CIR Project: Automatic Shopping Cart

By

Albert Compte Prades
Lukas Elias Paul Hageman
Aurelio Negri

Facultad de Informática de Barcelona
Universitat Politècnica de Catalunya

Professors:
Anaís Garrell Zulueta
Cecilio Angulo Bahon

Barcelona, January 2024

Contents

Abstract

This report contains the description and discussion of the Automatic Shopping Cart project for the course Cognitive Interaction with Robots. The application consists of three components: a smartphone-like app, a backend, and a robot cart in a simulation environment. Two application versions for the computer were developed: one for ROS and one for Unity. The former was not sufficient for testing with participants, hence we developed the latter one for that purpose. The results of the tests suggest, despite lots of room for improvement, that the robot cart made the grocery shopping experience faster and more convenient for the users.

Chapter 1

System Requirements

In consideration of the constraints around time and cost, the decision was made to pursue the development of the automatic cart robot application through simulation rather than opting for the design, manufacturing, and testing of a physical robot. This choice was made since it is logical to first test the system's usability before applying it to a physical modality. To already prepare for a potential physical application, we aimed to use *ROS* as the simulation program. Unfortunately, some struggles with the robot configuration and some limitations related to the physical constraints of the *ROS* environment hindered us from designing a testable user-friendly product for participants. Therefore, we replicated and improved our project in *Unity*.

To run the deprecated ROS simulation, the user needs a computer with the Ubuntu operating system installed, or a virtual machine fulfilling the same purpose. This virtual machine can be created with *VirtualBox* [Oracle 2023] using an *Ubuntu 20.04* image [Ubuntu 2023]. Some setting should be configured on this image before installing *ROS* such as installing another terminal and enabling root permissions on the file *etc/sudoers*. The project also requires to install *ROS Noetic* [Luqman 2022] with catkin and its TurtlebotV3 packages [Addison 2020]. Some tools such as *Gazebo* or *RViz* will be installed with *ROS*. Finally, a *ROS* package called Automatic-Cart should be created in the *catkin_ws/src* folder, and substitute the Automatic-Cart folder by our own project folder.

To run the Unity simulation, the user requires a computer with *Unity Hub* [Unity 2023b] and the version *Unity 2022.3.10f1* [Unity 2023a]. Then, they only have to download the project repository (Appendix C) and open it with *Unity Hub*. All the

configurations on the *Unity* environment will be already installed on the repository.

Lastly, to run the application, which is present in both projects, the user needs a *Python* environment installed on their computer. It could be installed in the operative system, a virtual environment or a docker. However, it should have the following packages installed: *tk 0.1.0*, *gTTS 2.5.0*, *networkx 3.1*, *numpy* and *playsound 1.2.2*. The app code for ROS is stored on *cart_app_ros.py*, while its version in Unity is stored on *cart_app_unity.py*. The first version should be run inside a ROS node, while the second one should be run directly on the Python environment. To do so, you can use any terminal which runs the previously mentioned Python environment.

Chapter 2

Description

2.1 Idea

The concept of the human-robot interaction application that we worked on is an automatic shopping cart. Such a cart should help (supermarket) customers to get a better shopping experience. ”Better” in the sense that it would be easier to find the desired products than it is currently. Currently, shopping lists are used by many people during their shopping experience. This is to perform the shopping task conveniently. Arnaud et al. (2015) showed for example that people from Generation Y (born between 1978 and 1994) use shopping lists to control their budgets, enforce discipline, and limit their purchases.

Nowadays, several large supermarket chains offer smartphone apps to customers in which they can make shopping lists from the products that are available. In Figure 2.1, you can see an example of such a smartphone app.

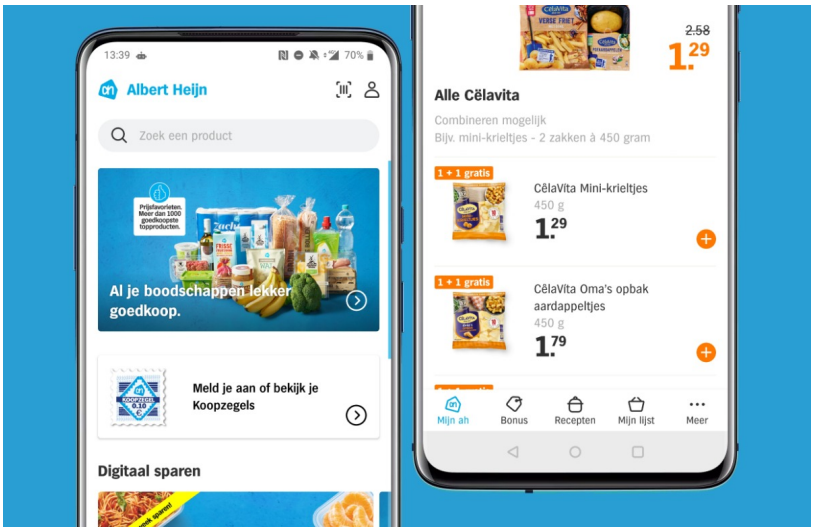


Figure 2.1: An example of a supermarket smartphone app. This one is from the Dutch supermarket Albert Heijn Klaphek (2020).

