Research Article

Micromouse 3D simulator with dynamics capability: a Unity environment approach

Pedro V. F. Zawadniak[1,3] · **Luis Piardi**[1] · **Thadeu Brito**[1] · **José Lima**[1,2] · **Pedro Costa**[2] · **André L. R. Monteiro**[3] · **Paulo Costa**[2] · **Ana I. Pereira**[1]

Received: 2 August 2020 / Accepted: 18 January 2021 / Published online: 2 February 2021

Abstract
The micromouse competition has been gaining prominence in the robotic atmosphere, due to the challenging and multidisciplinary characteristics provided by the teams' duels, being a gateway for those who intend to deepen their studies in autonomous robotics. In this context, this paper presents a realistic micromouse simulator developed with Unity software, a widely game engine with dynamics and 3D development platform used. The developed simulator has hardware-in-the-loop capabilities, aims to be simple to use, it can be customizable, and designed to be as similar as possible to the real robot configurations. In this way, the proposed simulator requires few modifications to port the microcontroller code to a real robot. Therefore, the framework presented in this work allows the user to simulate the development of new algorithm strategies dedicated to competition and also hardware updates. The simulation supports several mazes, from previous competitions and has the possibility to add different mazes elaborated by the user. Thus, the features and functionality of the simulator can serve to accelerate the project's development of the beginning and advanced competitors, using real models to reduce the gap between the mouse robot behavior in the simulation and the reality. The developed simulation environment is available to the community.

Keywords Micromouse · Robotics competition · Mobile robot · Simulation · Unity

1 Introduction

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 [1].

Adopting autonomous robots to explore unknown mazes can be fun and challenging, representing a unique tool to multidisciplinary and cognitive activity [2]. 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.

By this way, the micromouse challenge addresses a problem in 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 [3]. 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 lowest time

✉ Pedro V. F. Zawadniak, fzpedro@outlook.com; Luis Piardi, piardi@ipb.pt; Thadeu Brito, brito@ipb.pt; José Lima, jllima@ipb.pt; Pedro Costa, pedrogc@fe.up.pt; André L. R. Monteiro, almonteiro@utfpr.edu.br; Paulo Costa, paco@fe.up.pt; Ana I. Pereira, apereira@ipb.pt | [1]Research Center in Digitalization and Intelligent Robotics (CeDRI), Instituto Politécnico de Bragança, Bragança, Portugal. [2]Centre for Robotics in Industry and Intelligent Systems - INESC TEC, Faculty of Engineering of University of Porto (FEUP), Porto, Portugal. [3]Federal University of Technology - Paraná, Campo Mourão, Brazil.

possible [4]. Each robot gets a total of 10 min for the competition (exploration and find the best route), and whenever it returns to the start square a new run timer is started only its shortest run counts.

The micromouse competition begins in New York City [5] 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 Hardware-in-the-loop (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 [6]. 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.

Despite being widespread worldwide, micromouse competition presents some obstacles for new competitors and enthusiasts, not having a official simulator that represents 3D components, representations of dynamic models, and sensors for localization besides the possibility of applying these features in different mazes. In this sense, it becomes complex to implement codes and strategies that can be reproduced in real environments, resulting in a delay in real implementation due to errors that cannot be detected only in 2D simulators.

This work aims to develop a micromouse simulator in Unity, to enhance the experience of competing teams in the micromouse. The simulator has all the requirements and 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. Thus, the simulation shows the strategy performance inserted by the competitors respecting the microcontroller constraints used. Besides, a typical robot model and the floodfill 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 was developed for the Windows platform and is open-source. It is available at the repository [7]. The repository contains the source code, a built scene and the Arduino library that interfaces with the microcontroller.

This work is an extended version of the paper "A Micromouse Scanning and Planning Algorithm based on Modified Floodfill Methodology with Optimization" [8]. The previous work proposed a modification to the floodfill algorithm tested only in a 2D simulator comparing the path efficiency with the original algorithm applied in the competition. In this work, the algorithm proposed in [8] was tested in the 3D simulation environment presented in this paper in hardware-in-the-loop mode, therefore considering robot dynamics and scenario friction.

This paper is organized as follows. After an introduction in Sect. 1, the state of the art is presented in Sect. 2 with a review about micromouse simulators. In Sect. 3, the software Unity and its tools are described. Next, in Sect. 4, the details about fundamental parameters are specified, and Sect. 5 demonstrates a run with the robot model available to beginners, guided by the exploration algorithm proposed in Sect. 6. The obtained results are presented in Sect. 7. Finally, Sect. 8 concludes the paper and points some future work direction.

2 State of the art: review of micromouse simulators

Over time, simulation has been playing an increasingly important role in robotics. With the use of simulators, more development and testing can be explored without increasing the costs, giving the possibility to identify the optimal procedures [10].

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, dynamics, the inclusion of sensors and images more similar to the real ones. The simulators can handle all the electrical and physical characteristics, 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 Hardware-in-the-loop 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 of maturity that is almost not necessary to have a real maze to prepare a team.

3 Unity overview

The Unity is a framework for generating applications with 3D scenarios with dynamics, that are supported by several platforms such as WebGL, Windows, Mac and Linux distributions [16]. Beyond its dynamics, this framework was chosen since it provides an interactive editor with scripts, control and editing sections as presented in Fig. 1. It also generates standalone applications that can be executed without any development tools and allow access to hardware, as a main requirement for the proposed scenario.

