represented by a load with moment of inertia I, which rotates at angular velocity ω and is acted upon by a load torque T. The electrical part describes the armature circuit for a brushed, direct current dc motor. It consists of an inductance L_a and a resistance R_a, both with fixed values, connected in series. There is an input voltage v_a opposed by the back e.m.f v_b, resulting in current i_a flowing. Equation 3.1 is used as a starting point for describing the behaviour of the motor and establishing the relation between its parameters.

It is assumed that the values for parameters L_a, damping parameter λ, stall torque T_stall, stall current i_stall, no-load speed ω_nl and no-load current i_nl for a given input voltage $V_t extrated$ can either be measured or are stated in the motor datasheet, and the parameters R_a and K_t should be calculated.

$$v_\mathrm{a} = R_\mathrm{a} i_\mathrm{a} + L_\mathrm{a} \frac{\mathrm{d}i_\mathrm{a}}{\mathrm{d}t} + v_\mathrm{b} \tag{3.1}$$

The back e.m.f is proportional to ω and the back e.m.f constant K_v, as given by Equation 3.2. K_v depends on construction aspects of the motor.

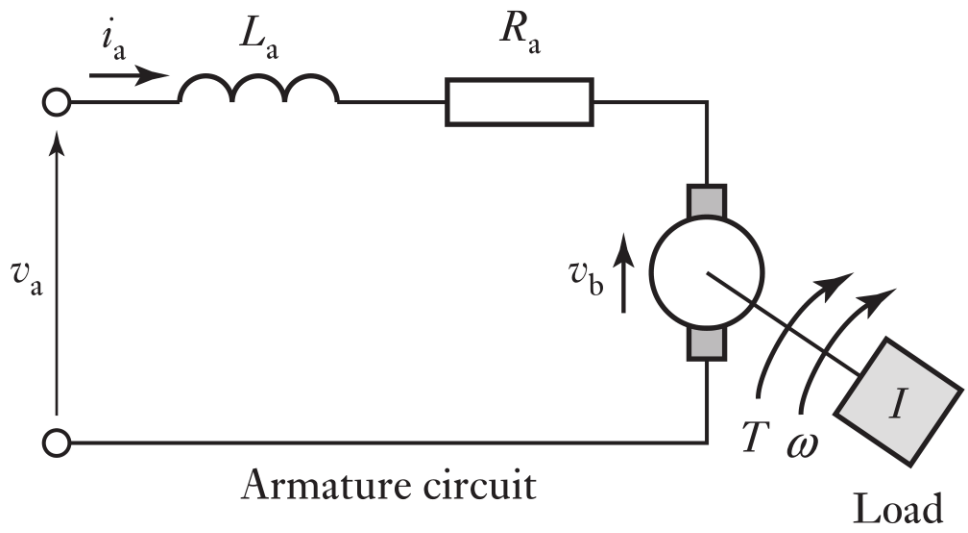


Figure 3.2: Schematic diagram for a DC motor connected to a load. Adapted from [39].

$$v_{\mathrm{b}} = K_{\mathrm{v}} \cdot \omega \tag{3.2}$$

When the the motor is in steady-state, that is, $\frac{\mathrm{d}i_{\mathrm{a}}}{\mathrm{d}t} = 0$, the effect of the inductance is neglected and Equation 3.1 leads to Equation 3.3. It can then be rearranged to obtain R_{a} from V_{rated}, K_{t}, ω_{nl} and i_{nl}.

$$v_{\mathrm{a}} = R_{\mathrm{a}} i_{\mathrm{a}} + v_{\mathrm{b}} \tag{3.3}$$

$$R_{\mathrm{a}} = \frac{v_{\mathrm{a}} - K_{\mathrm{t}}\omega}{i_{\mathrm{a}}} \tag{3.4}$$

$$R_{\mathrm{a}} = \frac{V_{\mathrm{rated}} - K_{\mathrm{t}}\omega_{\mathrm{nl}}}{i_{\mathrm{nl}}} \tag{3.5}$$

$$K_{\mathrm{t}} \equiv K_{\mathrm{v}} \tag{3.6}$$

The developed torque T_d is proportional to i_{a} by a factor of K_t, which is the torque constant. Rearranging the equation, the torque constant can be determined by T_{stall} and

i_{stall}.

$$T_{\text{d}} = K_{\text{t}} \cdot i_{\text{a}} \tag{3.7}$$

$$K_{\text{t}} = \frac{T_{\text{stall}}}{i_{\text{stall}}} \tag{3.8}$$

The torque T applied to the load is the developed torque T_{d} minus the damping torque T_{c}, where λ is the damping parameter, while the effect of the load inertia is modelled by the physics engine itself.

$$T = T_{\text{d}} - T_{\text{c}} \tag{3.9}$$

$$T_{\text{c}} = \lambda\omega \tag{3.10}$$

In the software side, the torque applied by the motor must be determined at every time step. Since it is equivalent to the current multiplied by a constant, it follows that the current must be calculated. As it is a real-time simulation, the calculations must depend only in variables calculated in previous time steps. Therefore, a backward difference of first order was used [40].

$$\frac{\text{d}f(x)}{\text{d}x} \approx \frac{f(x) - f(x-h)}{h} \tag{3.11}$$

$$f(x_0) = y_0 \tag{3.12}$$

By applying the approximation given by Equation 3.11 to Equation 3.1, it results in Equation 3.13, where $i_{\text{a}t}$ is the armature current for the present time step and $i_{\text{a}t-1}$ is the armature current for the previous time step.

$$v_{\text{a}} = R_{\text{a}} i_{\text{a}t} + L_{\text{a}} \frac{i_{\text{a}t} - i_{\text{a}t-1}}{\Delta t} + K_{\text{t}}\omega(t) \tag{3.13}$$

$$i_{\mathrm{a}}(0) = 0\,\mathrm{A} \tag{3.14}$$

It can then be rearranged into Equation 3.15.

$$i_{\mathrm{a}t} = \frac{L_{\mathrm{a}} i_{\mathrm{a}t-1} + V_{\mathrm{a}}\Delta t - K_{\mathrm{t}}\omega}{L_{\mathrm{a}} + R_{\mathrm{a}}\Delta t} \tag{3.15}$$

The motor has an extended shaft before the reduction gear box, in which the encoders may be mounted. Since it rotates faster than the shaft attached to the wheel, it effectively increases the encoder resolution.

Table 3.1 presents the calculated and intrinsic parameters for two DC motors. The values for K_{t}, R_{a} were calculated with Equations 3.8 and 3.5, L_{a} was assumed to be 0.001, and the λ parameter was set to 0, as there weren't performed measurements for it's estimation. The gear ratio and rated voltage parameters, as well as other parameters needed for the previous calculations, were given in their datasheets [41].

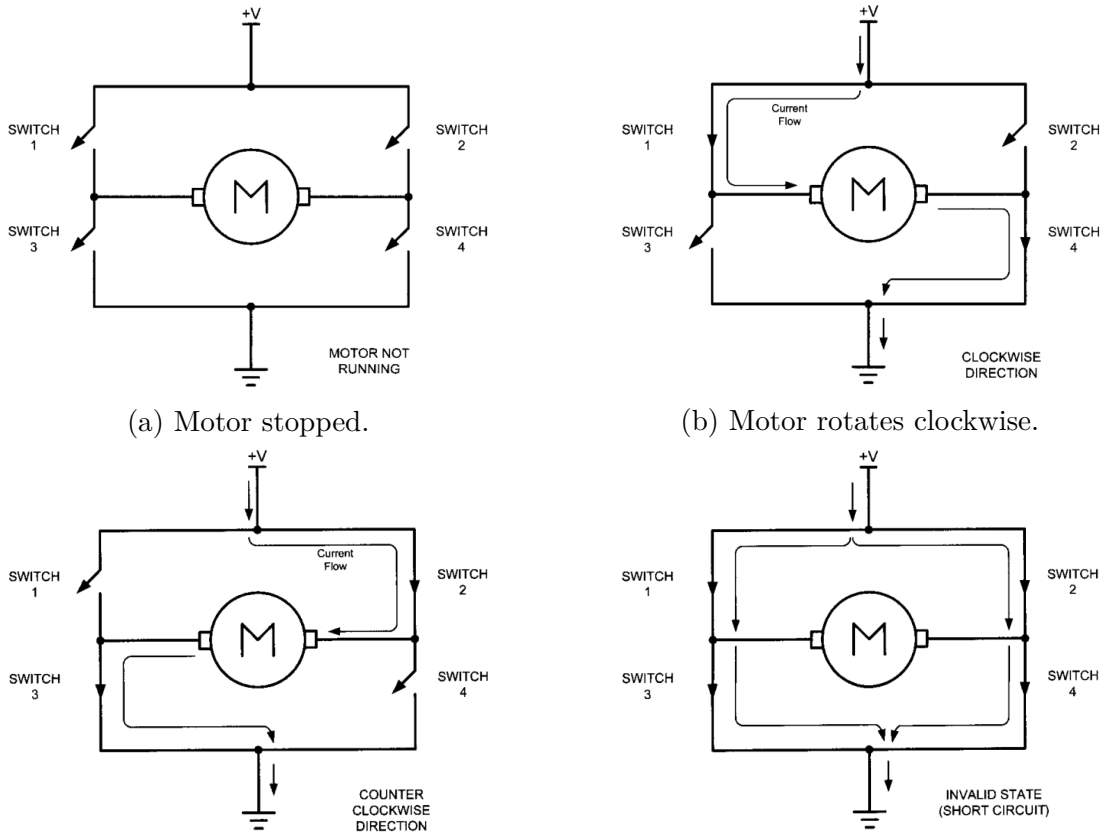
Table 3.1: Parameters for the motor models.

Motor	$V_{\mathbf{rated}}$ **(V)**	**Gear ratio**	$K_{\mathbf{t}}$ **(Nm/A)**	$R_{\mathbf{a}}$ **(Ω)**	$L_{\mathbf{a}}$ **(H)**
Pololu 30:1 LP	6	29.86 : 1	0.079025	16.66667	0.001
Pololu 50:1 LP	6	51.45 : 1	0.1110591	16.66667	0.001

3.2.2 H-bridge

H-bridges are electronic devices capable of controlling the the polarity and intensity of the voltage applied to a load. It is commonly used for bidirectional motor speed control [42]. A schematic diagram of an h-bridge and some logical states are shown in Figure 3.3. It contains four switches, SW1, SW2, SW3 and SW4, which change the polarity of the current going through the motor. Table 3.2 summarizes the possible bridge states and their effects, which can make the motor stop, rotate or unintentional short-circuit [43].

H-bridges may operate in two drive modes: sign-magnitude or locked anti-phase [42]. In sign magnitude, one wire controls the motor direction, and another sends a Pulse-width



(a) Motor stopped.

(b) Motor rotates clockwise.

(c) Motor rotates counter clockwise.

(d) Short-circuit.

Figure 3.3: H-bridge schematic diagram and its possible states. Extracted from [43].

modulation (PWM) signal which toggles the h-bridge on and off. In locked anti-phase drive, a single wire with PWM signal is used for switching between forward and backward states. Tables 3.3 and 3.4 summarize some inputs and the resulting motor motion.

Table 3.2: Description of the possible h-bridge states.

State	**SW1**	**SW2**	**SW3**	**SW4**	**Outcome**
1	Closed	Open	Open	Closed	Motor rotates clockwise
2	Open	Closed	Closed	Open	Motor rotates counter clock wise
3	Closed	Open	Closed	Open	Motor brakes
4	Open	Closed	Open	Closed	
5	Closed	Closed	x	x	Short-circuit
6	x	x	Closed	Closed	

Table 3.3: Sign-magnitude drive.

Sign	**Duty cycle (%)**	**Outcome**
1	100	Forward full speed
1	50	Forward half speed
1	0	Brake
0	0	Brake
0	50	Backward half speed
0	100	Backward full speed

Table 3.4: Locked anti-phase drive.

Duty cycle (%)	**Outcome**
100	Forward full speed
75	Forward half speed
50	Stop
25	Backward half speed
0	Backward full speed

$$V_{\text{out}} = D \cdot V_{\text{in}} \tag{3.16}$$

In practice, the switches are implemented with either Bipolar junction transistors (BJTs) or Field-effect transistors (FETs). There are intrinsic voltage drops across these

components, which haven't been modeled. For BJTs, the voltage drop corresponds to its specified collector-emitter saturation voltage $V_{\mathrm{CE(sat)}}$ and in FETs, the drop is $V = I \cdot R_{\mathrm{DS(on)}}$, where I is the current flowing and $R_{\mathrm{DS(on)}}$ is the drain-source resistance.

The developed model operates as a sign-magnitude drive, whose duty cycle is set with a value in the range $[-100\%; 100\%]$. The serial interface assumes that the h-bridge operates with a resolution of 10 bits for the magnitude and one bit for direction. The microcontroller sends such commands across 2 bytes, representing integers in the range $[-1023; 1023]$, which are mapped into valid duty cycle values.

3.2.3 Incremental encoders

Incremental rotary encoders are sensors that issue pulses when changes of angular position are detected. Quadrature encoders operate with two output channels, A and B, are able to detect both change in position and the direction of movement. The output signals are square waves, and the signal of each channel has a phase difference of 90°.

Figure 3.4 illustrates the signal outputs of a quadrature encoder. The direction of rotation is inferred depending on the current and previous output of channels A and B. The sequence of output values differ from one direction to the other.

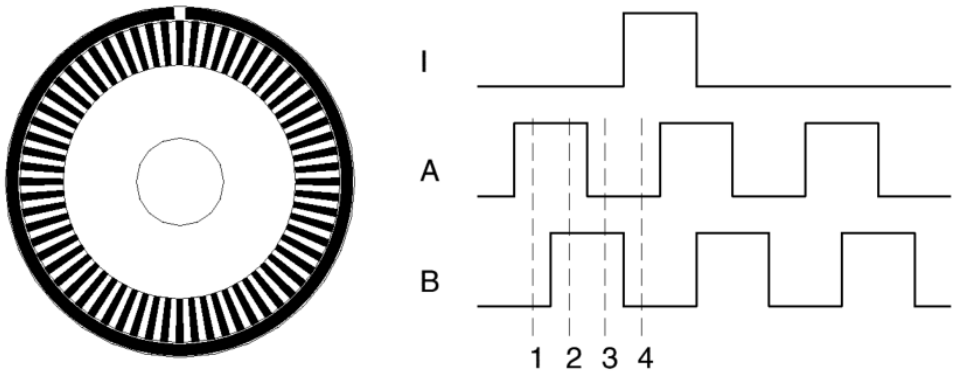


Figure 3.4: Signal output for quadrature encoders. Extracted from [44].