This gives the user a good sense of what they can buy and it also helps them prepare for shopping. We wanted to extend this useful idea with our project by connecting a smartphone app concept to the automatic shopping cart, reducing as well the effort of the user to move the cart. By using an algorithmic approach, a route can be computed and communicated via the app and the robot. In the next subsections, the three parts of our project and their integration are explained. In Appendix C, one can access all code that has been used for each part of this project.

2.2 App

As mentioned before, the app is based on already existing smartphone apps for supermarkets. The app is developed mainly by using the Python package Tkinter (Python Documentation). It consists of several screens that the user can switch between using a menu bar at the top. Even though the app is programmed for the computer, the app window is sized in such a way that it represents the proportions of a smartphone. This is done to give the testers the feeling it could work on a phone in a real-life application.

On the home screen (Figure 2.2), the user can choose which supermarket they want to do their groceries in by clicking a button. The next screen, Figure 2.3, contains a shopping list creator. The user can create a shopping list by starting to type a product they would like to buy. While typing, a suggestion list appears. This list

is based on the user input, first showing products starting with the typed characters and below those showing the products that contain the typed characters. By double-clicking, the user can add an item to the shopping list. Removing items is also possible by double-clicking on items in the list. When the user is satisfied, they can send the shopping list. Via the backend, explained in Subsection 2.3, the shortest route is calculated and shown on the route screen (Figure 2.4). Via this screen, the user can also control the robot cart by commanding it to pause or follow them. By clicking next, the user can let the robot know they picked an item and are ready to continue the path to the next item in the supermarket.

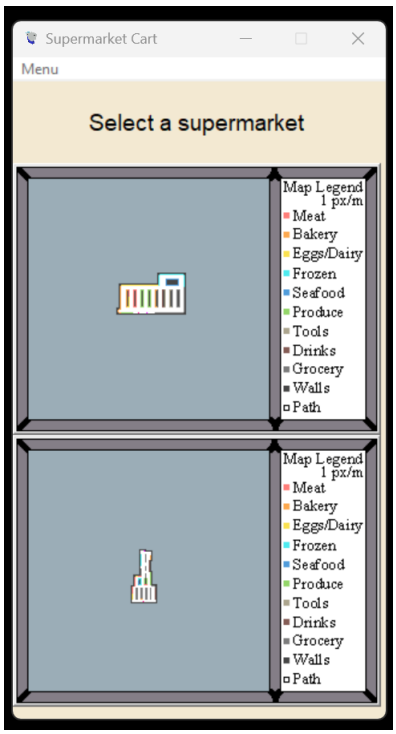


Figure 2.2: The home screen of the app on which the user can select a supermarket map. On top, you can see the menu bar which the user can use for navigation through the app.

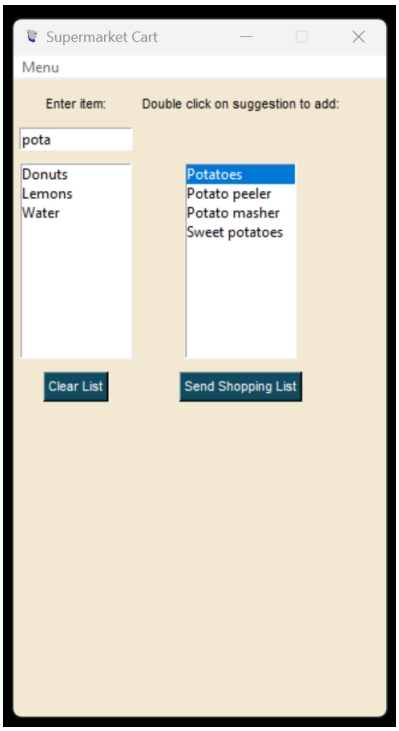


Figure 2.3: On this screen, the user can create their shopping list and send it to compute the optimal route.