3.1 Physics engine

Unity utilizes NVIDIA's open-source PhysX engine for physics simulation [17]. Amongst other functionality, the engine allows for 3D rigid body simulation. There are two essential components for a physics simulation: colliders and rigid bodies.

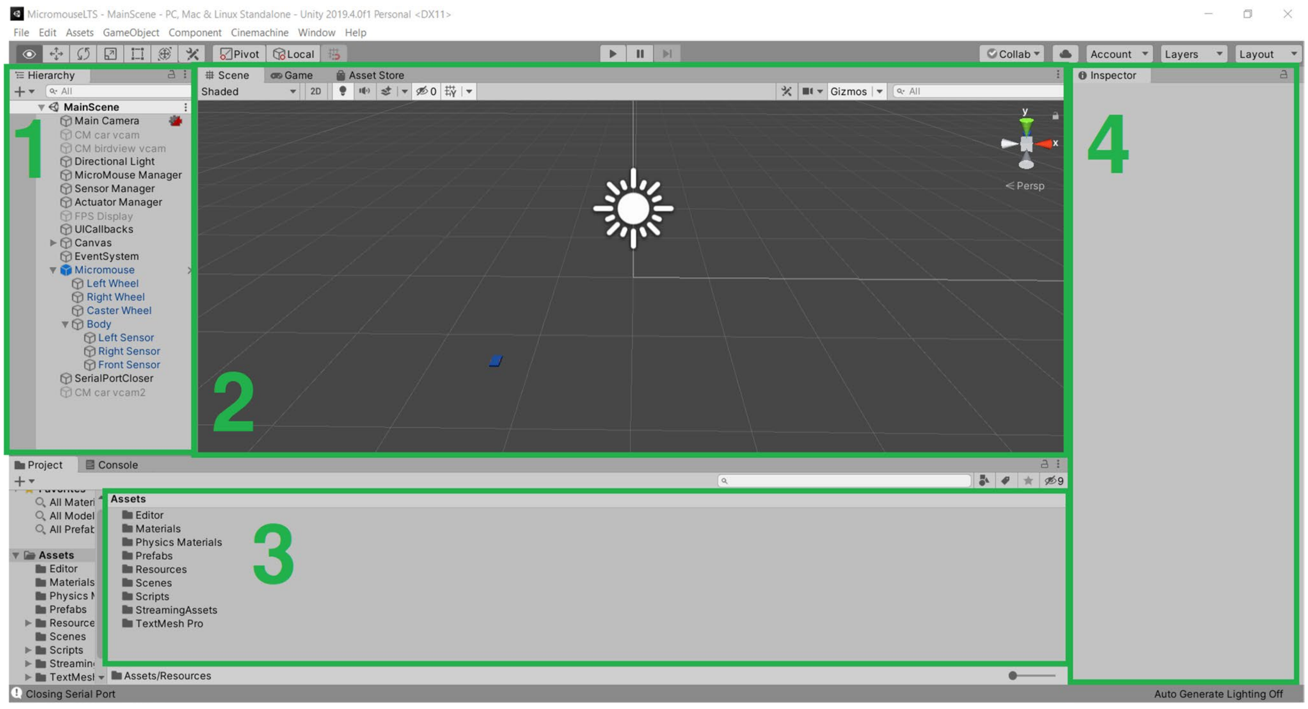


Fig. 1 The Unity Editor where the highlighted and labeled windows have the following role: ① The hierarchy window shows all `GameObjects` present in the Scene. They may be independent or exist as a child object, which indicates that their position and rotation are linked to their parent's; ② The Scene View allows for viewing and editing of the scene; ③ The project view contains all assets present in the projects. Amongst other things, game assets consist of all scripts, materials, audio files, configuration files and prefabs, which are pre-built `GameObjects`; ④ The inspector window shows the properties of a selected `GameObject`. It can be used to see and modify its parameters [9]

Colliders specify the object shape for the purposes of the physics simulation that handle the collisions with other objects or with themselves. Rigid Body component defines physical aspects of the game object, such as its mass and drag. The physics engine will only move objects that have a rigid body component attached.

It is possible to query the physics engine for information using the Unity physics scripting Application Programming Interface (API) [18]. For the micromouse simulator, the most relevant information that the physics engine provides is the position and speed of any `GameObject` and the result of performing a ray cast operation that determines with which objects a ray, with a given origin and direction, collides. These pieces of information are what makes the implementation of the encoders and time-of-flight (ToF) sensors possible.

The physics engine also responds to force and torque inputs. This is essential for implementing motor actuators. A joint system allows for connections between rigid bodies. There are many kinds of joints available, but the one used in this simulation is the Hinge Joints, which constrain movement in all degrees of freedom, except for one rotational axis. These are used to attach the wheels to the robot's body [19]. The model of the robot was built on the simulation environment with the same dimensions, parts and mass as the real one. A comparison focused on the dynamics between the real robot and the simulated one will be addressed further.

4 Simulation software: features

As mentioned before, the simulator includes representation of a real micromouse robot to help beginners. Its physical properties, sensor and actuator characteristics have predefined values, which may be edited in the Unity Simulator's Inspector window. Figure 2 presents the Unified Modeling Language (UML) class diagram for the sensors and actuators. All attributes represent specification parameters that can be customized by the users.