The amount of pulses issued in a complete revolution, CPR, vary depending on the encoder. The encoders presented in [45] issue 12 pulses in a revolution. The change in angle represented by one pulse is given by θ_p, defined in Equation 3.17.

Table 3.5: Sequence of states for quadrature encoders. From [44].

State	**Channel A**	**Channel B**
1	high	low
2	high	high
3	low	high
4	low	low

$$\theta_p = \frac{2\pi}{CPR} \tag{3.17}$$

If the wheel diameter d_{wheel} is known, the encoder can be used to estimate the ground contact speed v of the wheel. By multiplying θ_{p} by π and the number of pulses N issued over a period t, the wheel's ground contact speed is given by Equation 3.18.

$$v = \frac{\pi N d_{\text{wheel}}}{t} \tag{3.18}$$

The developed simulator works in discrete time steps, and the change in angle between consecutive time steps is given in Equation 3.19 by $\Delta\theta$, where ω is the angular speed and Δt is the time step. The amount of pulses reported by the encoder model at every time step is calculated according to Algorithm 1. When the program starts executing, θ_{acc} is initialized to 0. Then, at every time step, the pulse counter variable N is set to 0 and the physics engine is queried for the current angular speed of the encoder, and the change in $\Delta\theta$ is calculated for that time step. This value is added to the accumulator variable θ_{acc}. Then, two loops are used to calculate how many pulses θ_{acc} represents, by increasing or decreasing N, depending on whether θ_{acc} holds a value greater than θ_{p} or lesser than $-\theta_{\text{p}}$. Finally, the rest of the program is notified of N, and the current value stored in θ_{acc} is used in the following iterations.

$$\Delta\theta = \omega\Delta t \tag{3.19}$$

Algorithm 1: Calculates the number of pulses issued by the encoder.

Input: The wheel angular speed ω, the accumulated angle θ_{acc}, such that $|\theta_{\text{acc}}| < \theta_p$

Result: The number N of pulses generated and θ_{acc} for the next iteration

$N \leftarrow 0$
$\Delta\theta \leftarrow \omega\Delta t$
$\theta_{\text{acc}} = \theta_{\text{acc}} + \Delta\theta$
while $\theta_{\text{acc}} \geq \theta_{\text{p}}$ **do**
    $N \leftarrow N + 1$
    $\theta_{\text{acc}} = \theta_{\text{acc}} - \theta_{\text{p}}$
while $\theta_{\text{acc}} \leq -\theta_{\text{p}}$ **do**
    $N \leftarrow N - 1$
    $\theta_{\text{acc}} = \theta_{\text{acc}} + \theta_{\text{p}}$
return N, θ_{acc}

3.2.4 Time-of-flight range sensor

Time-of-flight (ToF) range finders are sensors that measure the range to nearby objects. The goal of this section is to present the procedures used to develop a sensor model for the VL53L0X range sensor [46].

The measurement model $p(z_t \mid x_t, m)$ for range finders is a mixture of four densities, representing different types of measurement error [47]. Consider z_t^{k*} as the ground-truth range, z_t^k as the measured range and $z_{\max}$ is the maximum allowable range value specified for the sensor. As observed in [48], the VL53L0X distance sensor also seems to have a minimum working distance $z_{\min}$ of about 3 cm, which will also be included in the model. The x_t and m variables represent the position of the robot and the map of the environment, respectively.

- A narrow Gaussian distribution denoted by p_{hit}, with mean z_t^{k*} and standard deviation σ_{hit} representing a small measurement error due to limited sensor resolution, atmospheric effects and so on. In practice, the values of this distribution are limited to the interval $[z_{\min}, z_{\max}]$.

- An exponential distribution denoted by p_{short} representing the likelihood of a moving obstacle being detected, causing an unexpected short sensor reading. A common

source of these kind of measurements is people moving in the environment.

- A pseudo-density $p_{\max}$, indicating the sensor failed to detect an object and returned its maximum allowable reading value $z_{\max}$.

- A uniform distribution p_{rand} representing random, unexplained measurements, possible due to cross-talk between different sensors.

The complete sensor model and its components are depicted in Figure 3.5.

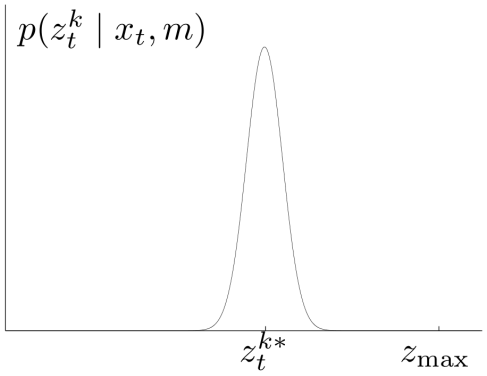
In order to obtain a mathematical model for the VL53L0X range sensor, 500 samples were taken with it facing a white wall, at distances of 5 cm, 10 cm, 20 cm, 30 cm and 40 cm. Figure 3.6 shows the histograms of the frequency of measurements alongside a Gaussian distribution representing each set of samples.

It was observed that all readings fit under the p_{hit} distribution, as there were no moving obstacles for p_{short}, no uniformly distributed unexpected readings for p_{rand}, and no $z_{\max}$ readings due to failed readings described by $p_{\max}$.

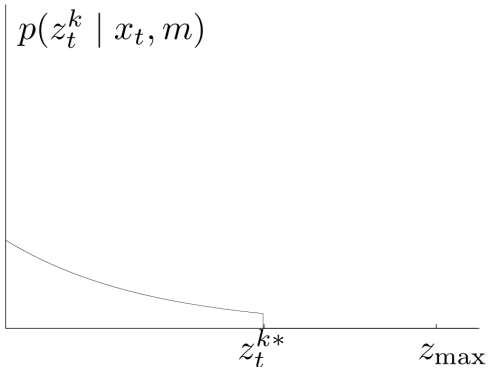
Some metrics for the readings distribution are shown in Table 3.6. Clearly, there is a mismatch between the actual distance and the mean of the reading distribution. Such offset could be corrected by properly calibrating the sensors, as described in calibrating sensors [49]. Also, it shows that the standard deviation of the measurements varies depending on the obstacle distance.

The coefficient of variation c_{v} metric, defined in Equation 3.20, was chosen as the noise parameter for the sensor model. The noise present in every reading is sampled from a normal distribution, defined in Equation 3.22, with mean z_t^{k*}, corresponding to the true distance between the sensor and the obstacle, and standard deviation $\sigma = c_{\text{v}} \cdot z_t^{k*}$. The sensor model probability distribution is given by Equation 3.21, where η is a normalization parameter given by Equation 3.23.

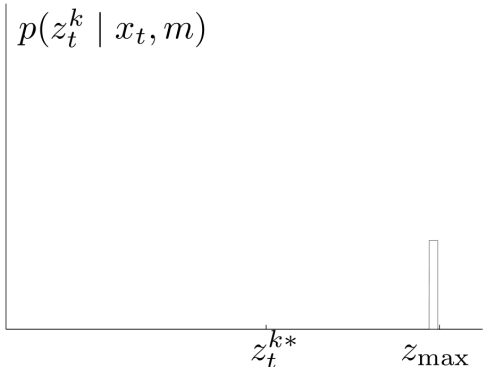
$$c_{\text{v}} = \frac{\sigma}{\mu} \tag{3.20}$$



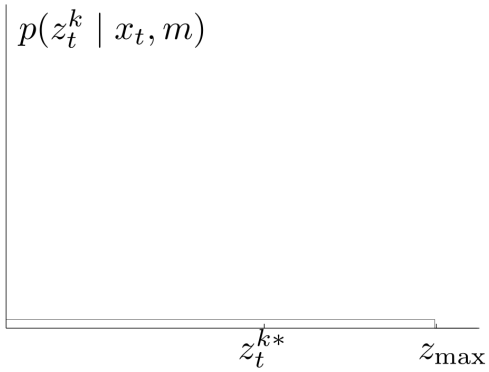
(a) Gaussian distribution p_{hit} component.



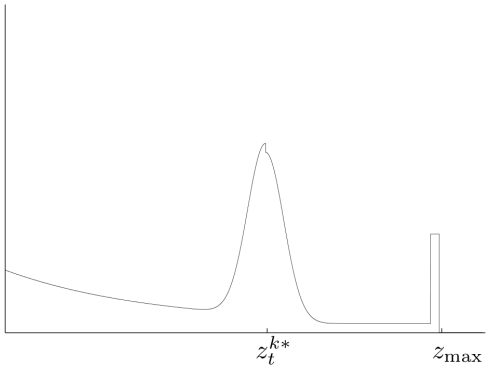
(b) Exponential distribution p_{short} component.



(c) Uniform distribution $p_{\max}$ component.



(d) Uniform distribution p_{rand} component.



(e) Complete range sensor model.

Figure 3.5: Components of the range finder sensor model proposed by [47].

$$p(z_t^k \mid x_t, m) = \begin{cases} \eta \mathcal{N}(z_t^k;\, z_t^{k*},\, c_{\mathrm{v}} \cdot z_t^{k*}) & \text{if } 0 \leq z_t^k \leq z_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{3.21}$$

$$\mathcal{N}\left(z_t^k;\, z_t^{k*},\, \sigma_{\mathrm{hit}}^2\right) = \frac{1}{\sqrt{2\pi\sigma_{\mathrm{hit}}^2}} e^{-\frac{1}{2}\frac{(z_t^k - z_t^{k*})^2}{\sigma_{\mathrm{hit}}^2}} \tag{3.22}$$

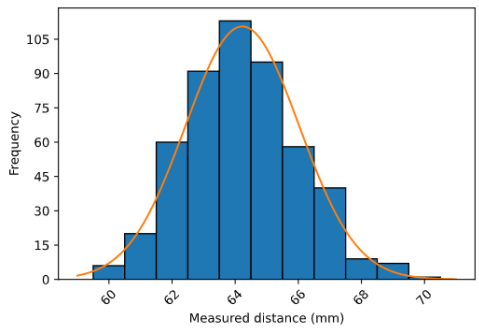
$$\eta = \left(\int_0^{z_{\max}} \mathcal{N}(z_t^k;\, z_t^{k*},\, c_{\mathrm{v}} \cdot z_t^{k*})\, \mathrm{d}z_t^k\right)^{-1} \tag{3.23}$$

Besides the parameters related to the distribution of measurements, one additional parameter must be defined, regarding the sensor's update rate, which indicates how long does it take to perform a measurement. And so, the range sensor model developed has four parameters: minimum range, maximum range, noise coefficient of variation, and sampling time. The parameters for the VL53L0X sensor are summarized in Table 3.8. The minimum range is set to 3 cm, as per the observation made in [48]. The maximum range is set to 1.2 m, which is the expected maximum in the default operation mode, as given in the product's datasheet [50]. The update rate is set to 30 ms, which also corresponds to the default mode of operation. As for the coefficient of variation, the chosen value corresponds to the maximum coefficient of variation across the data sets collected, making the model readings be, at most, as accurate as its physical counterpart, as Table 3.7 shows.

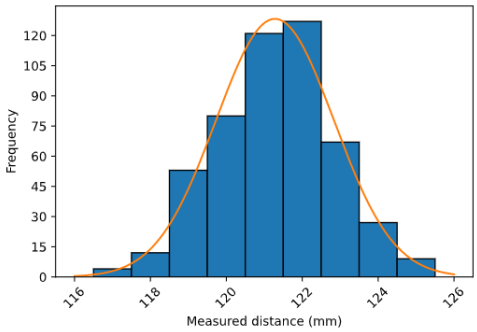
Table 3.6: Vl53L0X measurement metrics.

Actual Distance (mm)	**Samples**	**Mean (mm)**	**Std. dev (mm)**	$c_{\mathbf{v}}$
50	500	64.226	1.803727	0.028084
100	500	121.28	1.555132	0.012823
200	500	229.83	1.615649	0.00703
300	500	327.478	1.727276	0.005274
400	500	434.34	2.043711	0.004705

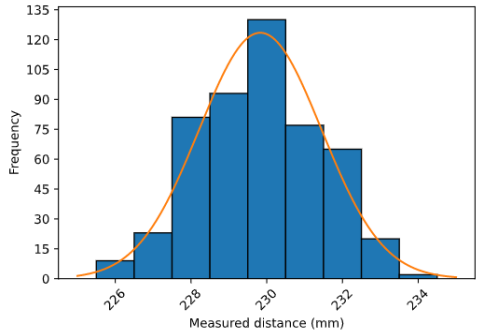
The output for the VL53L0X sensor is a 16-bit, unsigned integer value, which indicates the measured range. Every 30 ms, when the sensor performs a new reading, a ray cast



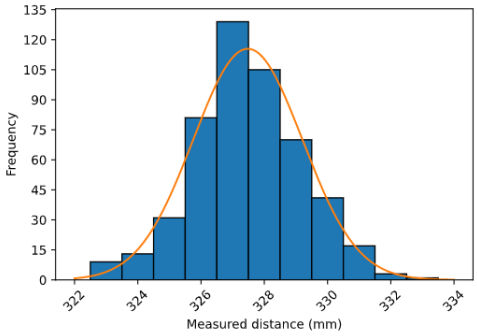
(a) $d = 50\,\text{mm}$.



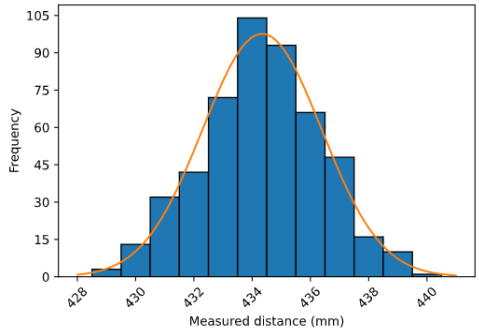
(b) $d = 100\,\text{mm}$.



(c) $d = 200\,\text{mm}$.



(d) $d = 300\,\text{mm}$.



(e) $d = 400\,\text{mm}$.

Figure 3.6: Measurements performed for the VL53L0X range sensor.

Table 3.7: Comparison between the standard deviation of the model and the sampled measurements.

Actual distance (mm)	**Sample std. dev. (mm)**	**Model std. dev. (mm)**
50	1.803727	1.803727
100	1.555132	3.406035
200	1.615649	6.45456
300	1.727276	9.196912
400	2.043711	12.19803

Table 3.8: Parameters for the VL53L0X range sensor model.