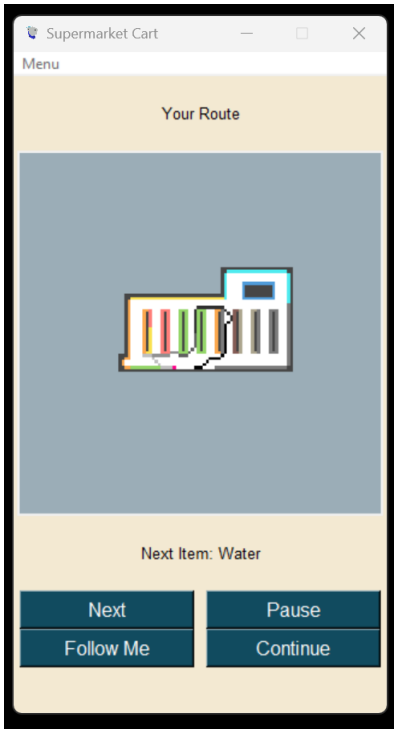


Figure 2.4: Here, the user can see the calculated route and give commands to the robot. They see the next item, and can indicate they collected it by pressing ”Next”.

2.3 Backend

The goal of the backend is to connect the app and the robot together, which we will describe in more detail in the integration section 2.5. The bulk of the code is located in the Python file *utils.py* and in *backend.py* where we implemented some functions to be used both by the app and by Unity.

2.3.1 Text-To-Speech

To add some more interaction, we decided to make the robot able to speak with Text-To-Speech (TTS). We tried two different solutions; since originally we were not aiming for the app to have internet access our first choice was to use the Python library *pyttsx3* [pyttsx3 Documentation], which relies on the TTS software from the Operating System (Windows uses SAPI5 while Linux uses eSpeak). The issue with this library was that, while the SAPI5 was producing understandable and decently natural utterances, eSpeak was hard to understand and very unnatural. Therefore, we opted to use *gTTS* which is the TTS library provided by Google and relies on a Google API [gTTS Documentation]. This means it requires internet access but at the same time is OS-agnostic, delivers better quality utterances, and gives more configuration options.

2.3.2 Graph Generation

The goal of the graph generation is to go from a list of items and the image of a supermarket floor plan to a graph represented in the Python library *networkx* [NetworkX Documentation]. To achieve this, we initially establish a scale relationship between the image and real-life dimensions. For simplicity, we have chosen a scale of one meter per pixel. At the same time each node, represented by the tuple (x,y) of the image, will map to a pixel in the image. As depicted in Figure 2.5, we employ predetermined colours for distinct item categories. This ensures that when we generate nodes for different categories, we evenly place all items of the respective category in their designated locations. In real life, this will mean that the item can be found on the shelf at those coordinates.

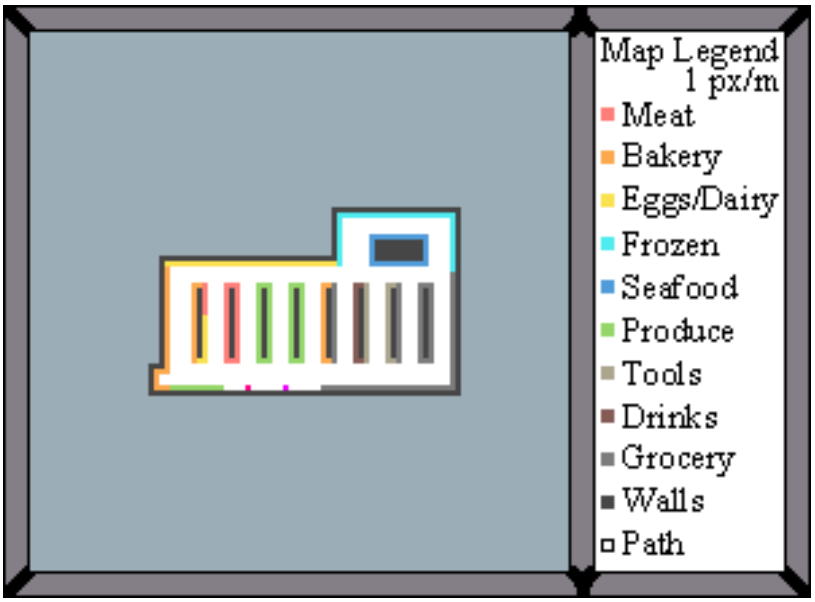


Figure 2.5: A store layout modeled after the LIDL located in Barceloneta.

Regarding the edges, we make some distinctions. The walkable nodes, which are essentially the white pixels, will have edges towards their 8-neighbourhood (also known as Moore neighbourhood), while the ones representing all the shelves will only use a 4-neighbourhood. Important to note is that edges originating from a shelf can only go towards a walkable node so you cannot actually walk inside shelves. The length of the edges is measured using simple geometry to respect the real-life distances. There are a few reasons to use these separate methods; the most important one is that it avoids cutting corners inside shelves.