4.1 Hardware in the loop system

The Simulator has the ability to communicate with a microcontroller through the serial port. Figure 3 presents the hardware-in-the-loop architecture, in which the developed simulator is prepared to support. The microcontroller can be the same that will be embedded in the real robot. It is responsible for analyzing the data from the distance sensors and encoder, updating the location, and making decisions according to each team's path planning algorithm. Consequently, the microcontroller will assign a voltage level that will be applied to the robot's DC motors, resulting in the direction and speed applied to the robot, that is, the movement.

The simulator abstracts and represents the robot's model, the characteristics of the motor DC, the distance values obtained by each sensor embedded in the simulated robot, and the information from the encoders to contribute to the location system. Besides, it also has several mazes to test the algorithms implemented in the

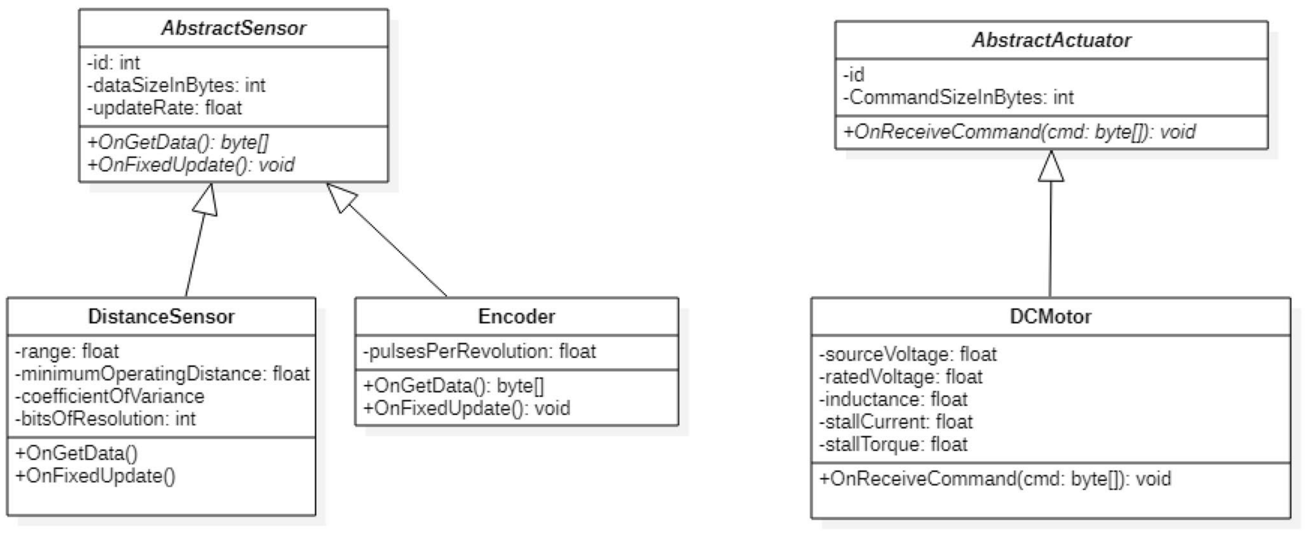


Fig. 2 The UML class diagram for the for the sensors and actuators. The concrete component definitions derive from abstract classes that define common properties and behaviours

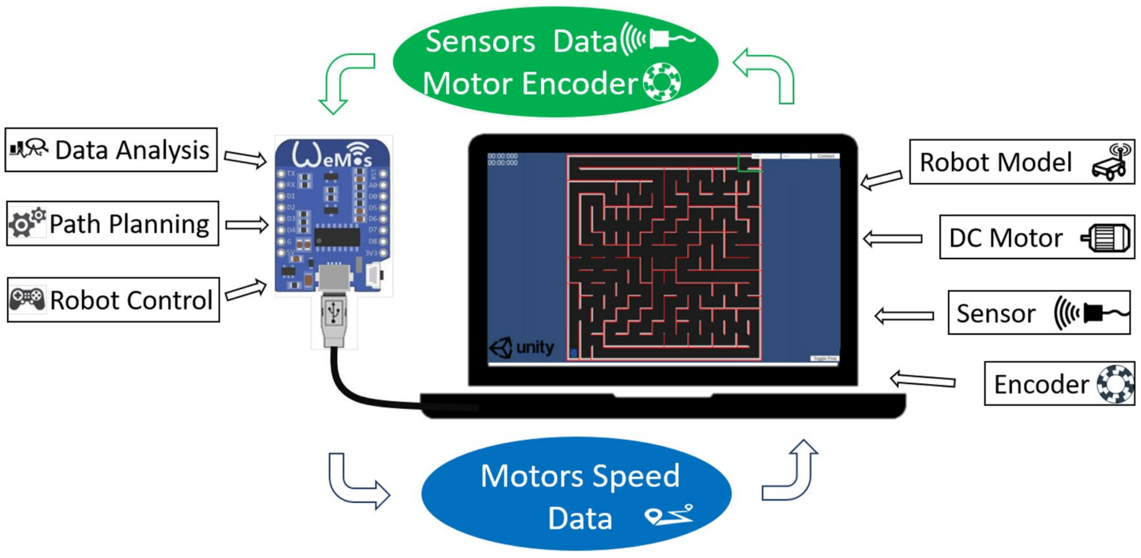


Fig. 3 System architecture of the hardware in the loop system

Table 1 Physical dimensions of the real robot (from [20]) and its simulated model