VL53L0X range sensor model	
Minimum range	3 cm
Maximum range	1.2 m
Noise coefficient of variation	0.028084
Measurement time	30 ms

operation is performed, mapped into a 16-bit value and transmitted to the microcontroller, following the communication protocol described in Section 4.2.

3.3 Differential drive kinematics

The differential drive kinematic model describes the motion of the simulated robot. This model consists of two independently-controlled wheels mounted in a common axis [51], and is presented in Figure 3.7.

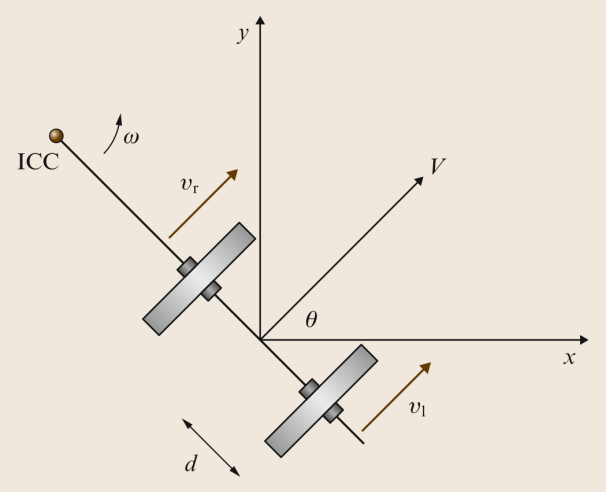


Figure 3.7: Differential drive kinematics. Extracted from [51].

If both wheels keep in contact with the ground, then the robot will describe a circular motion about an Instantaneous center of curvature (ICC). In the particular case in which the ground contact speed for both wheels is the same, the robot will move in a straight line.

If the left wheel and right wheel ground contact speed, denoted by v_{l} and v_{r}, respectively, and the wheels are separated by distance $2d$, then Equations 3.25 and 3.24 hold true. Rearranging them leads to Equation 3.26, which defines the angular speed ω in which the robot rotates about the ICC center and R the distance between the ICC and the robot center, according to Equation 3.27.

$$v_{\mathrm{r}} = \omega(R + d) \tag{3.24}$$

$$v_{\mathrm{l}} = \omega(R - d) \tag{3.25}$$

$$\omega = \frac{v_{\mathrm{r}} - v_{\mathrm{l}}}{2d} \tag{3.26}$$

$$R = d\frac{v_{\mathrm{r}} + v_{\mathrm{l}}}{v_{\mathrm{r}} - v_{\mathrm{l}}} \tag{3.27}$$

The instantaneous velocity of the point midway between the robot wheels is given by Equation 3.28.

$$V = \frac{v_{\mathrm{r}} + v_{\mathrm{l}}}{2} \tag{3.28}$$

In order to estimate the position of the robot in a fixed reference frame, Equations 3.29, 3.30 and 3.31 are used.

$$x(t) = \int V(t)\cos[\theta(t)]\mathrm{d}t \tag{3.29}$$

$$y(t) = \int V(t)\sin[\theta(t)]\mathrm{d}t \tag{3.30}$$

$$\theta(t) = \int \omega(t)\mathrm{d}t \tag{3.31}$$

3.4 Maze solving algorithm

The exploration of the maze is guided by a modified version of the flood fill algorithm [52]. The proposed approach has been analyzed in the following subsections.

Under the assumption that the robot has a simplified movement scheme, where every action takes the same amount of time and the allowed movements are limited to:

- Move forward once cell;
- Turn 90° clockwise;
- Turn 90° counter-clockwise.

Unlike the traditional flood fill implementation, the modified flood fill movement set is directly related to the possible robot's movements.

A cell is considered explored once the robot has moved to it at least once. As the robot enters the cell for the first time, it is assumed that all of its walls, if any, are detected.

In the hypothetical maze depicted in Figure 3.8, both versions of the algorithm disagree about which is the shortest path once the maze has been fully explored. In this maze, the robot starts at the bottom-left corner cell `S`, facing up, and must reach one of the goal cells `G`. The robot has to choose between the upper and lower paths, which have different lengths in terms of straight-line and turns, as shown in Table 3.9.

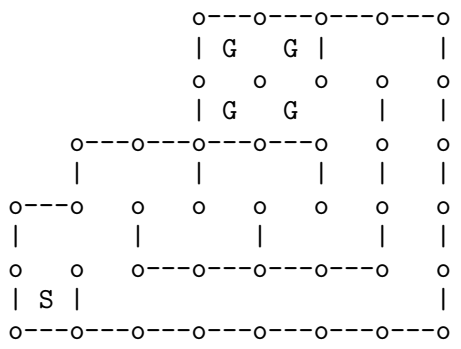


Figure 3.8: Hypothetical maze in which the modified flood fill algorithm results differ.

Table 3.9: Upper and bottom path analysis.

Path	Straights	Turns	Total movements
Upper	13	11	24
Lower	15	7	22

3.4.1 Flood fill algorithm

The set of possible robot movements in a 3×3 grid without any wall are illustrated in Figure 3.9.

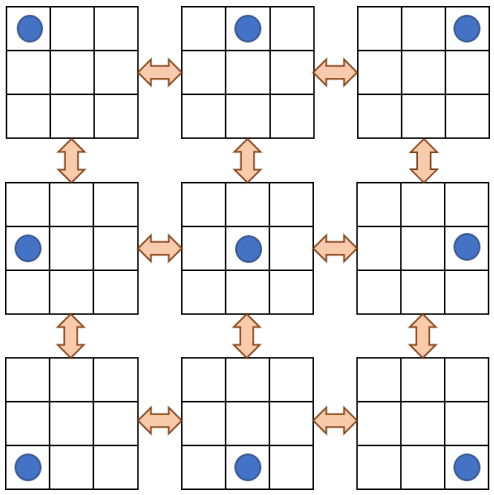


Figure 3.9: Robot movement considered in the traditional flood fill.

The blue colored circle represents the robot and the arrows represent the possible movements, which all have unitary costs. The allowed movements are left and right, up and down. Diagonal movements are not considered. Note that the unitary cost movements of this algorithm do not reflect the movements considered for the robot. For example, the real robot is not allowed to move one cell forward and then immediately returning to its starting position, as this algorithm implies; it would need to rotate 90° twice before moving back. The algorithm works by calculating how many movements are needed to reach a goal cell from each of the other cells.

In the hypothetical maze, the cumulative cost of the shortest path from the starting cell to the goal is 13, achieved through the upper path, as shown in Figure 3.10. The cost 13 is equivalent to the amount of straight-line movements present in the upper path,

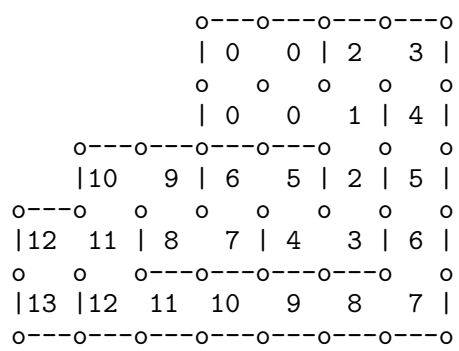


Figure 3.10: Costs assigned by the traditional flood fill algorithm

ignoring the amount of turns.

By following this algorithm, the robot would choose the upper path, despite the cumulative cost of the lower path being lower, which is undesired.

3.4.2 Proposed modified flood fill algorithm

The proposed modification to the flood fill algorithm takes into account both straight-line movements and turns, as desired for the robot movement scheme considered. In this algorithm, the state of the robot is defined by its orientation (up, down, left or right) in addition to the (x,y) location of the cell it is in.

Some of the allowed movements for the robot are illustrated in Figure 3.11. The robot is represented by a blue colored shape that points in the direction it is facing. The robot is allowed to rotate by 90°, to the left or right, but may only move to the cell it is currently facing.

If the position of the robot is represented by a tuple $p = (x, y, t)$, where x and y represent the cell position and t the robot orientation, which can only hold values of "up", "down", "left" or "right", consider two sets $S_1(p)$ and $S_2(p)$. Set $S_1(p)$ contains the set of positions reachable from position p. Set $S_2(p)$ contains the set of positions that may reach position p.

For example, assuming that the robot is in position $p = (1, 1, \text{right})$ and there isn't a wall in front of it. In this case, set $S_1(p)$ contains three other positions: two of them are

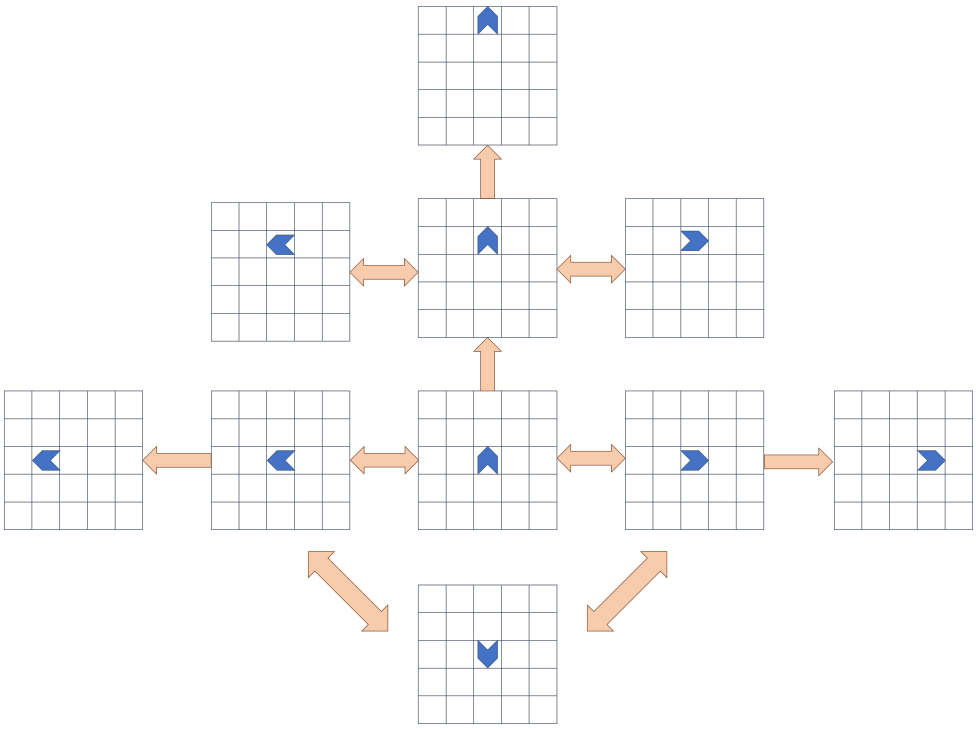


Figure 3.11: Robot movement considered in the modified flood fill.

reached by rotating 90° clockwise or counterclockwise, namely $(1, 1, \text{up})$ and $(1, 1, \text{down})$. The other one is $(2, 1, \text{right})$, reached by moving forward. Meanwhile, set $S_2(p)$ would also contain $(1, 1, \text{up})$ and $(1, 1, \text{down})$, that reach p by rotating, and $(0, 1, \text{right})$ that reaches p by moving forward.

Algorithm 2 describes how the modified flood fill works. Consider the set of all goal positions G. This will be the starting point of the algorithm. Positions in G will be assigned cost 0 and will be used as a baseline for the cost assigned to all other positions.

There is a First-in First-out queue Q, which stores positions that haven't been processed. Every position added to this queue will be processed in the same order they were added. The set P of position that have already been added to Q. This exists in order to assure that each position is added to the queue Q only once, and thus is processed only once.

At each iteration, the algorithm processes a position $\mathbf{p}$ from the queue Q. This means that $\mathbf{p}$ will have a cost assigned, which is equal to the lowest cost of the positions in $S_1(p)$

plus one. Also, every position in $S_2(p)$ that hasn't been added to P already will be added to Q, so it will be processed later on. This ensures that all accessible positions will be processed at some point during execution.

Algorithm 2: Modified Flood fill Algorithm

```
1  Initialize Initialize Cost(p) ← 0, for all p ∈ G;
2  Initialize Cost(p) ← ∞, for all p ∉ G;
3  foreach p ∈ G do
4      Add g to Q;
5      Add g to P;
6  while Q is not empty do
7      Dequeue p from Q;
8      foreach s_2 ∈ S_2(p) do
9          if s_2 ∉ P then
10             add s_2 to Q;
11             add s_2 to P;
12         if p ∉ G then
13             Cost(p) ← min Cost(S_1(p)) + 1
```

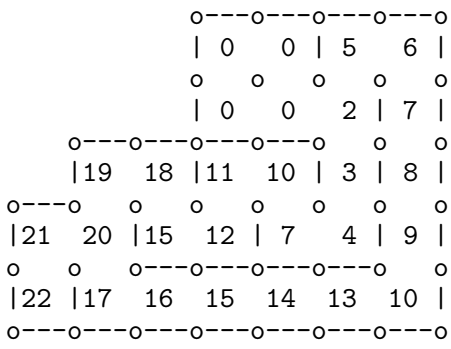
The costs assigned to the hypothetical maze are presented in Figure 3.12. Each of the sub figures indicate that the robot is facing a different direction. The cost assigned to the starting cell when the robot is facing up is 22, which correspond to the cumulative cost of the lower path.

As expected, the path with the lowest cost corresponds to the lower path, that has the least amount of necessary movements.

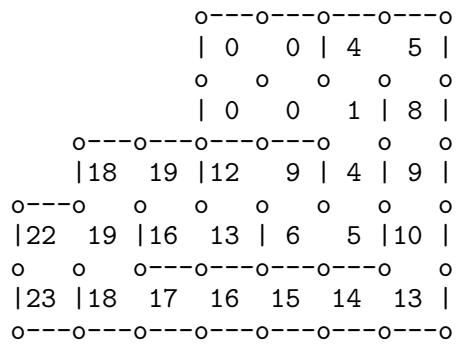
3.4.3 Finding the shortest path

The flood fill algorithm is used to assign costs that represent how far the robot is from the maze center. Once the costs of the positions in a maze have been calculated, two paths are defined: the open path and the closed path.