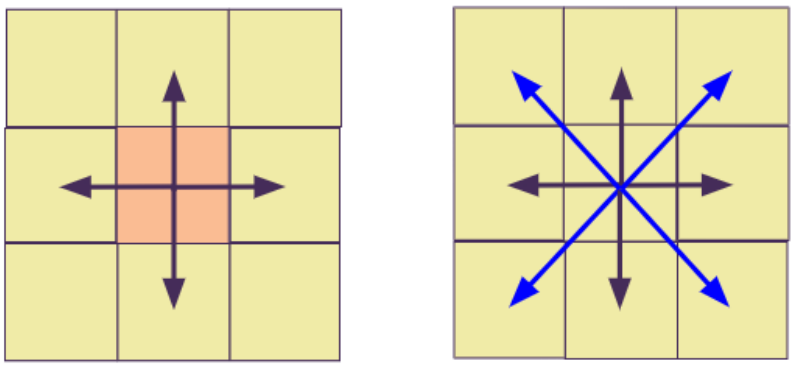


Figure 2.6: Diagram of how the edges are added. On the left for non-walkable nodes and on the right for walkable ones.

2.3.3 Path Algorithm

Now that the graph is completed we can talk about the navigation aspect, arguably the central part of this project. The goal is, when given the shopping list, to return the ordered list such that the distance walked to collect all the items in that order is minimised. This problem essentially is a modification of the Shortest Hamiltonian path over a complete graph, where the nodes are all the items in the shopping list while edge weights are the distances between the items measured using Dijkstra's algorithm. The complexity of the problem then depends on the number of items on the shopping list. For a small number of items we can just run a greedy algorithm by trying all possible paths, on modern laptops this is doable in a reasonable amount of time for an app if the items are less or equal to 11 (takes about 1.5 seconds). Since the problem grows factorially already doing it for 12 items will increase the run time to over 15 seconds. Since the Hamiltonian Path is an NP-complete problem, we can only aim for an approximation for a longer shopping list. In this case, we developed our own algorithm that works in an iterative matter. At each time step, it takes the smallest edge in the algorithm and adds it to the path, if the two conditions are met:

- The edge does not connect the start and end nodes, otherwise the final path will not contain all the items.
- The edge does not create a circle: Similar to before this would prevent us from reaching the end from the start.

With these approximations, we ran some tests and compared the length of the path found by the precise algorithm versus the approximation which gives promising results to the validity of our approximations.

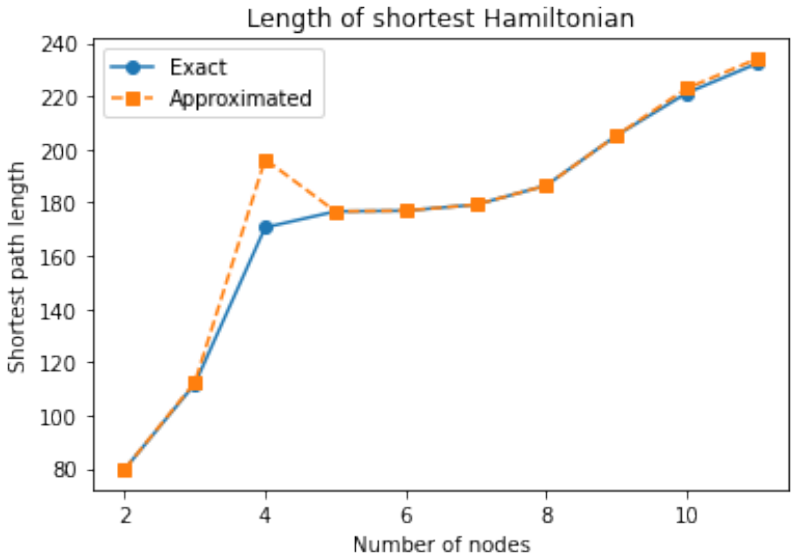


Figure 2.7: Shortest Hamiltonian Path length found by the two algorithms.

2.4 Robot

As mentioned previously, the robot is not physical. The design that was created was a 3D model made in *Blender* [Blender 2023]. To not deviate the shopping cart from the regular ones that are known to users already, the design was made such that the appearance was as similar as possible to the regular ones. Although it includes some extra components to represent some extra parts of the robot, such as cameras and Lidars. Figure 2.8 shows the robot's design.

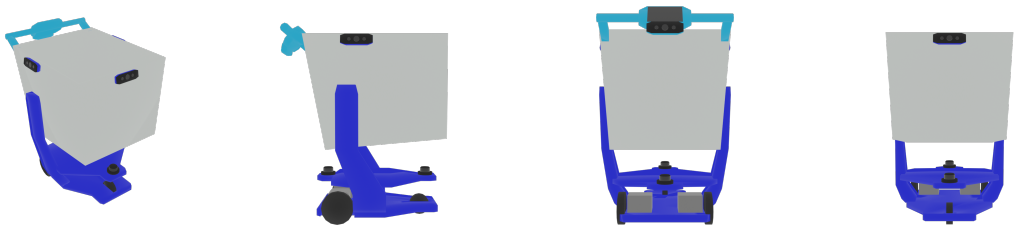


Figure 2.8: Robot design in Blender.