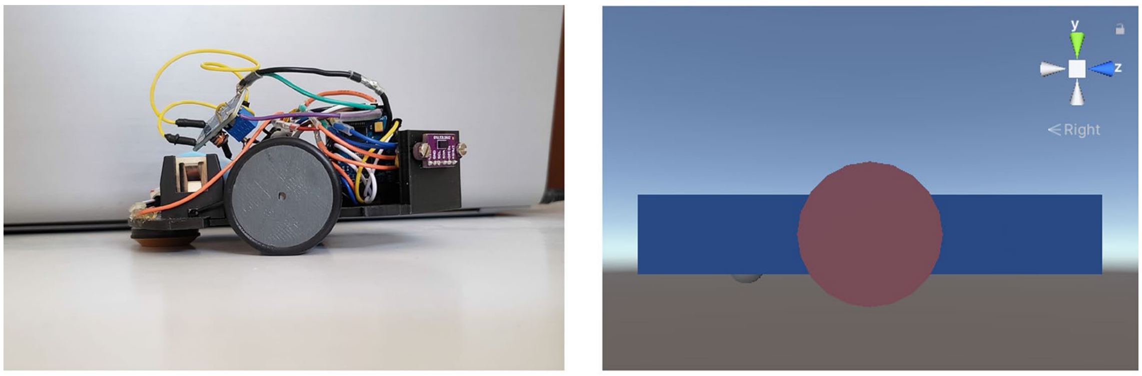
Parameter	Real robot	Simulation model (m)
Width	0.096 m	0.096
Length	0.120 m	0.120
Height	Unspecified	0.02
Wheel diameter	0.032 m	0.032
Wheel thickness	0.008 m	0.008

microcontroller for different scenarios. This feature that the developed simulator presents can accelerate prototypes for the micromouse competition, being a good methodology to develop and validate the strategies due to the speed of development and cost before migrating the tests to the real robot.

4.2 Real and simulation robot characteristics

The robot model provided has two DC motors, wheel encoders and three ToF distance sensors. The simulated robot dimensions is approximate to the real robot, shown in Table 1. The real robot (show in Fig. 4a), developed by Eckert [20] weighs approximately 157 g, with batteries included. In the simulated model, the body weighs 127 g, and both side wheels and the caster weigh 10 geach.

In the simulation, the robot body has a box shape, with dimensions 96 cm $\times$ 120 cm $\times$ 2 cm. All three distance sensors are placed at the frontal side: one of them is centered, facing forward, and the other two are placed in at the left



(a) Real robot prototype. **(b)** Simulated robot

Fig. 4 Visual comparison between the real robot and the simulated robot applied to the micromouse competition

and right corners, facing the diagonals at 45°. Therefore, the simulated robot is displayed in Fig. 4b.

4.3 Sensors and actuators

The simulator has implementations of incremental encoders, ToF sensors and DC motors. Each sensor and actuator is implemented as a `MonoBehaviour` script and should be attached to the rigid body that they depend on (respecting the hierarchy).

4.3.1 DC motor

The developed robot is equipped with two DC motors coupled with gear motor. The modelled DC Motor can be described in next equations, where *i* is the armature current, *R* is the armature resistance, *L* is the inductance and V_b is the counter-electromotive force [21].

$$V = Ri + L\frac{di}{dt} + V_b \quad (1)$$

The motor applies torque (*T*) to the robot wheel's, dictated by the relation expressed in Eq. 2. It depends on the motor constant K_t and the armature current. To calculate the current (i_t) flowing through the motor armature at each discrete time-step, defined by Δt, a first-order backward difference model is used, resulting in Eq. 3. The current depends on the torque constant K_t, resistance *R*, applied voltage *V* wheel angular speed ω and the current at the previous time-step i_{t-1}. The motor constant and resistance are calculated with Eqs. 4 and 5, using the values for the motor rated voltage V_{rated}, stall torque T_{stall} and stall current i_{stall}, which can be provided in the component's datasheet or measured empirically. The wheel speed is reported by the physics engine and the applied voltage depends on the user's input.

$$T = K_t i \quad (2)$$

$$i_t = \frac{V - K_t\omega + \frac{L}{\Delta t} i_{t-1}}{R + \frac{L}{\Delta t}} \quad (3)$$

$$K_t = \frac{T_{\text{stall}}}{i_{\text{stall}}} \quad (4)$$

$$R = \frac{V_{\text{rated}}}{i_{\text{stall}}} \quad (5)$$

The simulation model in the simulator takes 11-bit signed integer as a control parameter. The sign bit indicates the movement direction and the other 10 bits represent how much voltage from the battery should be applied to the armature (such as a PWM).

4.3.2 Incremental encoder

The incremental encoder measures the pulses according to the wheels' change in angle. The number of pulses in one wheel revolution is set through the Pulses Per Revolution (PPR) parameter. The change in angle at each time-step is given by Eq. 6. The pulses are calculated by Eq. 7, which takes into account the change in angle and the encoders' characteristics [22].

$$\Delta\theta = \omega\Delta t \quad (6)$$

$$\Delta\text{Pulses} = \frac{\Delta\theta}{2\pi} \times \text{PPR} \quad (7)$$

4.3.3 Time-of-flight distance sensors

The ToF distance sensors measures how much time a beam takes to bounce off of an obstacle and goes back to the sensor [23]. If an object is too far away, or the reflected beam doesn't go back to the sensor, the sensor will time-out, and the reported reading value will be invalid. The sensor model has the following configurable parameters: minimum distance, which indicates the sensor's blind spot; range, which means the maximum measurement range; and coefficient of variance, which sets the intensity of the Gaussian noise added to the measurement.