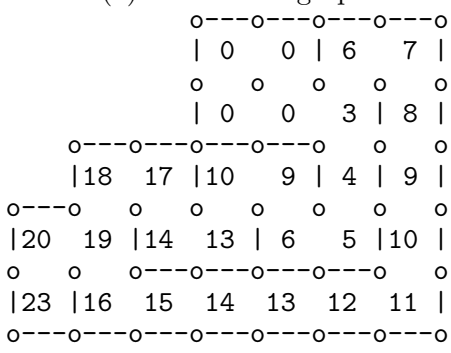
The open path is the most optimistic guess that makes the robot move from the starting cell to one of the goal cells. It considers that there are no walls in the cells that



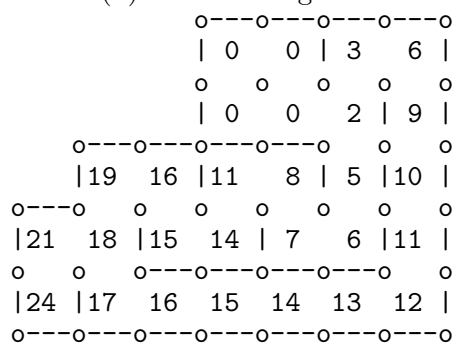
(a) Robot facing up



(b) Robot facing left



(c) Robot facing right



(d) Robot facing down

Figure 3.12: Costs assigned by the modified flood fill algorithm to the robot configurations.

haven't been explored yet. The closed path is the shortest path from the starting cell to one of the goal cells that makes the robot move through only cells that have already been explored, and thus is known to exist.

By using an approach called Adachi Method 0, described by Peter Harrison in [53], the robot moves back and forth between the center of the maze and the starting position, guided by the flood fill algorithm which is executed at the start of the competition and every time a new wall is detected, exemplified by Figure 3.13. Once the open and closed paths have the same cost, the algorithm halts, because there isn't a path shorter than a currently known closed path, even if it goes through cells that haven't been explored yet.

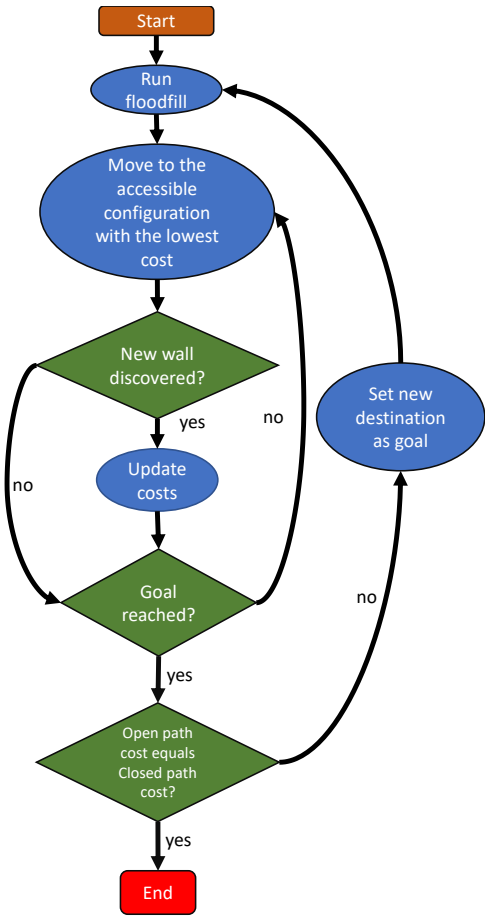


Figure 3.13: Flowchart to micromouse Flood fill strategy adopted. Extracted from [52]
.

The shortest path is found once the cost of the closed path is equal to the cost of the

open path. This situation is assured to happen once all cells have been explored, but may also happen earlier during execution.

At the start of the competition, all cells are unknown. When the robot tries to follow the open path towards the goal. If it succeeds, the open path will be the same as the closed path, and that is the best path. If a wall is blocking the path, new cells have been explored and the robot is closer to exploring the maze fully. Eventually, the entire maze will be explored and the shortest path calculated.

3.5 Arduino library

Two libraries were implemented for the purpose of this work. One of them is essential for utilizing the software, and addresses the communication between the simulator and the HIL. The other implements the proposed flood fill algorithm modification detailed in Section 3.4.2.

3.5.1 Simulator integration

A library compatible with the Arduino environment is provided with the simulator. The library is intended to be easy to use, by providing an interface for addressing each component placed in the robot. The whole library architecture is illustrated by the Unified modeling language (UML) diagram in Figure 3.14. The diagram covers entities that should be instantiated directly, and others that are meant for the internal workings of the library. The role of each entity is described in Tables 3.10 and 3.11.

Additionally, a Facade [54] class `SimulationRobot`, illustrated by Figure 3.15, has been implemented in order to facilitate the interaction with the robot model provided with the simulator by simplifying the interaction with the underlying component objects. It saves the user from having to declare each component individually and contains a `RobotConstructionSpecification` object, which contains physical characteristics of the robot, such as wheel radius, wheel-to-wheel distance, encoder pulses and motor gearbox

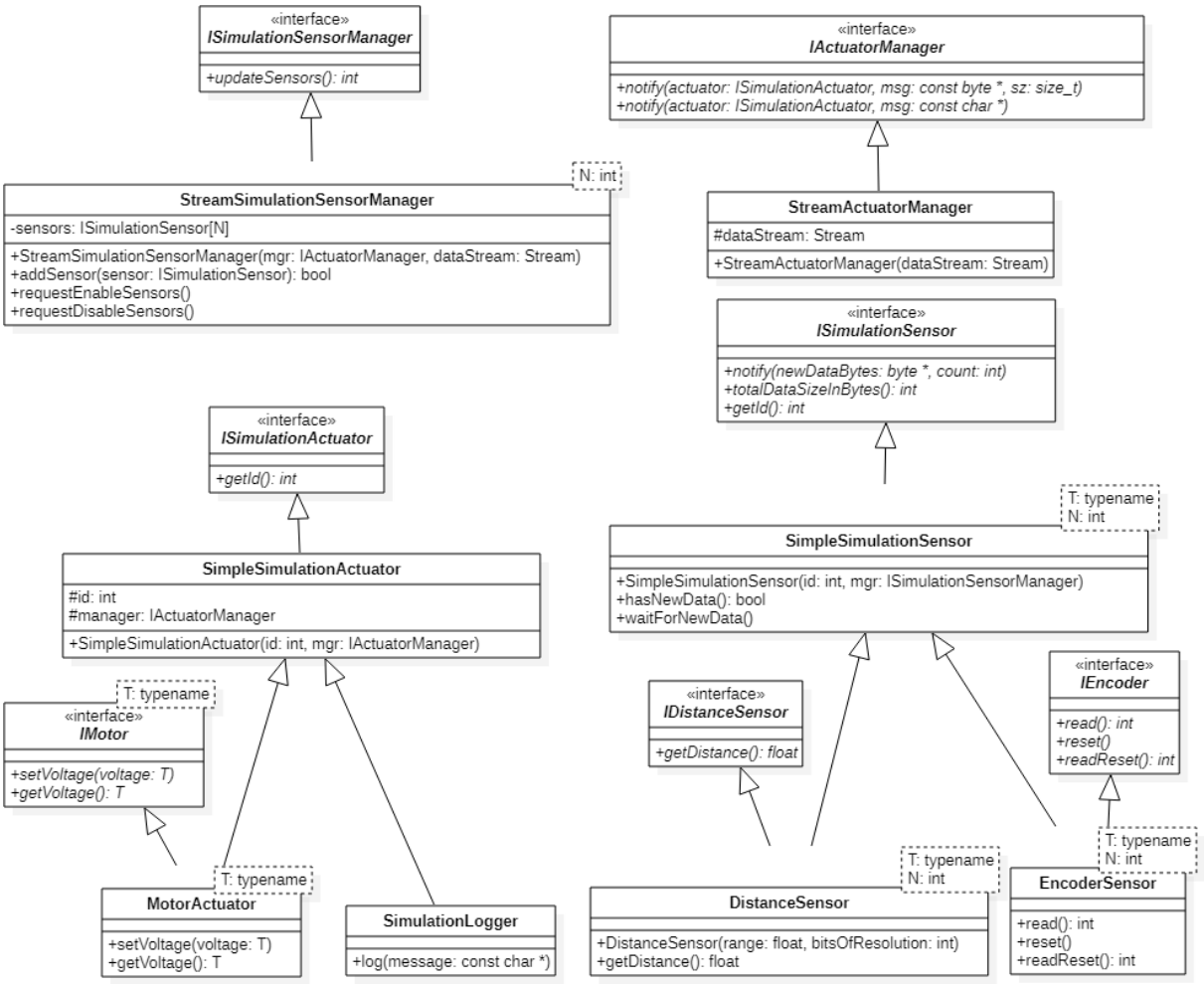


Figure 3.14: Class diagram for Arduino library for interacting with the simulator.

Table 3.10: Description of the internal library elements.

Entity name	Type	Description
ISimulationSensorManager	Interface	Interface for interpreting the data representing the sensor readings and notifying the corresponding sensor objects
IActuatorManager	Interface	Interface for sending data to the simulator in the correct format according to the defined protocol
StreamSimulationSensorManager	Class	Implementation for `ISimulationSensorManager` based off in the `Stream` class defined by the Arduino environment
StreamActuatorManager	Class	Implementation for `IActuatorManager` based off in the `Stream` class defined by the Arduino environment
ISimulationSensor	Interface	Interface for common sensor functionality, such as waiting for a new reading, and the possibility of being notified.
ISimulationActuator	Interface	Interface for sending actuator commands to the simulator
SimpleSimulationSensor	Class	Implementation of `ISimulationSensor`, that stores data sent by the simulator when notified.
SimpleSimulationActuator	Class	Implementation responsible for sending actuator commands to the simulator, by means of the `IActuatorManager`.

Table 3.11: Description of the user-facing Arduino library elements.

Entity name	Type	Description
DistanceSensor	Class	Specialization of `SimpleSimulationSensor` that exposes a method that calculates the measured distance based off the sensor specification.
EncoderSensor	Class	Specialization of `SimpleSimulationSensor` that accumulates the pulse count sent by the simulator and exposes the current count and may be reset.
MotorActuator	Class	Specialization of `SimpleSimulationActuator` that controls the voltage applied to a motor.
Simulation Logger	Class	Specializationn of `SimpleSimulationActuator` that logs information to the application console.

ratio. Its corresponding UML diagram is shown in Figure 3.16. Usage of the Facade class is exemplified in Listing 3.1.

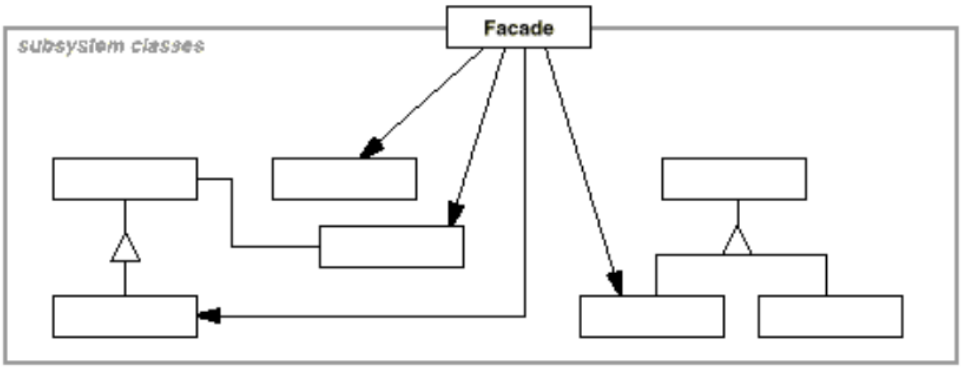


Figure 3.15: The Facade design pattern.

Listing 3.1: Interacting with the simulator through the facade object.

```
1  void loop() {
2    robot.update(); // Processes input from the simulator
3
4    /* Range sensor readings */
5    float leftDist = robot.leftDistance();
6    float rightDist = robot.rightDistance();
7    float frontDist = robot.frontDistance();
8
9    /* Encoder count readings */
10   int leftCounts = robot.leftEncoder();
11   int rightCounts = robot.rightEncoder();
12
13   /* Setting motor speed (10-bit magnitude value) */
14   robot.motorLeft(-1023); // 100% backward speed
15   robot.motorRight(1023); // 100% forward speed
16 }
```

If the user intends to port the code that interacts with the simulator to a real hardware project, one of the possibilities is to implement a class that follows the Adapter [54] pattern, presented in Figure 3.17. For example, if there is a class `VL53L0XHardwareLibrary` that controls a real-world ToF sensor defining a `doMeasurement` method, the user should implement a `VL53L0XAdapter` class, which instantiates an object of type `VL53L0XHardwareLibrary` and implements the `IDistanceSensor` interface. The `getDistance` method definition should adapt the output of the `doMeasurement` method to the appropriate format. Then,

the `VL53L0XAdapter` may act as a direct replacement for the `DistanceSensor` class, as they share a common interface.

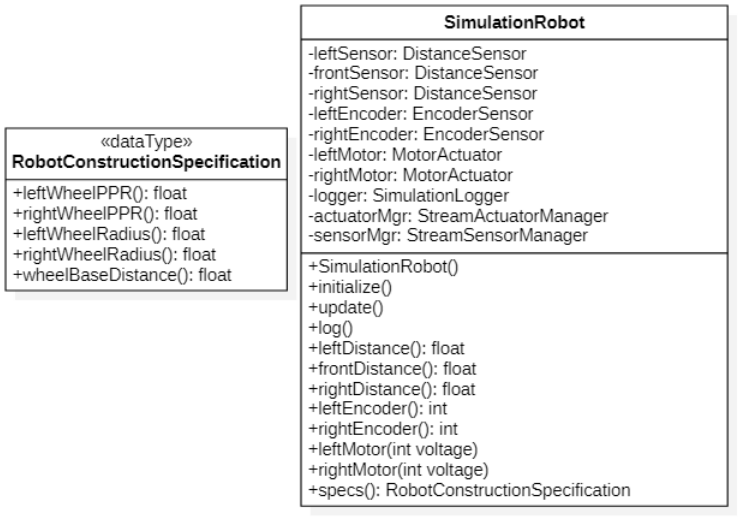


Figure 3.16: Robot facade class for simplified usage.

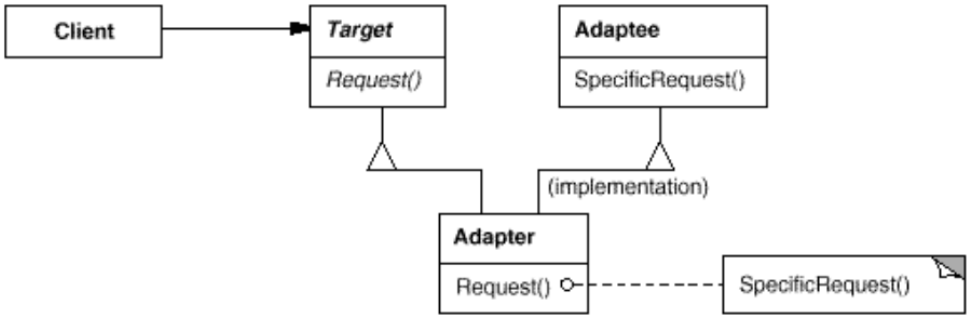


Figure 3.17: The Adapter design pattern.