In the following sections, we will explain how we implemented the robot in *Unity* as it was the version used in the testing procedure. To know about the implementation in *ROS*, go to Appendix A.

2.4.1 Unity Implementation

The design of the robot in Unity is done by components, by assigning a box collision to the robot and a *Rigidbody* component which represents the mechanics of the robot. Its exact position and rotation are directly given by the environment, not requiring any odometry system to extract those values. The spatial data used by the *backend.cs* code is the same as the one received by a "tf" or "odometry" subscriber in ROS, with the difference that the origin of the coordinates is located at the origin of the environment instead of the start position of the robot. The robot also uses a simpler version of a lidar implemented with a set of raycasts. These raycasts are used to prevent possible collisions and detect the user and other customers on the map (if they are close to the robot). Figure 2.9 shows the distribution of the raycasts.

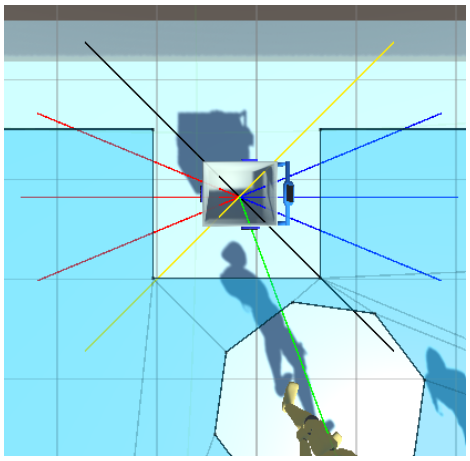


Figure 2.9: Lidar implementation. The red rays detect objects in front of the robot, while the blue rays detect objects at the back of the robot. The green ray is always directed to the user and determines if the user is close to the robot.

2.4.2 Robot Behaviour

The robot interacts with the data given by the app but also with the information given by the raycasts. In the internal code of the robot, these data modify the functioning mode of the robot and provide some extra information that the robot could use for performing any of its operations.

In general, the app can modify the robot's status in four different modes:

- The Continue mode is used when the user requests the robot to follow a path to a specific product and the robot has a path to traverse. Depending on the

situation of the robot with the user or the path, the status of the robot can transition to other statuses, such as User In Path, Destination or Wait. The User in Path status happens when the user is in the middle of the path and the robot cannot continue with its journey. In that situation, the robot will request the user to move away using TTS. The Destination status happens when the robot reaches one of the products and the user has to update the path directly from the app. Lastly, the Wait status happens when the user is too far from the robot location or the robot is almost occluded by other objects. As the robot cannot continue the journey without the user, it waits for them while it requests them to follow it, just in case the user loses the robot. If the user advances the robot in the product's path, the robot will continue its journey to catch on them.

- The Follow Me mode is activated when the user requests the robot to follow him. In that case, the robot path is directed to the user and it is updated periodically by the app. This mode has two extra statuses to stop the robot: when it is close enough to the user (Follow Me Wait, 1.66 m in real life) or making it go back if the user is approximating the robot (Follow Me Back, $<$ 1.25 m in real life).

- The Pause and Stop modes, which are used to force the robot to stay at the same location.

These modes or statuses are represented in Figure 2.10.

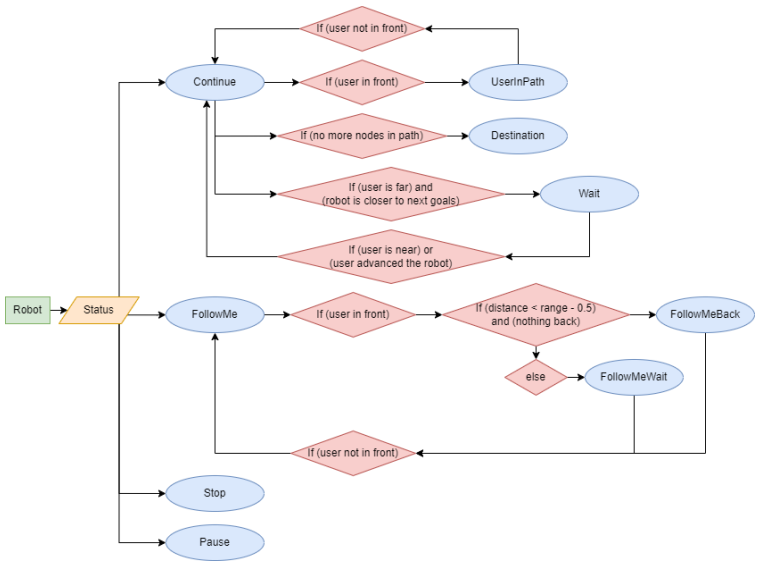


Figure 2.10: Transition between the statuses of the robot.