The distance sensors query the physics engine for a ray cast operation, starting at their current position and towards their heading direction. The returned value is equivalent to an ideal measurement. The measurement is processed before it is sent to the microcontroller. Firstly, it is clamped between the minimum operation distance and the measurement itself. Then, Gaussian noise is added, according to the coefficient of variance parameter. And lastly, the result is clamped between 0 and the sensor range.

4.4 Supported maze files

Maze files are generated from a human-readable maze description text file, located in ./StreamingAssets/Mazes/maze.txt, relative to the simulator folder. The accepted file format uses the "o" character to represent posts, which are required at the corners of every maze cell. Horizontal and vertical walls are represented by the "– -" and "|" character strings, respectively. A database of such files is maintained on micromouseonline's repository [24].

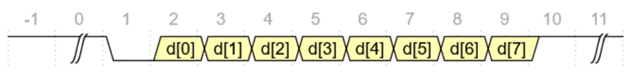


Fig. 5 Timing diagram for the 8N1 serial communication setting

Table 2 The time it takes to transmit a byte of data with the 8N1 protocol (8 bits of data, 2 bits of overhead)

Baud rate	Time per byte of data (μs)
9600	1042
115,200	86.81
12,000,000	0.8333

4.5 Serial communication protocol

The communication between the microcontroller and simulator is established via the serial port, using the 8N1 (8 data bits, no parity bit, one stop bit) configuration setting [25]. Figure 5 shows a packet of data transmitted with the 8N1 setting. The actual data bits have labels "2" through "9", while the start and stop bits are labeled "1" and "10" respectively.

In this setting, there are 2 bits of overhead for every 8 bits of data. Effectively 10 bits are transmitted for every byte of data. The simulator listens for actuator commands through the serial port. It expects one byte for the actuator ID, followed by however many data bytes that actuator is configured to receive. Besides the actuator ID, the DC motors support 11-bit commands, which are transmitted over two bytes. Therefore, sending a voltage command to a DC motor takes 3 bytes.

Every new measurement performed by a sensor will be transmitted through the serial port. The simulator sends one byte with the sensor ID, followed by the sensor measurement data. The encoders send 4 bytes of data, while the distance sensors send 2 bytes of data. The time it takes to transmit one byte of data is summarized in Table 2, for some standard baud rates.

As the robot consists of the integration of several devices, Table 3 lists the number of bytes needed for each of these devices to send data and time transmission.

5 Using the simulator

There are three elements to the user interface, demonstrated by Figure 6. There is the "Toggle post" button, at the lower-right corner of the screen. It toggles whether the maze has a post at its center. In micromouse competitions, it is usually up to the competitor to have placed or not.

Table 3 The time it takes to transmit data for every robot component (baud rate set to 115,200 bps)

Component	No. of bytes	Transmission time (μs)
DC motor	3	260.4
Distance sensor	3	260.4
Encoder	5	434.0

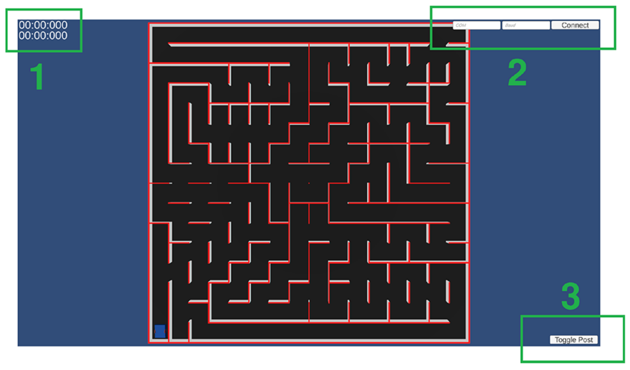


Fig. 6 ① Global (top) and current lap (bottom) timers; ② Serial port configuration; ③ Center- maze post toggling

In the upper-left corner, there are two timers. The first one is the global timer, which tracks how much time has passed since the robot left the starting area for the first time. The second is the "lap timer", which tracks how much time has passed since the robot started the current run towards the maze. The "lap timer" stops when the robot reaches the goal, and resets when the robot returns to the starting position. After the 10-min mark, the timers will turn red, indicating that the allocated time for the competitor has run out.

Two text fields and a "Connect" button are placed in the upper-right corner of the screen. The text fields are used to set the configuration parameters of the serial communication. One text field sets the port name and the other sets the baud rate. Once they are set, confirm by pressing the "Connect" button. Once the connection has been established, the simulator will listen to commands sent through the serial port. The first command should be `requestEnableSensors`, which enables serial output containing sensor readings. The referred command is defined in the provided Arduino libraries, along with the DC motor commands and sensor reading commands.

The user may also change the camera position, to observe the robot from different perspectives. Three cameras are available, static birdview (Fig. 6), following the robot, as Fig. 7, and close-up view, as Fig. 8.