3.5.2 Pose estimation and modified flood fill

The `PoseEstimator` is used to estimate the (x, y, θ) pose of the robot, which can be instantiated by passing in the `RobotConstructionSpecification` object. It implements the `updatePose` method, which takes as parameters the pulse counts for the left and right encoders, the `getPose` method, which returns the current estimated pose, and the `setPose` method, which can be used to set an initial value for the pose or to correct the estimate based on some absolute measurement.

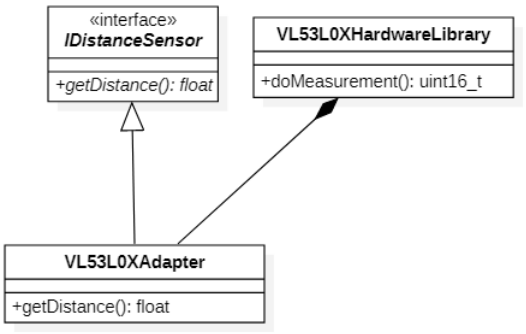


Figure 3.18: Integrating an existing library with the adapter design pattern

Table 3.12: Robot properties available as variables.

Wheel radius	0.032 m
Wheel base distance	0.096 m
Effective encoder CPR	358.32

The `Maze` class defines a representation of the micromouse maze. The size of the maze is passed as template parameters and are used to allocate an array of `Cells` of sufficient size. Each `Cell` stores an 8-bit variable. Bits one through four are used to indicate whether the cell contains a wall in the up, right, left or down directions, respectively. The high-nibble is unused. The `Maze` class implements methods `placeWall`, `removeWall` or `hasWall` for setting, clearing or checking the existence of a wall at some cell position.

The `FFSolver` takes in a `Maze` object as a constructor parameter and implements the modified flood fill algorithm. The `calculateCosts` executes the algorithm.

Finally, the `PathManager` class implements the Adachi Method 0, which makes the robot move back and forth between the maze goal and the starting cell by means of the modified flood fill algorithm, exploring unknown cells along its path.

Chapter 4

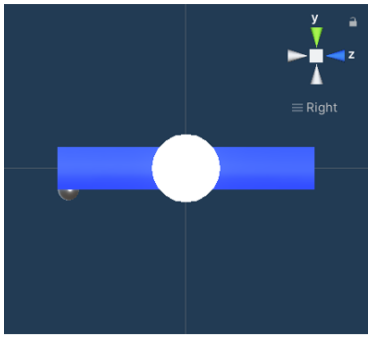
System Architecture

This chapter deals with the definition of the robot model used in the simulator and the serial communication protocol used to provide HIL capabilities to the simulator. It also presents the maze file format expected for importing the maze into the 3D environment.

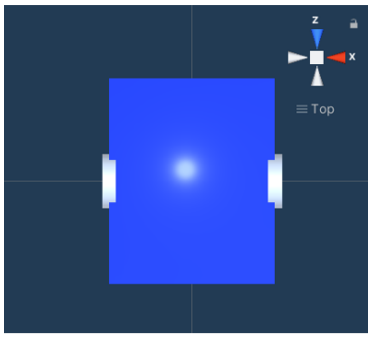
4.1 Robot simulation model

The robot model built for the simulation consists of four interconnected rigid bodies, and is based on the differential drive robot built for the micromouse competition by Eckert [55]. It consists of a base plate, two side wheels actuated by DC motors and a caster wheel for balancing. The physical properties and positioning of each component is described in Table 4.1. The static and dynamic friction values for the caster wheel were set to 0.1, and, for the side wheels, the coefficients were set to 0.25 and 0.35, respectively. These coefficient values are used by the physics engine but aren't directly related to real world friction coefficients.

The caster wheel and the side wheels are connected to the main plate by joints, as shown in Table 4.2. The caster wheel linear motion is locked to the base plate, but it may rotate freely in all rotational axis. There isn't a standard joint type for such behaviour, so the configurable joint was used. The side wheels are connected by hinge joints, which restrict all movement except for rotation about one axis.



(a) Side-view of the robot model.



(b) Top-view of the robot model.

Figure 4.1: The simulation robot model.

Table 4.1: Physical properties of the robot model. Mass is given in kilograms, position in meters and rotation in degrees.

Name	**Mass**	**Shape**	**Position**			**Scale**			**Rotation**		
			X	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**
Plate	0.127	Cube	0	0	0	0.096	0.02	0.12	0	0	0
Left wheel	0.01	Cylinder	-0.048	0	0	0.032	0.004	0.032	0	0	90
Right wheel	0.01	Cylinder	0.048	0	0	0.032	0.004	0.032	0	0	-90
Caster wheel	0.01	Sphere	0	-0.01	-0.055	0.01	0.01	0.01	0	0	0

Table 4.2: Description of the joint connections.

Body	**Connected to**	**Joint type**	**Linear Motion**			**Angular Motion**		
			X	**Y**	**Z**	**X**	**Y**	**Z**
Left wheel	Plate	Hinge	Lock	Lock	Lock	Lock	Free	Lock
Right wheel	Plate	Hinge	Lock	Lock	Lock	Lock	Free	Lock
Caster wheel	Plate	Configurable	Lock	Lock	Lock	Free	Free	Free

Three ToF sensors are placed along the the frontal part of the robot. The central sensor points forward and detects obstacles directly in front of the robot, while the lateral sensors point diagonally. Their positioning is described in Table 4.3. Their position and rotation is relative to the robot base plate, as it is defined as their parent. The colors chosen for drawing the ray on the screen were green, for when the an object is detected, and red otherwise. The serial interface is configured as a 16-bit output, corresponding to the VL53L0X component.

Table 4.3: Description of the Time-of-flight sensors. Position is relative to its parent, rotation is given in degrees.

Component	**Parent**	**Position**			**Rotation**			**Ray color**		**Serial Interface**	
		X	**Y**	**Z**	**X**	**Y**	**Z**	**Hit**	**Miss**	**ID**	**Resolution**
Left ToF	Plate	-0.5	0	0.5	0	0	0	Green	Red	0	16 bits
Right ToF	Plate	0.5	0	0.5	0	0	-90	Green	Red	1	16 bits
Front ToF	Plate	0	0	0.5	0	0	90	Green	Red	2	16 bits

Table 4.4: Models used for the robot components.

Component	**Model**
Battery	Ideal
Left ToF	VL53L0X
Right ToF	
Front ToF	
Left h-bridge	Ideal
Right h-bridge	
Left motor	Pololu LP 30 : 1
Right motor	
Left encoder	12 CPR
Right encoder	

The counts per revolution parameter for the encoders were chosen according to [45]. Their resolution was arbitrarily chosen to be 2 bytes, meaning that a 16-bit integer, signed integer value is sent to the microcontroller at every simulation time step. The encoders are connected to the extended motor shaft, according to Table 4.5.

Table 4.5: Description of the encoders.

Component	Connected to	Encoder CPR	Serial Interface	
			ID	Resolution
Left encoder	Left motor shaft	12	3	2 bytes
Right encoder	Right motor shaft	12	4	2 bytes

Both h-bridges are powered by a 6 volts regulated power supply and have 10-bit resolution for their duty cycle similar to Wemos D1 motor shield [56][57]. Their commands are transmitted across two bytes. These parameters are summarized in Table 4.6.

Table 4.6: Description of the h-bridges.

Component	Voltage Source	Connected to	Serial Interface	
			ID	Resolution
Left h-bridge	6 V	Left motor	5	10 bits
Right h-bridge	6 V	Right motor	6	10 bits

4.2 Serial Communication protocol

The microcontroller communicates with the simulator through Universal serial bus (USB). Whilst microcontrollers usually don't support USB natively, they support the Universal asynchronous receiver-transmitter (UART) protocol, and development boards such as the Arduino UNO include USB adapter chips.

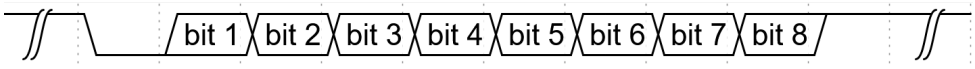


Figure 4.2: Timing diagram for one packet transmitted using the UART protocol, configured as 8N1.

The serial port connection is configured to the 8N1 standard, meaning that besides the mandatory start bit, each packet of data contains 8 data bits, no parity bit, and one stop bit. Therefore, each byte of data requires 10 bits to be transmitted. This configuration for the UART protocol is illustrated in Figure 4.2.

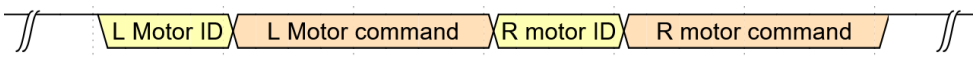


Figure 4.3: Timing diagram for transmitting the voltage commands for the left and right motors sequentially.

Every component object that interacts with the serial port has a unique ID, configured at both the simulator and the microcontroller program. The ID of a component pair is a matching, uniquely-identifiable 1-byte value. Every exchange of data between the microcontroller and the simulator begins with the component ID, followed by the bytes the component expects to receive. This situation is shown in Figure 4.3, where the ID byte for the left motor is transmitted, followed by two bytes representing the speed command. Then, the same happens for the right motor.

At the start of every physics update phase, the simulator checks for serial input. After reading all incoming bytes, if any, all sensors are updated and may notify the program if they have performed new readings. If there are any new readings, they are transmitted at the end of the physics update loop.

Equation 4.1 is used to calculate how much time t it takes to transmit N bytes of sensor readings or actuator commands at baud rate B. The results are presented in Table 4.7, for up to four bytes.

$$t = \frac{10N}{B} \tag{4.1}$$

The first entry represents the ID packet overhead, which is a single byte. Two bytes would represent a sensor with digital output, or with an analog sensor with maximum resolution of 8 bits. The third entry is related to all sensors and actuators described in this work; the first byte is the overhead, and the remaining two bytes are either the sensor readings or motor voltage commands. The last entry, for five bytes, could be used to transmit single-precision floating point values.

The values presented only account for transmitting via the Universal asynchronous receiver-transmitter (UART), and doesn't consider the overhead of the communication protocol used in the Universal serial bus (USB).

Table 4.7: UART transmission times.

Data bytes	Total bits	Baud rate	Time
1	10	9600	1.042 ms
		38400	260.4 us
		115200	86.80 us
2	20	9600	2.083 ms
		38400	502.8 us
		115200	173.6 us
3	30	9600	3.125 ms
		38400	781.3 us
		115200	260.4 us
4	40	9600	4.167 ms
		38400	1.042 ms
		115200	347.2 us
5	50	9600	5.208 ms
		38400	1.302 ms
		115200	434.0 us

4.3 Maze Generator

The maze generator module textual input standard is defined by [58] and represent the mazes into a 3D environment in accordance to micromouse competition standards given in Table 4.8. There are poles in the corner of every cell, and there may be a wall at each edge of every cell.

The “o” characters represent the poles, three hyphens “---” represent an horizontal wall and a pipe “|” represents a vertical wall. The center of each cell may contain the character “G”, indicating one of the set goal cells, and an “S” indicates the starting point for the robot. There may be one or more goal cells, but only one starting point.

A collection of such files is available in micromouseonline’s Github repository [58], and includes mazes used in a variety of competitions around the world. Figure 4.4 shows the maze used in All Japan’s 2018 contest, in the appropriate text format.

Table 4.8: Obstacle physical dimensions.

Obstacle	**Size (cm)**		
	X	**Y**	**Z**
Wall	16.8	1.2	5
Pole	1.2	1.2	5

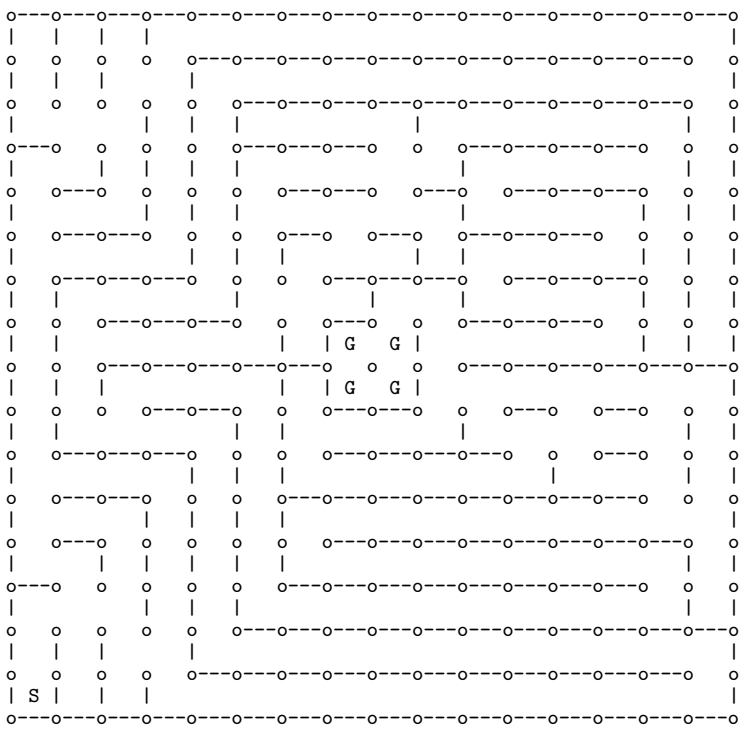


Figure 4.4: Textual representation for the maze used in All Japan's 2018 edition.

Chapter 5

Results

This chapters presents the generated environment for the micromouse contest and a case-study analysis for the proposed modified flood fill algorithm.

Besides the micromouse contest, three other scenarios are presented. Those were developed during Verão com Ciência and have been used during the RoboSTEAM hackathon.

5.1 Simulated environments

During the software execution, the user may open the main menu by pressing the Escape (ESC) key or clicking the button in the bottom-left corner of the screen.

The menu appear on the screen, as Figure 5.1, allowing the user to resume the program execution, quit the application, or choose one of the simulated environments: Wall distance control, speed control, time trial or micromouse.