The robot behaviour is represented in Figure 2.11, where the robot adjusts its linear and angular velocities according to the status of the robot, its location, the path to traverse and other situations related to the lidar's information. In general, the robot computes the direction of the following node of the path, removes the nodes that are closer to a certain threshold, and adjusts the orientation of the robot according to its angle error. Certain statuses provide a bit of flexibility on the robot movement, such as the Follow Me and Destination, where the user can "push" the robot away without any explicit order in the app. These mechanisms are normally included to prevent the robot from blocking the user in certain places on the map.

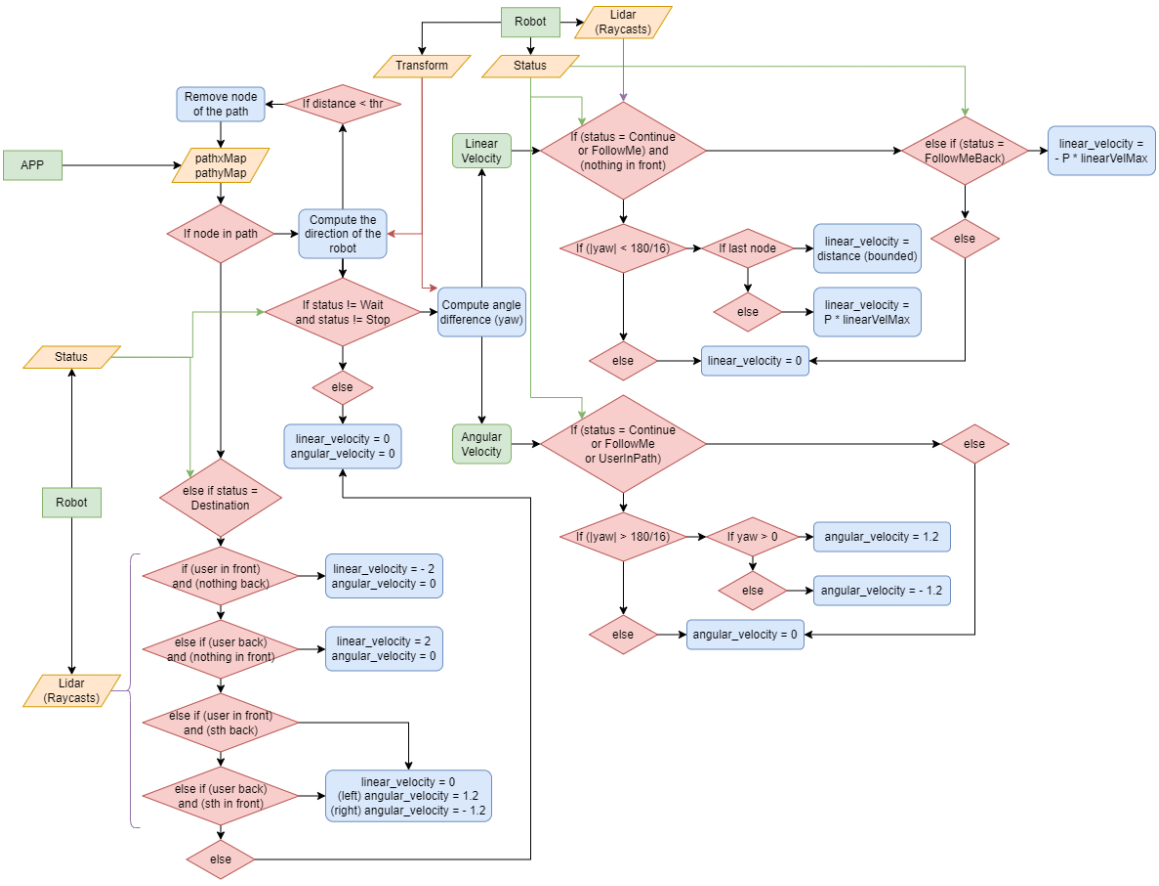


Figure 2.11: Diagram of the robot behaviour.

2.5 Integration

The integration of the three main components, the App, the Backend, and the Robot, was performed by taking the app as the central node of the project. In our design, the App has access to the functions provided by the Backend code and

interacts with the Robot via a TCP connection. Both App and Backend were coded in Python as they were originally implemented to work inside a ROS node, while the Robot and other components of the simulation are coded in C#.

To perform the TCP connection, the robot script *backend.cs* acts as the server of the connection while the app script *cart_app_unity.py* acts as the client. The connection is performed on the localhost address 127.0.0.1 and port 8888, while the data is serialised to JSON format.