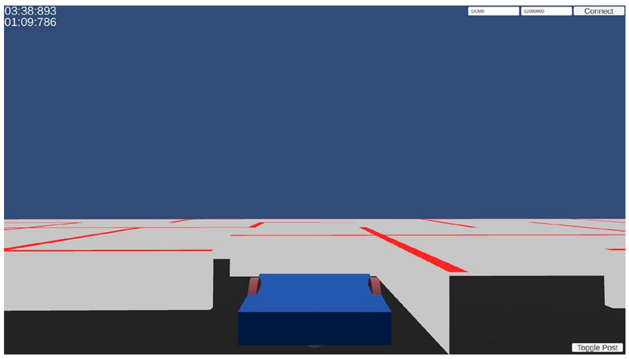


Fig. 7 Camera 2 view (robot's perspective). Emphasizes how smooth the robot trajectory is

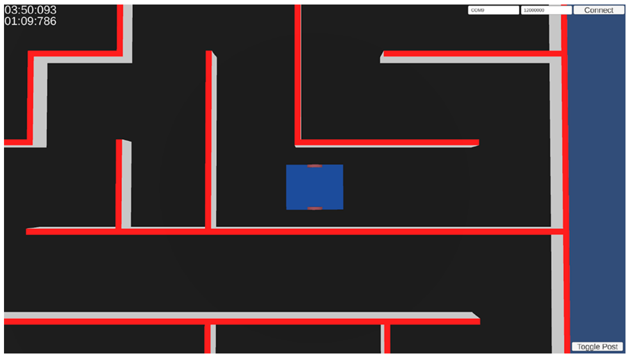


Fig. 8 Camera 3 (top view). It makes it easier to see how distant the robot is from the walls

An Arduino-compatible library provides a set of classes to interact with the simulator. Two classes, namely the sensor manager and the actuator manager, that handle receiving and sending serial data, respectively. There is also one class specifically for each robot component that is, motors, encoders and distance sensors, which control all interactions with the robot and the environment. There are example sketches provided with the simulator, explained with comments in the code. A UML diagram shows the public API of the library on Fig. 9. Usage examples and documentation are provided as Arduino sketches.

The motors designed as `GameObject`, accepts voltage commands via the `setVoltage` parameters. It expects an 11-bit, signed integer value. The accumulated encoder pulses are accessible by the `.getCounts` method, and accumulated total noise can be reset to zero with the `.reset method`. Besides, distance sensors (also based on `GameObject`) provide two pieces of information, through the `.getReading` method: the measured distance and whether a timeout has happened.

The simulator will only transmit sensors data after the sensors have been enabled. To do so, the user must call the `requestEnableSensors` method on the `SensorManager` object, after the serial port connection has been established.

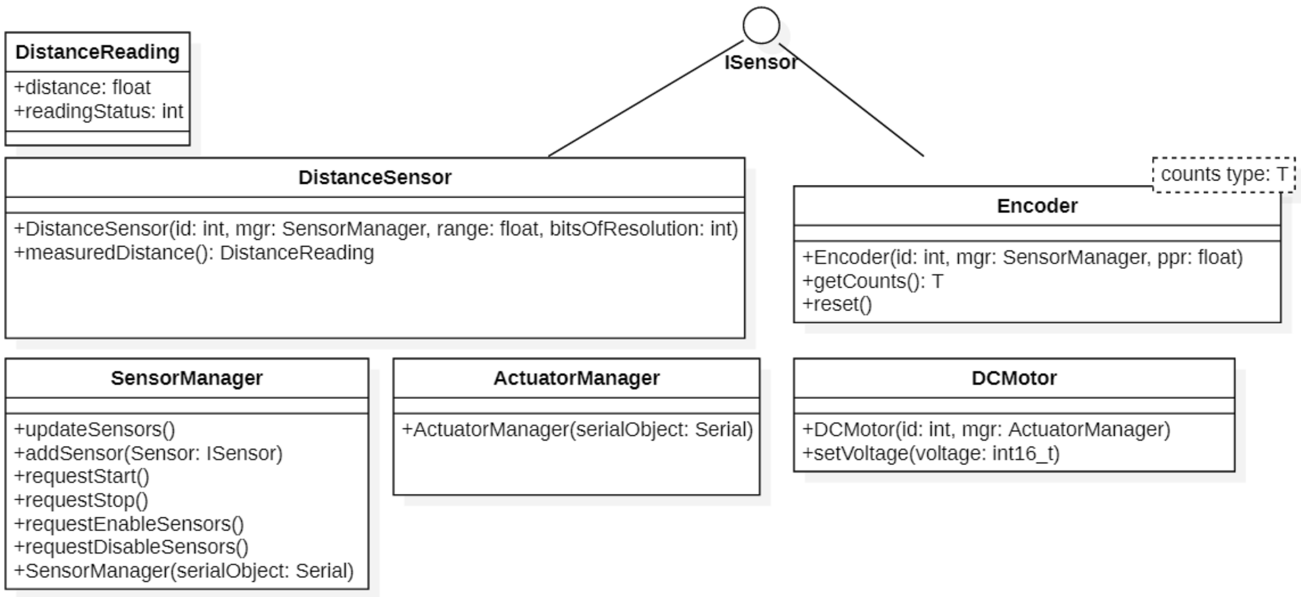

ISensor
DistanceReading
+distance: float
+readingStatus: int
DistanceSensor
+DistanceSensor(id: int, mgr: SensorManager, range: float, bitsOfResolution: int)
+measuredDistance(): DistanceReading
counts type: T
Encoder
+Encoder(id: int, mgr: SensorManager, ppr: float)
+getCounts(): T
+reset()
SensorManager
+updateSensors()
+addSensor(Sensor: ISensor)
+requestStart()
+requestStop()
+requestEnableSensors()
+requestDisableSensors()
+SensorManager(serialObject: Serial)
ActuatorManager
+ActuatorManager(serialObject: Serial)
DCMotor
+DCMotor(id: int, mgr: ActuatorManager)
+setVoltage(voltage: int16_t)

Fig. 9 The UML class diagram for the Arduino library that provides the interface between the hardware and the simulator

6 The floodfill algorithm

In any given moment, the robot's position is defined by a its configuration *c*. In the traditional Floodfill algorithm, the configuration is given by the (x, y) position of the cell it is currently in.