In all of the environments, the graphical interface at top-right corner of the screen, as illustrated by Figure 5.2, allows the user to establish a connection to the microcontroller via the serial port. The connect button should be pressed after specifying the baud rate and port name fields.

Implementations for the algorithms presented in the following subsections are available in the simulator repository [2].

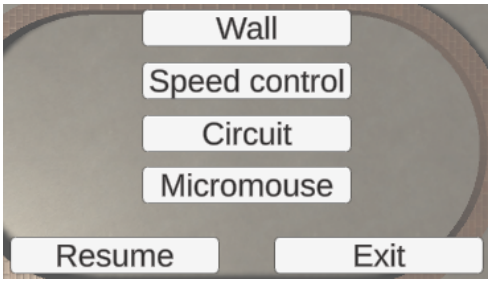


Figure 5.1: Screen capture of the main menu, showing the buttons for resuming execution, quitting the program, and switching between the simulated scenarios.

Figure 5.2: Graphical interface for establishing the serial port connection.

5.1.1 Speed control challenge

In this environment, the goal is to make the robot maintain its linear and angular velocities for 5 seconds at the predefined values of 1 m/s and 2 rad/s, respectively, with a tolerance of ±10 %. The current robot velocity can be estimated with the robot's kinematic equations and reading the encoder's output.

When this environment is selected, the user is presented with the screen shown in Figure 5.3. The current velocity of the robot is displayed at the top-left corner of the screen, in red or green, depending on whether it is within the challenge goal.

One way of completing this challenge is to follow Algorithm 3. If the estimated linear velocity is different from the goal, the speed of both wheels should either increase or decrease. If the estimated angular velocity is different from the goal, the speed of one wheel should increase while the other decreases.

5.1.2 Distance control challenge

The user is presented with the robot's central point positioned 10 cm away from a wall, as Figure 5.4, and the goal is to move alongside the wall at a distance of 50 cm, with a tolerance of ±10 %, for 5 seconds

The values for the goal and current distance are shown at the top-left of the screen.

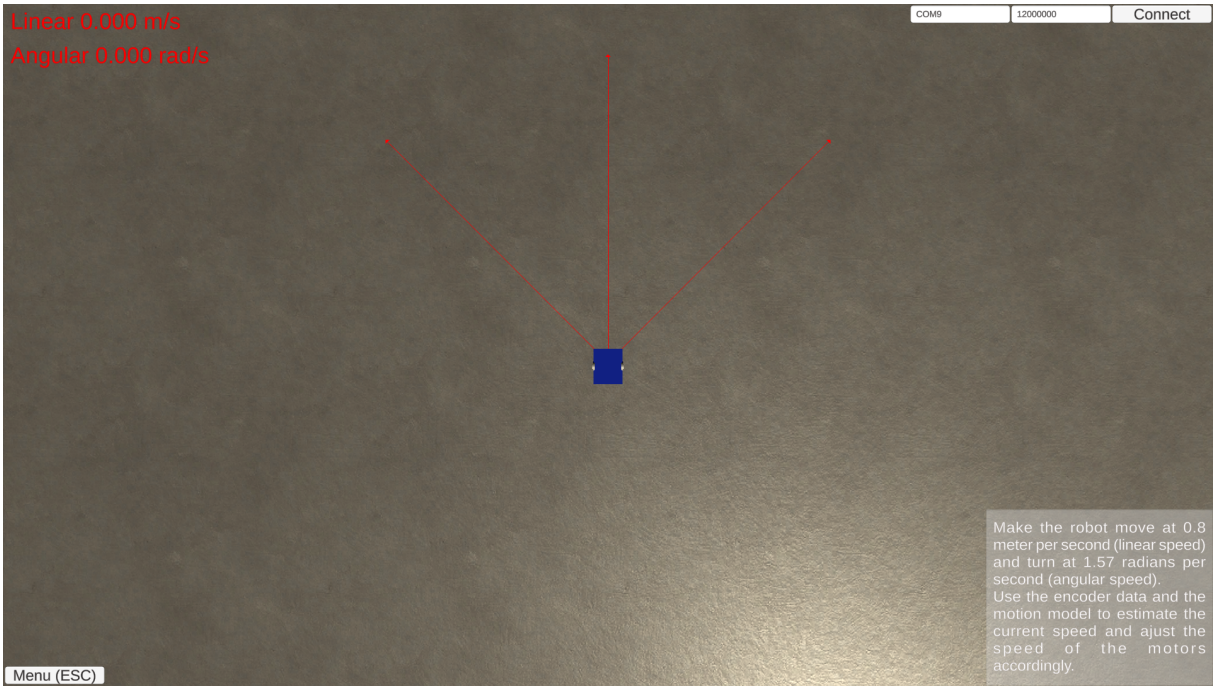


Figure 5.3: Screen capture during the speed control challenge.

Algorithm 3: Speed control.

Input: Encoder sampling time t, the target angular speed ω, the target linear speed V

Result: The robot moves at angular speed ω and linear speed V

$BaseSpeed \leftarrow 0$;
$OffsetSpeed \leftarrow 0$;
for *every* t *elapsed* **do**
    $\omega_{calc} \leftarrow AngularSpeed$;
    $V_{calc} \leftarrow LinearSpeed$;
    $\omega_{error} \leftarrow \omega_{calc} - \omega$;
    $V_{error} \leftarrow V_{calc} - V$;
    $BaseSpeed \leftarrow BaseSpeed + V_{error} \cdot k_{\omega}$;
    $OffsetSpeed \leftarrow OffsetSpeed + \omega_{error} \cdot k_{V}$;
    $LeftMotorSpeed \leftarrow BaseSpeed + OffsetSpeed$;
    $RightMotorSpeed \leftarrow BaseSpeed - OffsetSpeed$;

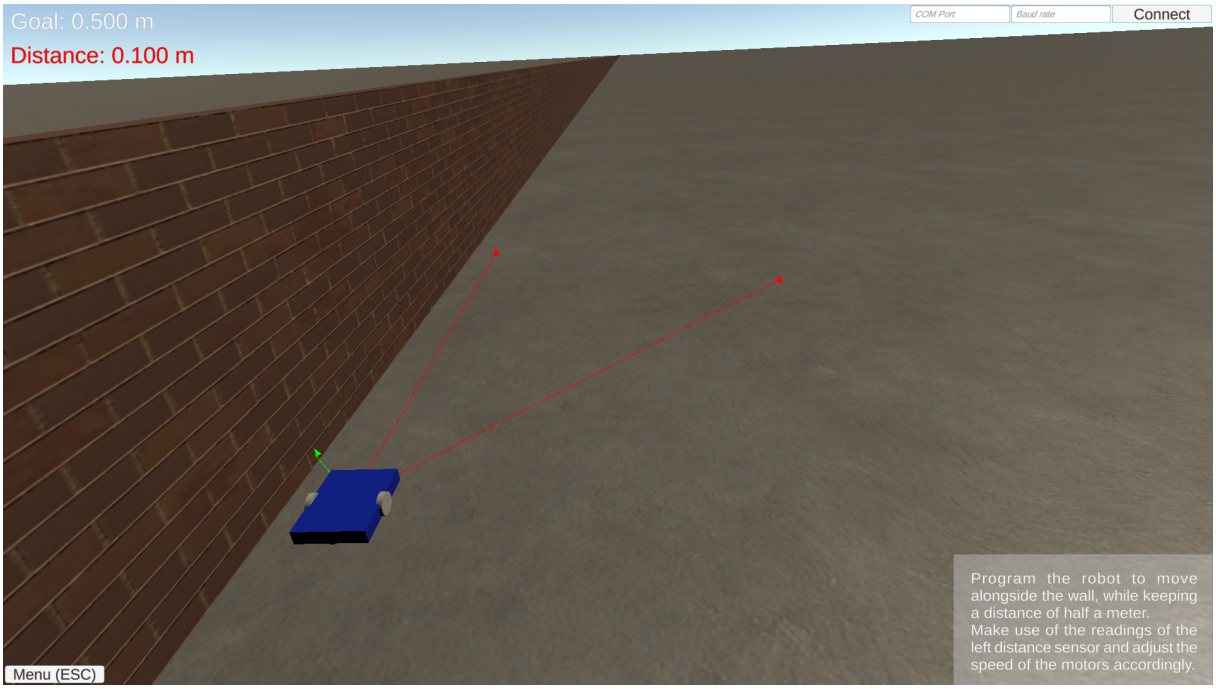


Figure 5.4: Screen capture during the distance control challenge.

Once the distance is within the specified goal, the text color turns green.

One way of solving this challenge is to implement a proportional controller, based on the readings of the left ToF sensor. Algorithm 4 describes this approach. Firstly, the desired sensor reading z_{goal} is calculated, based on the robot's geometry and sensor positioning. Then, an *error* value is calculated by subtracting the desired reading z_{goal} from the current reading z_{left}. If *error* has positive value, the speed of the left motor should increase while the speed of the right motor increases. Else, the opposite should happen.

5.1.3 Time trial

The object is placed in a circular track and should complete laps in the shortest amount of time possible. A screen capture is shown in Figure 5.5. In the top-left corner, the current lap number is displayed alongside the current elapsed time. Once the first lap is completed, the best lap's time is also displayed.

Besides trying out different programs and observing their results, there is no concrete

Algorithm 4: Following the wall at distance d.

Input: The robot width L, the target distance d, base speed s, proportional gain k

Result: The robot follows the wall at d distance

$z_{goal} \leftarrow \cos(45°) \cdot d + L/2$;
for *every new reading* z_{left} **do**
    $e \leftarrow z_{left} - z_{goal}$;
    $LeftMotorSpeed \leftarrow s - e \cdot k$;
    $RightMotorSpeed \leftarrow s + e \cdot k$;

goal in this challenge.

One possible way of completing the laps is to implement a wall-follower algorithm similar to Algorithm 4, except that it would use the right wall as a reference.

5.1.4 Micromouse contest

Different scenarios for the micromouse contest are generated with the maze generator tool described in Section 4.3. Figure 5.6 is a screen capture of the scenario generated for All Japan's 2018 maze, alongside the robot model described in Section 4.1, placed at the bottom-left corner of the maze. At the top-left corner, the 10-minute competition countdown is displayed. It starts as the robot leaves the starting cell.

When the robot is performing a movement, either rotating or moving forward, its pose is estimated by sampling the encoder readings. Algorithm 5 explains how the 90° rotation movement is executed. Firstly, the motors speed commands are set to opposite values. Secondly, the change in angle is accumulated in the θ_{acc} variable after every sampling time t, and θ_{acc} is updated based on the estimated angular velocity ω. Whenever θ_{acc} is greater than 90°, the movement action has ended.

Algorithm 6 describes how the robot moves one cell forward and stays aligned in the center of the cell: the speed of both motors is set to a reference value M, and its pose estimate is updated by following Equations 3.29, 3.30 and 3.31. The velocity values are obtained by sampling the encoder readings after the sampling time t, and then calculating linear and angular velocities.

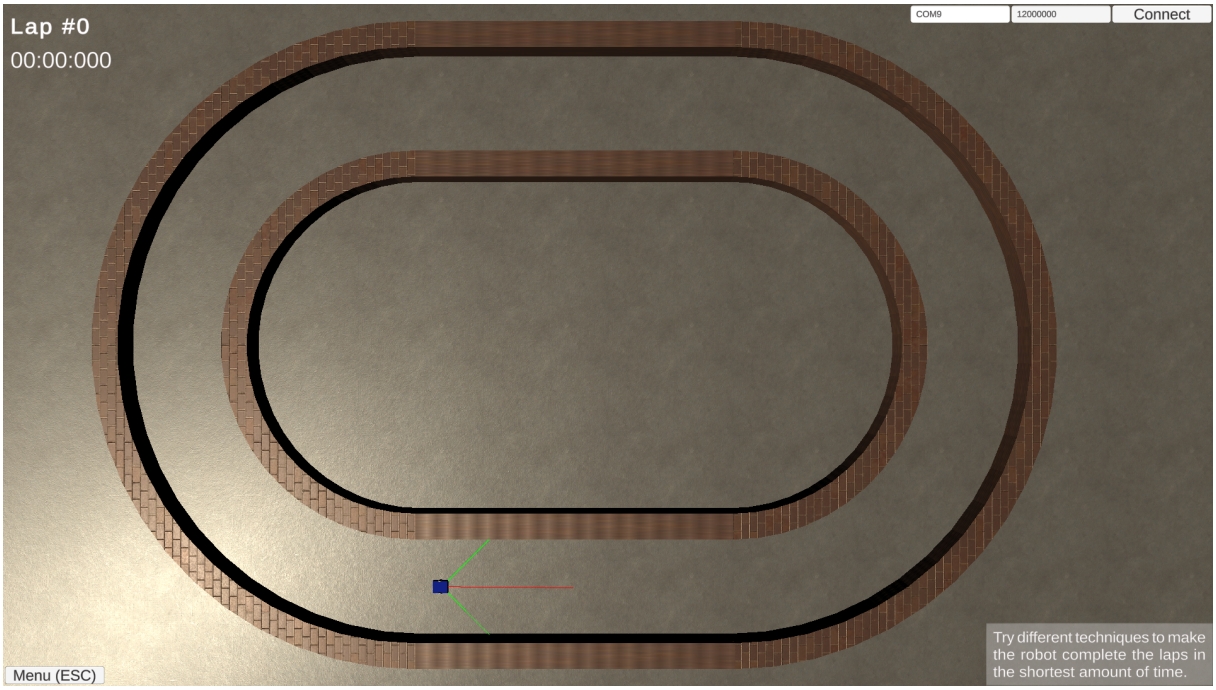


Figure 5.5: Screen capture during the time trial.

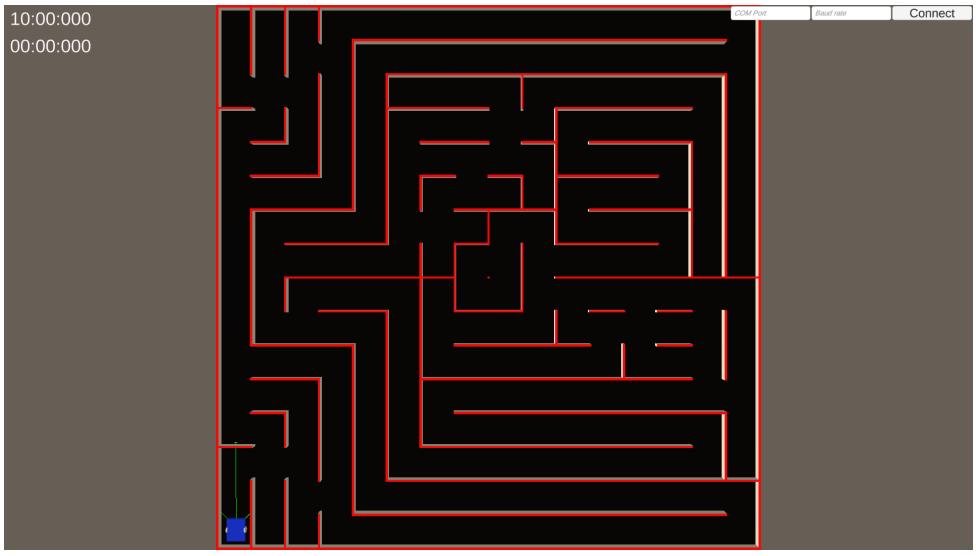


Figure 5.6: Screen capture of All Japan's 2018 maze generated environment.