The messages sent from the App to the Robot contain, but not always necessary are:

- **Mode**: Testing mode of the project (A or B). More information about this in Chapter 3.
- **Status**: Keyword to identify the type of message the user aims to send. It is different according to the button clicked on the app.
- **pathx**: Array of integers which represents the x coordinate of a path.
- **pathy**: Array of integers which represents the y coordinate of a path.
- **productsx**: Array of integers which represents the x coordinate of the products (mainly used to configure the environment).
- **productsy**: Array of integers which represents the y coordinate of the products (mainly used to configure the environment).
- **productsNames**: Array of strings which indicates the name of products (mainly used to configure the environment).
- **mapName**: Name of the map selected (mainly used to configure the environment).

While the messages sent from the Robot to the App contain:

- **Status**: Mode or status of the robot, previously discussed in Subsection 2.4.2.
- **x, y**: coordinates of the robot, indicated in the app format.
- **xUser, yUser**: coordinates of the user, indicated in the app format.

These App messages are sent after certain events of the App, while the Robot messages are sent periodically, both activating certain events on the receiver. These events can be classified according to the status specified on each message. In the case of the messages sent by the App we have:

Screen	Event	Status	Data	Response
Home	Map select.	World Init	pathx, pathy, mapName	The environment initialises the map "mapName" and locates the cart and the user on the coordinates given by "pathx" and "pathy". [Status = Stop]
Shopping	List created	Item List	pathx, pathy, productsx, productsy, productNames	The shopping list is initialised and the products are located in the map. The robot receives the path to the first product. [Status = Stop]
Route	Next	Next Item	pathx, pathy	The robot receives the path to the next product. [Status = Continue]
Route	Next (no items)	Destination	pathx, pathy	The robot receives the path to the exit. [Status = Continue]
Route	Follow Me	FollowMe	pathx, pathy	The robot receives the path to the user and starts following him. [Status = FollowMe]
Route	Pause	Pause	-	The robot pauses its journey [Status = Pause]
Route	Continue	Resume	pathx, pathy	The robot continues with the previous path [Status = Continue]

Table 2.1: Messages sent by the App. All the messages sent the mode and status of the app.

While the App responds the Robot messages on the following way:

Status	Response
Stop	No response.
Continue	No response.
UserInPath	Requests the user to move away (TTS).
Wait	Requests the user to follow the robot (TTS).
Destination	If it is the destination of a product, request the user to pick up the product (TTS). Otherwise, thank the user for attending the test (TTS).
FollowMe	If the user is far away ($>$ 10 m approx.), request the user to wait for the robot (TTS).
FollowMeBack	No response.
FollowMeWait	No response.
Pause	No response.

Status	Response

Table 2.2: Messages sent by the Robot.

In all the cases, the app updates the location of the robot and the user on its internal map in order to generate paths in real-time. All the coordinates sent correspond to the app coordinates. So, the *backend.cs* code should transform them during the communication between the robot and the app. These transformations can be summarised with the following formulas:

$$X[Map] = -((Y[App] + Y[Origin]) - image\_Width/2 + 0.5) \cdot imageScale \quad (2.1)$$

$$Y[Map] = -((-X[App] - X[Origin]) + image\_Height/2 - 0.5) \cdot imageScale \quad (2.2)$$

$$Y[App] = (-X[Map]/imageScale - 0.5 + image\_Width/2) - Y[Origin] \quad (2.3)$$

$$X[App] = (Y[Map]/imageScale - 0.5 + image\_Height/2) - X[Origin] \quad (2.4)$$

2.6 Simulation

Together with the design of the robot's behaviour, we also designed an entire environment to perform the bystander tests. There, some metadata is stored in log files in order to perform the subsequent tests analyses.

2.6.1 User

In order to interact with the shopping cart, the products, and the supermarket, we implemented a user *GameObject* in the Unity environment with a First Person View camera. The controls mimic those commonly found in video games, employing the WASD keys for multi-directional movement and the mouse for adjusting the camera perspective. When the user is close to a supermarket product, they collect the product by updating their shopping list and destroying the GameObject of the product. To model the user and their animations, we used the assets from Games (2022) and Iglesias (2023). Figure 2.12 shows what the user looks like.

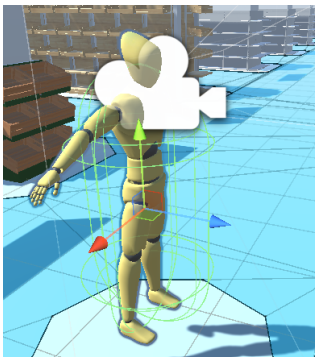


Figure 2.12: Design of the user as a "Banana man" with a First Person View Camera.