For every configuration, it is considered that the robot can reach another set of configurations. The set of configurations that (x, y) can reach is $(x \pm 1, y)$ or $(x, y \pm 1)$, as long as there are no walls blocking. That means the robot can only move north, south, east or west.

The algorithm assigns a given cost to each configuration. The cost represents the lowest cost of all sequence of actions that will take the robot from that particular configuration to a goal configuration. For instance, If the robot has to move north, east, then move north again, the cost will be 3.

After reaching the goal cells at the center of the maze, the cost assigned to configuration (0, 0) will indicate the shortest path the robot has found, according to a metric in which moving to any adjacent cell has the same cost. It doesn't take into account the robot's rotation time, a feature added to the new modified algorithm.

6.1 The modified algorithm

In the modified Floodfill algorithm, the configuration is defined to also include the rotation of the robot: besides the cell position (x, y), it also has a rotation state, which may be north, south, east or west. This representation is closely related to the movement of the robot we will develop. The algorithm 1 presents the developed approach that takes into account the rotations to find the best way to solve the micromouse mazes.

There is a set of configurations that the robot may reach from it's current configuration. It may rotate clockwise, counter clockwise and, if there isn't a wall it may also move forward. This is the set $S_1(c)$.

For example, assuming that the robot is in configuration $c = (1, 1, \text{East})$ and there isn't a wall in front of it. In this case, set $S_1(c)$ contains three other configurations: two of them are reached by rotating 90° clockwise or counterclockwise, namely (1, 1, North) and (1, 1, South). The other one is (1, 2, East), reached by moving forward.

Similarly, there is a set of configurations that may reach the current configuration. This is set $S_2(c)$. Considering the previous example, the set $S_2(c)$ would contain (1, 1, North) and (1, 1, South). They may reach (1, 1, East) by rotating clockwise and counterclockwise, respectively. $S_2(c)$ would also contain (0, 1, East), except if there is a wall blocking this path.

Consider the set of all goal configurations *G*. This will be the starting point of the algorithm. Configurations in *G* will be assigned cost 0 and will be used as a baseline for the cost assigned to all other configurations.

There is a First-in First-out queue *Q*, which stores configurations. Every configuration added to this queue will be processed in the same order they were added.

The set *P* of configurations have already been added to *Q*. This exists in order to assure that each configuration is added to the queue *Q* only once, and thus only once processed.

At each iteration, the algorithm processes a configuration **c** from the queue *Q*. This means that **c** will have a cost assigned, which is equal to the lowest cost of the configurations in $S_1(c)$ plus one. Also, every configuration in $S_2(c)$ that hasn't been already added to *P* will be added to *Q*, so it will be processed later on. This ensures that all accessible configurations will be processed at some point during execution.

Algorithm 1 Modified Floodfill Algorithm

```
Initialize $G$ as the goal configurations
Initialize $Q$ as an empty Queue
Initialize $P$ as an empty Set
Initialize $Cost(c) \leftarrow 0$, for all $c \in G$
Initialize $Cost(c) \leftarrow \infty$, for all $c \notin G$
for all $g \in G$ do
  Add $g$ to $Q$
  Add $g$ to $P$
end for
while $Q$ is not empty do
  Dequeue $c$ from $Q$
  for all $s_2 \in S_2(c)$ do
    if $s_2 \notin P$ then
      add $s_2$ to $Q$
      add $s_2$ to $P$
    end if
    if $c \notin G$ then
      $Cost(c) \leftarrow \min_{c' \in S_1(c)} Cost(c') + 1$
    end if
  end for
end while
```

The drawback of this modification is that there are four times more possible configurations and thus, more memory requirements.

7 Results

After the development of the simulation scenario based on the Unity platform, it is necessary to test the proposed models. In this way, the experiment intends to observe the micromouse simulator's performance as a whole, that is, to analyze the algorithm and robot templates, as well as the communication for HIL's execution. Thus, mazes

Table 4 Time taken for each run to be complete in All Japan's 2018 maze

Lap	Start time	End time	Run time
1	00:00	01:39	01:39
2	02:44	03:44	01:00
3	04:42	05:37	00:55
4	06:35	07:30	00:55

Table 5 Time taken for each run to be complete in All Japan's 2019 maze

Lap	Start time	End time	Run time
1	00:00	01:15	01:15
2	02:20	03:26	01:06
3	04:47	05:43	00:56
4	06:45	07:41	00:56

from previous competitions can serve as an alternative to compare the proposed simulator's results. In this sense, Tables 4 and 5 show the performance of the simulated micromouse for the mazes used in All Japan's micromouse event held in 2018 and 2019, using the described modified floodfill algorithm for exploration. Each lap starts when the robot leaves the starting position, and ends when it reaches one of the central positions.

A video [26] shows the robot performing the first run on All Japan's 2018 maze. Firstly, it sets the central maze cells as its goal. As the robot makes it to the center, it assigns the maze's starting position as the goal. By returning to the starting point, the robot may start it's next run.