Algorithm 5: Algorithm for rotating counter-clockwise.

Input: Motor speed command M
Result: The robot has rotated by 90° degrees about its central point

$\theta_{\text{acc}} \leftarrow 0$;
$RightMotorSpeed \leftarrow M$;
$LeftMotorSpeed \leftarrow -M$;
while $\theta_{acc} \leq 90°$ **do**
  for *every t elapsed* **do**
    $\omega \leftarrow AngularSpeed$;
    $\theta_{\text{acc}} \leftarrow \theta_{\text{acc}} + \omega t$;

The robot aligns itself in the center of the cell by using a proportional controller acting upon a calculated $offset$ value. Whenever both left and right walls are present, the offset is the difference between the left and right sensor ToF sensor readings z_{l} and z_{r}.

If only one of the walls is present, the sensor reading is compared to a reference value D_{l} or D_{r}, which is the expected sensor reading whenever the robot is located exactly at the center of a cell.

The robot has finished moving forward whenever the Euclidian distance, given by Equation 5.1, between the current pose estimate and the starting pose is greater than 18 cm, the distance between the center of two adjacent cells.

$$\text{dist}\left((x_0, y_0), (x_1, y_1)\right) = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \tag{5.1}$$

5.2 Modified flood fill case study

The modified flood fill algorithm has also shown different results when compared to the traditional flood fill algorithm for mazes that have been used in official micromouse competitions. This section presents a case study for the mazes used in All Japan's micromouse contests held in 2018 and 2019.

Results obtained by comparing the modified and traditional flood fill implementations will be presented. Firstly, the iterations of the Adachi method 0 that lead to the best

Algorithm 6: Algorithm for moving one cell forward.

Input: Motor speed command M, proportional gain k, reference distances D_l and D_r, the robot pose (x_0, y_0, θ_0)

Result: The robot has moved one cell forward

$x \leftarrow x_0$;
$y \leftarrow y_0$;
$\theta \leftarrow \theta_0$;
$RightMotorSpeed \leftarrow M$;
$LeftMotorSpeed \leftarrow M$;
while $dist((x, y), (x_0, y_0)) < 18\,\text{cm}$ **do**
    for *every t elapsed* **do**
        $V \leftarrow LinearSpeed$;
        $\omega \leftarrow AngularSpeed$;
        $x \leftarrow x + Vt\cos(\theta)$;
        $y \leftarrow y + Vt\sin(\theta)$;
        $\theta \leftarrow \theta + \omega t$;
    for *Every ToF reading* z_l, z_r **do**
        if *left and right walls detected* **then**
            $offset \leftarrow z_\text{l} - z_\text{r}$;
        else if *only left wall detected* **then**
            $offset \leftarrow z_\text{l} - D_\text{l}$;
        else if *only right wall detected* **then**
            $offset \leftarrow D_\text{r} - z_\text{r}$;
        $LeftMotorSpeed \leftarrow M - k \cdot offset$;
        $RightMotorSpeed \leftarrow M + k \cdot offset$;

path will be presented, and finally a numerical comparison of execution time.

5.2.1 Steps to find the best path

On All Japan 2018's maze (presented in Figures 5.7, 5.8, 5.9) the robot took three trips in order to find the shortest path: it found the center cells, returned to the starting cell, and then went back to the center. The visited cells are represented by light-blue color. The orange and blue lines represent the open path and the closed path, respectively, calculated as the robot reached it's destination.

After it's first trip, the algorithm lead robot to explore the upper part of the maze, up until the point where it found the center, goal cells. This is shown on Figure 5.7. A closed path through the route taken was calculated, which does not contain the dead-ends it has discovered. As most of the bottom part of the maze is unknown, there is a very short open path, which the robot will likely try to follow on it's way back to the starting cell.

The robot did explore the bottom part of the maze on it's second trip, depicted on Figure 5.8. Notice that many walls have been discovered and now the open path is not as short. A new closed path has also been traced: by having knowledge of the cells located on the bottom of the maze, it found out that there is a shorter path than the one known right after the first trip.

As the robot reaches the center cells for the third time, both closed and open paths have the same cost, and contain the same cells, meaning the robot knows it has found and optimal path according to the modified flood fill algorithm.

On All Japan 2019's maze (presented in Figures 5.10, 5.11, 5.12, 5.13), the robot took one extra trip: the shortest path was found after returning to the starting position for the second time. The visited cell are represented by light-blue color. The orange and blue lines represent the open path and the closed path, respectively, calculated as the robot reached it's destination

As can be seen on Figures 5.11, 5.12, 5.13 and Table 5.2, the closed path traced was

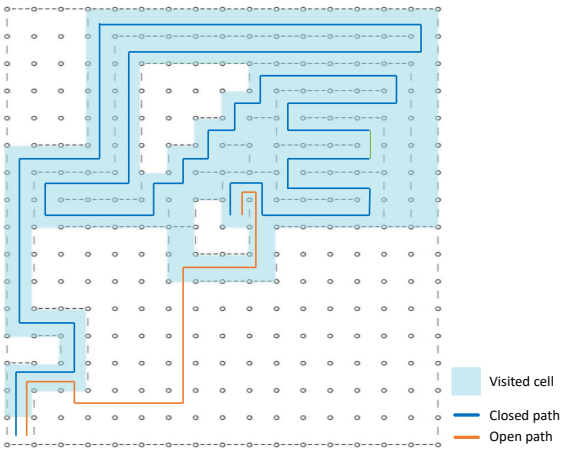


Figure 5.7: Known maze and paths after first trip (All Japan 2018).

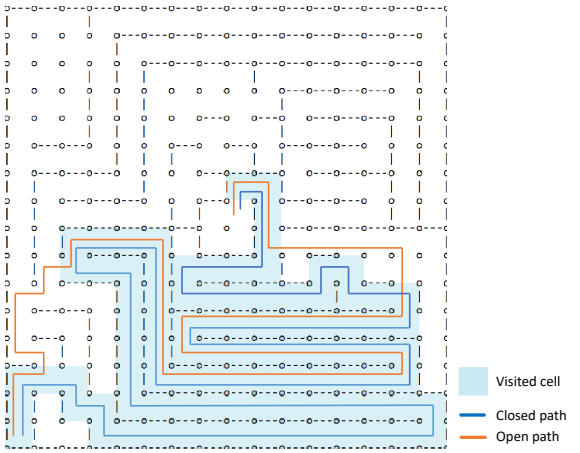


Figure 5.8: Known maze and paths after second trip (All Japan 2018).

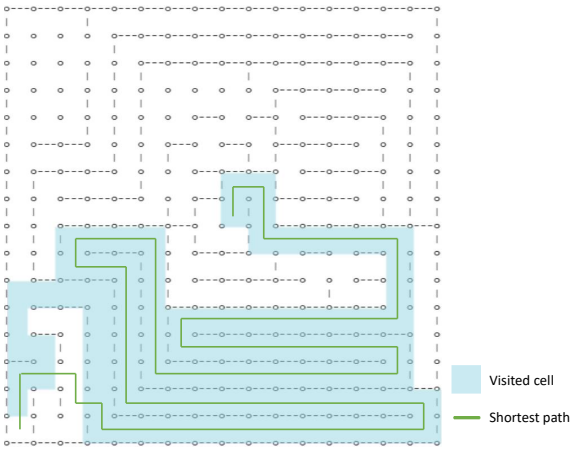


Figure 5.9: Known maze and paths after third trip (All Japan 2018).

already the one with the lowest possible cost. However, it wasn't known for sure that it was the case: trips 2, 3 and 4 lead the robot to explore different parts of the maze, so that there couldn't be possible to exist an open path with a lower cost.

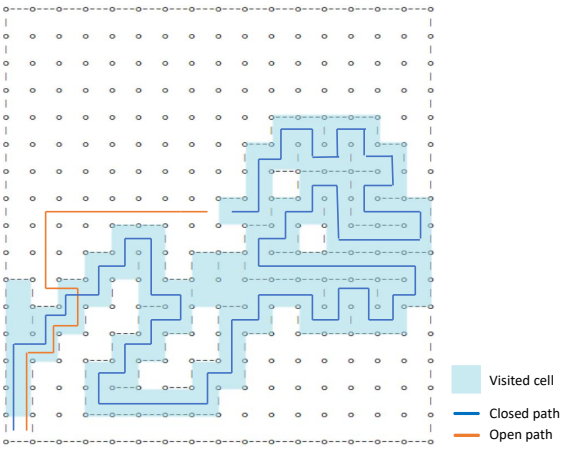


Figure 5.10: Known maze and paths after first trip (All Japan 2019).

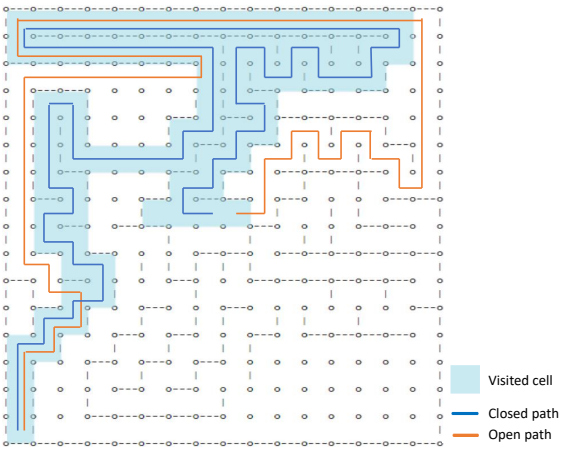


Figure 5.11: Known maze and paths after second trip (All Japan 2019).

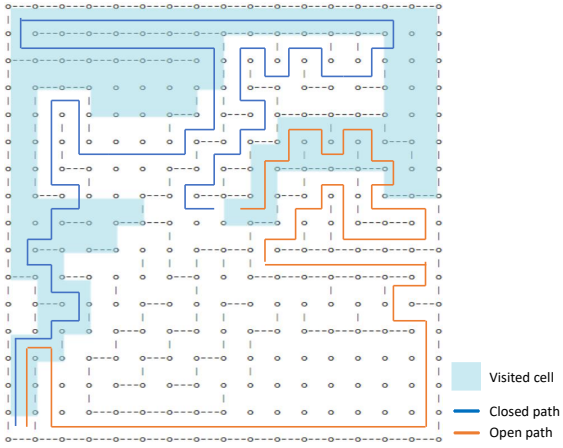


Figure 5.12: Known maze and paths after third trip (All Japan 2019).

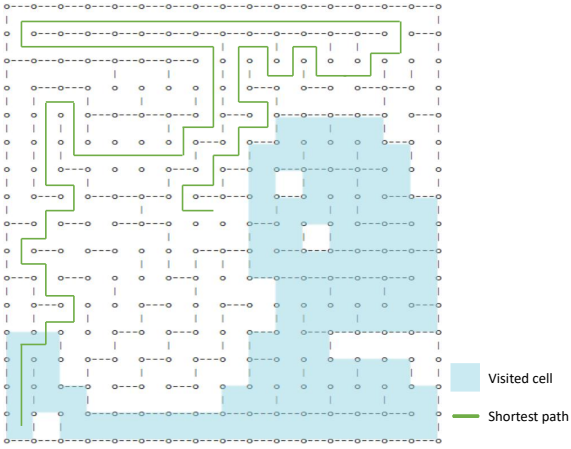


Figure 5.13: Known maze and paths after fourth trip (All Japan 2019).

Tables 5.1 and 5.2 show the cost of the open and closed paths after each trip, given the knowledge of the maze the robot had at the time. The completion time for each trip is also recorded. The time for the odd-numbered trips would be tracked at a real contest, as they are a speedrun from the starting point to the goal. When the least-cost path is

found, the completion time is the shortest.

Table 5.1: Trips to All Japan 2018 maze.

	Open path cost	Closed path cost	Completion time (mm:ss)
Trip 1	30	134	01:39
Trip 2	78	122	01:00
Trip 3	108	108	00:55
Trip 4	108	108	00:55

Table 5.2: Trips to All Japan 2019 maze.

	Open path cost	Closed path cost	Completion time (mm:ss)
Trip 1	24	137	01:15
Trip 2	95	118	01:06
Trip 3	97	118	00:56
Trip 4	118	118	00:56

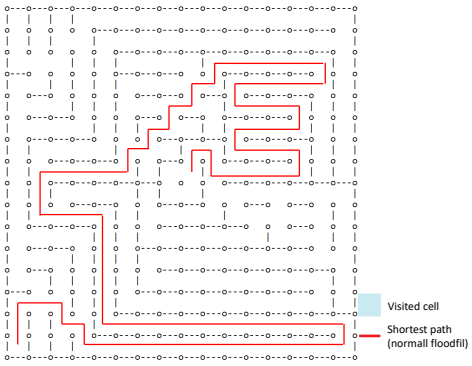


Figure 5.14: Shortest path according to the traditional Flood fill (All Japan 2018).

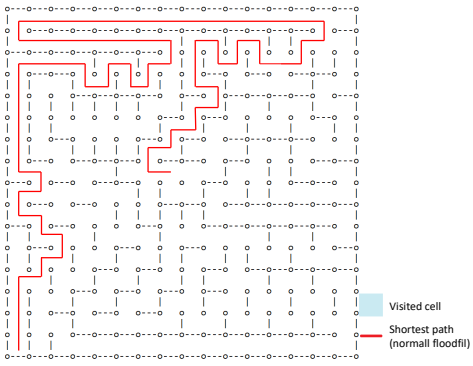


Figure 5.15: Shortest path according to the traditional Flood fill (All Japan 2019).

5.2.2 Comparison between traditional and modified algorithm

Tables 5.3 and 5.4 compare the path found by the proposed algorithm and the path that the traditional Flood fill algorithm would have found. The comparison contains the time which each algorithm took to compute a path and the cost. It was considered that moving one cell ahead has cost 1, and rotating 90° also has cost 1.

Table 5.3: Algorithm comparison for All Japan 2018 maze.

	Modified	Traditional
Time (ms)	26	9
Cost	108	118

Table 5.4: Algorithm comparison for the All Japan 2019 maze.

	Modified	Traditional
Time (ms)	26	9
Cost	118	120

As expected, the proposed algorithm requires more time to compute; approximately three times as much as the traditional one. The computation of the traditional Flood fill finishes after 9 ms, and the modified algorithm finishes after 26 ms (on ESP8266). This is due to the higher amount of robot positions it has to evaluate. Although it takes more time to compute, it finds what we consider to be a better path. On both analysed mazes, the paths found by both algorithms visit the same amount of cells of the maze. On the point of view of the traditional Flood fill algorithm, this means that both paths have the same cost. However, as the traditional Flood fill finds a path with more turns, the proposed approach found a path with lower cost.

As the proposed algorithm takes more time to be processed, the exploration phase will take more time. Even then, the extra amount of time taken is minimal. If the algorithm runs once for each of the 256 cells of the maze, the extra amount of time equals $(26-9)\text{ms} \times 256 = 4.4$ seconds. This means the robot would have roughly 4.4 seconds less from a total of 10 minutes. In speedruns, the performance of the robot is not affected, as it doesn't have to do any exploration.

5.3 RoboSTEAM hackathon

The developed simulator was used in the RoboSTEAM [59] Hackathon, that is part of the Erasmus+ KA2 project [60]. It took place in Laboratório de controlo, automação e

Figure 5.16: Activities during the RoboSTEAM hackathon.

robótica (LCAR), in the Instituto Politécnico de Bragança (IPB) building, on November 3rd, 2020. Figure 5.16 shows a photograph taken during the event.

In this event, the basic usage of the simulator was explained to two groups of international students, and later they were presented the challenge scenarios described in Sections 5.1.1, 5.1.2 and 5.1.3, along with template code that provided a starting point for implementing solutions for the challenges.

One pertinent consideration was made during the development of a solution for the wall distance challenge: the reading of the left ToF sensor does not directly correspond to the distance between the robot and the wall, due to the diagonal sensor placement. This makes it evident that the physical structure of the robot is related to the meaning of a sensor's readings.

During the development of their code, the students noticed the inability to access the values of the stored variables and debug their code, as their attempt to use Arduino's `Serial.println` method call would cause the simulator to malfunction.

Chapter 6

Conclusions and Future Work

This work presented a simulator with dynamics for the micromouse contest. Sensors and actuators are embedded into the simulator, that can be interacted with through the use of a microcontroller connected to the computer's serial port, in a HIL based approach. This way, limitations such as memory requirements or processing power of the real hardware can be detected.

There have been two works based on this dissertation. One work, entitled "A Micromouse Scanning and Planning Algorithm based on Modified Floodfill Methodology with Optimization" [52] has been published on 20th IEEE International Conference on Autonomous Robot Systems and Competitions and has had an oral presentation. The other work, entitled "Micromouse 3D simulator with dynamics capability: a Unity environment approach" is an extended version of the previous publication and is under the submission process for the SN Applied Sciences journal.

The simulator can be used to compare and analyse different algorithms for the micromouse contest, as is the case of the proposed modified flood fill algorithm. Although it has been implemented in the simulator, further validation with a real robot is desired. The developed models for the robot also require comparisons with their real counterparts for validation.

Although the simulator is an interesting tool to test the performance of algorithms without the need of a real robot or maze structure, some limitations are evident. For

example, it lacks debugging tools for visualizing the robot path and inspecting the values for microcontroller code variables.

As future work, the simulator could implement different robot components, such as Inertial measurement units (IMUs). Also, it could be extended to support more robots in the same environment and more robot types, such as drones or robots with different wheel configurations.

Bibliography

[1] Universidade de Trás-os-Montes e Alto Douro, *Portuguese Micromouse*, `https://micromouse.utad.pt/index.php/en/`.

[2] P. V. F. Zawadniak, *GitHub: UniMouse repository*, `https://github.com/PDR5/MicromouseUnity`, Online; accessed 28 November 2020.

[3] P. Sarhadi and S. Yousefpour, "State of the art: Hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software," *International Journal of Dynamics and Control*, vol. 3, Jan. 2014. DOI: `10.1007/s40435-014-0108-3`.

[4] H. M. Zhang, L. S. Peh, and Y. H. Wang, "Micromouse solve maze based on flood-fill algorithm," *Applied Mechanics and Materials*, vol. 513–517, pp. 4227–4230, 2014.

[5] D. Oppliger, "Using first lego league to enhance engineering education and to increase the pool of future engineering students (work in progress)," in *32nd annual frontiers in education*, IEEE, vol. 3, 2002, S4D–S4D.

[6] C. Ferrada, F. J. Carrillo-Rosúa, D. Díaz-Levicoy, and F. Silva-Díaz, "La robótica desde las áreas stem en educación primaria: Una revisión sistemática," *Education in the Knowledge Society (EKS)*, vol. 21, p. 18,

[7] T. Bräunl, *Robot Adventures in Python and C*. Springer, 2020.

[8] L. Eckert, L. Piardi, J. Lima, P. Costa, A. Valente, and A. Nakano, "3d simulator based on simtwo to evaluate algorithms in micromouse competition," in *World Conference on Information Systems and Technologies*, Springer, 2019, pp. 896–903.

[9] R. Allan, "Microprocessors: The amazing micromice: See how they won: Probing the innards of the smartest and fastest entries in the amazing micro-mouse maze contest," *IEEE spectrum*, vol. 16, no. 9, pp. 62–65, 1979.

[10] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2018.

[11] J. Cai, X. Wan, M. Huo, and J. Wu, "An algorithm of micromouse maze solving," in *2010 10th IEEE International Conference on Computer and Information Technology*, IEEE, 2010.

[12] Mack Mackorone, *GitHub: A Micromouse simulator*, `https://github.com/mackorone/mms`, Online; accessed 27 July 2020, 2019.

[13] Miguel Peque, *GitHub: Micromouse Maze Simulator server*, `https://github.com/Bulebots/mmsim/`, Online; accessed 27 July 2020, 2018.

[14] J. C. M. Huo J. Wu and B. Song, "Micromouse competition training method based on 3d simulation platform," in *2010 10th IEEE International Conference on Computer and Information Technology*, IEEE, 2010, pp. 2174–2179.

[15] L. Piardi, L. Eckert, J. Lima, P. Costa, A. Valente, and A. Nakano, "3d simulator with hardware-in-the-loop capability for the micromouse competition," in *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, IEEE, 2019, pp. 1–6.

[16] S. Mishra and P. Bande, "Maze solving algorithms for micro mouse," in *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*, IEEE, 2008, pp. 86–96.

[17] R. Klein and T. Kamphans, "Pledge's algorithm-how to escape from a dark maze," in *Algorithms Unplugged*, Springer Berlin Heidelberg, 2011, pp. 69–75.

[18] M. H. J. Cai J. Wu and J. Huang, "A micromouse maze sovling simulator," in *2010 2nd International Conference on Future Computer and Communication*, 2010, pp. V3 683–689.

[19] Ł. Bienias, K. Szczepański, and P. Duch, "Maze exploration algorithm for small mobile platforms," *Image Processing & Communications*, vol. 21, no. 3, pp. 15–26, 2017.

[20] M. Sharma and K. Robeonics, "Algorithms for micro-mouse," in *2009 International Conference on Future Computer and Communication*, IEEE, 2009, pp. 581–585.

[21] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

[22] G. Law, "Quantitative Comparison of Flood Fill and Modified Flood Fill Algorithms," *International Journal of Computer Theory and Engineering*, vol. 5, no. 3, pp. 503–508, 2013, ISSN: 17938201. DOI: 10.7763/ijcte.2013.v5.738.

[23] Vipul Aggarwal, "Optimization of Flood Fill Algorithm Using Iterative Look-Ahead and Directional Technique," *International Journal of Computer Science and Engineering (IJCSE)*, vol. 2, no. 5, pp. 89–94, 2013.

[24] Z. Cai, L. Ye, and A. Yang, "FloodFill maze solving with expected toll of penetrating unknown walls for micromouse," *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications, HPCC-2012 - 9th IEEE International Conference on Embedded Software and Systems, ICESS-2012*, pp. 1428–1433, 2012. DOI: 10.1109/HPCC.2012.209.

[25] S. Tjiharjadi, M. Wijaya, and E. Setiawan, "Optimization maze robot using a* and flood fill algorithm," *International Journal of Mechanical Engineering and Robotics Research*, vol. 6, pp. 366–372, Sep. 2017. DOI: 10.18178/ijmerr.6.5.366-372.

[26] Unity Technologies, *Multiplatform Unity*, https://unity.com/features/multiplatform, Online; accessed 28 November 2020.

[27] ——, *Unity Manual: Physics*, https://docs.unity3d.com/Manual/PhysicsSection.html, Online; accessed 28 November 2020.

[28] ——, *Unity Manual: Unity's Interface*, https://docs.unity3d.com/Manual/UsingTheEditor.html, Online; accessed 28 November 2020.

[29] ——, *Unity Manual: GameObjects*, https://docs.unity3d.com/Manual/GameObjects.html, Online; accessed 28 November 2020.

[30] ——, *Unity Manual: Using Components*, https://docs.unity3d.com/Manual/UsingComponents.html, Online; accessed 28 November 2020.

[31] ——, *Unity Manual: Creating and Using Scripts*, https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html, Online; accessed 28 November 2020.

[32] ——, *Unity Manual: Order of Execution for Event Functions*, https://docs.unity3d.com/Manual/ExecutionOrder.html, Online; accessed 28 November 2020.

[33] ——, *Unity Manual: Rigidbodies Overview*, https://docs.unity3d.com/Manual/RigidbodiesOverview.html, Online; accessed 28 November 2020.

[34] ——, *Unity Manual: Colliders*, https://docs.unity3d.com/Manual/CollidersOverview.html, Online; accessed 28 November 2020.

[35] ——, *Unity Manual: Joints*, https://docs.unity3d.com/Manual/Joints.html, Online; accessed 28 November 2020.

[36] ——, *Unity Scripting API: Physics*, https://docs.unity3d.com/ScriptReference/Physics.html, Online; accessed 28 November 2020.

[37] ——, *Unity Manual: Prefabs*, https://docs.unity3d.com/Manual/Prefabs.html, Online; accessed 28 November 2020.

[38] ——, *Unity Manual: Scriptable Objects*, https://docs.unity3d.com/Manual/class-ScriptableObject.html, Online; accessed 28 November 2020.

[39] W. Bolton, *Mechatronics : electronic control systems in mechanical and electrical engineering*. Harlow, England New York: Pearson, 2015, ISBN: 978-1-292-07668-3.

[40] S. Chapra and R. Canale, *Numerical methods for engineers*. New York, NY: McGraw-Hill Education, 2015, ISBN: 978-0-07-339792-4.

[41] *Micro Metal Gearmotors*, Rev. 4.2. Available at `https://www.pololu.com/file/0J1487/pololu-micro-metal-gearmotors-rev-4-2.pdf`, Pololu Corporation, Dec. 2019.

[42] E. Williams, *Make, AVR programming*. Sebastopol, CA: Maker Media, 2014, ISBN: 978-1-449-35578-4.

[43] M. Mazidi, R. Mckinlay, and D. Causey, *PIC microcontroller and embedded systems : using Assembly and C for PIC18*. Upper Saddle River, N.J: Pearson Prentice Hall, 2008, ISBN: 0-13-600902-6.

[44] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. Cambridge, Mass: MIT Press, 2011, ISBN: 978-0-262-01535-6.

[45] Pololu, *Magnetic encoder pair kit for micro metal gearmotors*, `https://www.pololu.com/product/3081`, Online; accessed 28 November 2020.

[46] STMicroelectronics, *Vl53l0x: Time-of-flight ranging sensor*, `https://www.st.com/en/imaging-and-photonics-solutions/vl53l0x.html`, Online; accessed 28 November 2020.

[47] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. Cambridge, Massachusetts: The MIT Press, 2006, ISBN: 978-0-262-20162-9.

[48] Pololu, *Vl53l0x time-of-flight distance sensor carrier with voltage regulator, 200cm max*, `https://www.pololu.com/product/2490`, Online; accessed 28 November 2020.

[49] STMicroelectronics, *World smallest time-of-flight ranging and gesture detection sensor application programming interface*, `https://www.st.com/resource/en/data_brief/stsw-img005.pdf`, Online; accessed 28 November 2020.

[50] ——, *Time-of-flight ranging sensor*, `https://www.st.com/resource/en/datasheet/vl53l0x.pdf`, Online; accessed 28 November 2020.

[51] B. Siciliano, *Springer handbook of robotics*. Berlin: Springer, 2016, ISBN: 978-3-319-32550-7.

[52] P. Zawadniak, L. Piardi, T. Brito, J. Lima, P. Costa, A. L. Regis Monteiro, and A. Pereira, "A micromouse scanning and planning algorithm based on modified floodfill methodology with optimization," Apr. 2020, pp. 245–250. DOI: 10.1109/ICARSC49921.2020.9096193.

[53] Peter Harrison, *Adachi micromouse maze solving algorithms*, http://www.micromouseonline.com/2008/05/27/adachi-micromouse-maze-solving-algorithms/, Online; accessed 28 November 2020, 2008.

[54] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612. [Online]. Available: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[55] L. T. Eckert, "Development of an Autonomous Mobile Robot with Planning and Location in a Structured Environment," M.S. thesis.

[56] Wemos, *GitHub: LOLIN I2C Motor library*, https://github.com/wemos/LOLIN_I2C_MOTOR_Library, Online; accessed 28 November 2020, 2018.

[57] ——, *Motor shield*, https://docs.wemos.cc/en/latest/d1_mini_shield/motor.html, Online; accessed 28 November 2020, 2019.

[58] Peter Harrison, *GitHub: micromouseonline mazefiles*, https://github.com/micromouseonline/mazefiles, Online; accessed 15 February 2020, 2013.

[59] *RoboSTEAM*, Online; accessed 28 November 2020.

[60] *Erasmus+ KA2: cooperation for innovation and exchange of good practices*, https://eacea.ec.europa.eu/erasmus-plus/actions/key-action-2-cooperation-for-innovation-and-exchange-good-practices_en, Online; accessed 28 November 2020.