After each run, the robot explores more parts of the maze until, eventually, the algorithm will have found what it considers to be the best path. At this point, the run time won't be significant changed, as it will always move through the same maze cells.

8 Conclusions and future work

The presented paper addressed a simulator environment with dynamics, based on Unity, to apply on the micromouse competition. The sensors and actuators are modelled and embedded in the simulator. Although, it is possible to change and modify the environment since it is available for free. Users can try it developing and testing their own algorithms or adjust the floodfill suggested one. The proposed simulator allows using the hardware-in-the-loop methodology to detect limitations of the real microcontroller and hardware capabilities. The presented results show that the developed and available simulator is an interesting solution to implement and test algorithms without the maze. As future work, it can be pointed out some points that could be addressed further, such as the model of sensors and motors accuracy as well as the dynamics of the simulated robot improvement, other algorithms and its comparison.

Funding Information This work has been supported by FCT - Fundaçã para a Ciência e Tecnologia within the Projects UIDB/05757/2020 and UIDB/50014/2020.

Availability of data and materials Not applicable.

Compliance with ethical standards

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Code availability The project files for the simulator and corresponding Arduino library are available at the repository [7].

References

1. Oppliger D (2002) 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, vol 3. IEEE, pp S4D–S4D
2. Ferrada C, Carrillo-Rosúa FJ, Díaz-Levicoy D, Silva-Díaz F (2020) La robótica desde las áreas stem en educación primaria: una revisión sistemática. In: Education in the knowledge society (EKS), vol 21, p 18
3. Bräunl T (2020) Robot adventures in Python and C. Springer, Berlin
4. Eckert L, Piardi L, Lima J, Costa P, Valente A, Nakano A (2019) 3d simulator based on simtwo to evaluate algorithms in micromouse competition. In: World conference on information systems and technologies. Springer, pp 896–903

5. Allan R (1979) 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 Spectr 16(9):62–65
6. Juliani A, Berges V-P, Vckay E, Gao Y, Henry H, Mattar M, Lange D (2018) Unity: a general platform for intelligent agents. arXiv preprint arXiv:1809.02627,
7. Zawadniak PVF et al (2020) Unity micromouse simulator repository. https://cloud.ipb.pt/d/4050c982b3ee4cf09a2f/
8. Zawadniak P, Piardi L, Brito T, Lima J, Costa P, Monteiro AL Regis, Pereira A (2020) A micromouse scanning and planning algorithm based on modified floodfill methodology with optimization, 04 pp 245–250
9. Unity Technologies (2020) Unity manual, unity's interface. https://docs.unity3d.com/Manual/UsingTheEditor.html. Accessed 27 July 2020
10. Ivaldi S, Padois V, Nori F (2014) Tools for dynamics simulation of robots: a survey based on user feedback
11. Cai MHJ, Wu J, Huang J (2010) A micromouse maze sovling simulator. In: 2010 2nd International conference on future computer and communication, pp V3 683–689
12. Mack Mackorone (2020) GitHub: a micromouse simulator. https://github.com/mackorone/mms. Accessed 27 July 2019
13. Miguel Peque (2020) GitHub: micromouse maze simulator server. https://github.com/Bulebots/mmsim/. Accessed 27 July 2018
14. Huo JCM, Wu J, Song B (2010) Micromouse competition training method based on 3d simulation platform. In: 2010 10th IEEE international conference on computer and information technology. IEEE, pp 2174–2179
15. Piardi L, Eckert L, Lima J, Costat P, Valente A, Nakano A (2019) 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, pp 1–6
16. Unity Technologies (2020) Unity manual, Gamebjects. https://docs.unity3d.com/Manual/GameObjects.html. Accessed 27 July 2020
17. Unity Technologies (2020) Unity manual, physics. https://docs.unity3d.com/Manual/PhysicsSection.html. Accessed 27 July 2020
18. Unity Technologies (2020) Unity scripting api, physics. https://docs.unity3d.com/ScriptReference/Physics.html. Accessed 27 July 2020
19. Unity Technologies (2020) Unity manual, joints. https://docs.unity3d.com/Manual/Joints.html. Accessed 27 July 2020
20. Eckert LT (2019) Development of an autonomous mobile robot with planning and location in a structured environment. Master's thesis, Polytechnic Institute of Bragança, Portugal
21. Scuiller F, Semail E (2014) Inductances and back-emf harmonics influence on the torque, speed characteristic of five-phase SPM machine. In: IEEE vehicle power and propulsion conference (VPPC). IEEE, pp 1–6
22. Petrella R, Tursini M, Peretti L, Zigliotto M (2007) Speed measurement algorithms for low-resolution incremental encoder equipped drives: a comparative analysis. In: International Aegean conference on electrical machines and power electronics. IEEE, pp 780–787
23. Wang D, Yu X, Wan W, Xu H (2008) A new method of infrared sensor measurement for micromouse control. In: International conference on audio, language and image processing. IEEE, pp 784–787
24. micromouseonline "mazefiles" (2020). https://github.com/micromouseonline/mazefiles/tree/master/classic. Accessed 27 July 2020
25. Brown GM, Pike L (2006) Easy parameterized verification of biphase mark and 8n1 protocols. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp 58–72
26. Zawadniak PVF (2020) 3d micromouse simulator in unity-all Japan 2018 maze. https://youtu.be/6HuG_72jt6M. Accessed 31 July 2020