2.6.2 Supermarkets

The supermarkets were designed using both *Blender* and *Unity*. The former was used to design the walls from the supermarket layouts, and the latter was used to include the furniture and electrical appliances of the supermarket, as part of the models can be obtained from the Unity Asset Store. In general, the supermarkets are 3 meters tall. Nonetheless, in order to prevent collisions of the robot with some corners of the supermarket, we decided to re-scale the map and the characters to 1.2. The distribution of furniture and appliances was designed according to the sections defined by the layout, while some signs were distributed throughout the supermarket to indicate each of its sections. Some of the assets applied in these sections were from Enozone (2023), katarina842 (2023), and CharlotteUA (2023). Other models were directly designed in Blender. Figure 2.13 shows what our design of the LIDL store looks like.

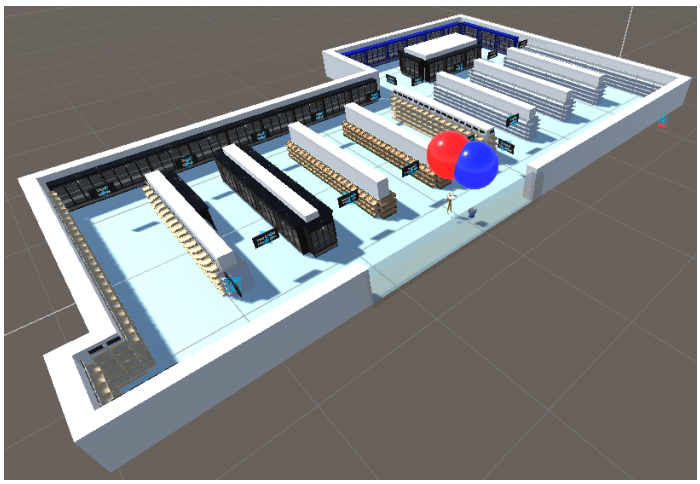


Figure 2.13: Design of the LIDL store taking Figure 2.5 as a reference.

2.6.3 User Interface

The user interface provides some extra information to the user in order to prevent them to not getting lost in the Unity environment (Figure 2.14). On the upper-left corner of the interface, we included a shopping list with the number of products collected. If the user collects a product, the number of products collected increases and the product collected is followed by an "OK" text. The lower-right corner contains a mini-map which indicates how the user is oriented and what the supermarket looks like. The red dot in the mini-map represents the user while the blue dot represents the shopping cart. Initially, we added a purple path on the mini-map to indicate where the shopping cart is going. However, after some tests with bystanders, we realised that one user only focused on the mini-map, ignoring the robot, so we removed it.

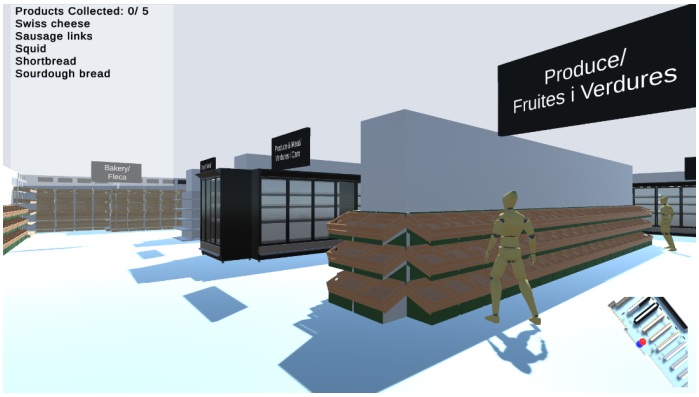


Figure 2.14: User interface in game. On the background there are some customers looking at the shelves.

The products are designed as purple emitting spheres which appear at the places of the selected products. Its location is adjusted to the shelves in their surroundings, and these are destroyed by the interaction of the user. Figure 2.15 shows how the user interacts with the products.

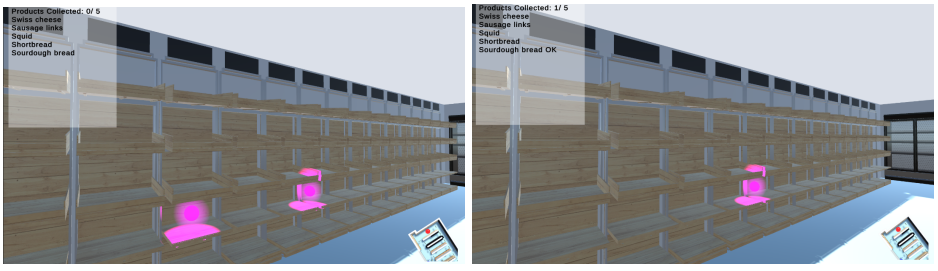


Figure 2.15: Interaction of the user with the products. The purple spheres represent the products the user has to collect, which can be collected by proximity.

2.6.4 Customers

The last component of the supermarket is the set of customers, who walk through the entire map to specific shelves of the supermarket. They are designed with two different C# scripts: *CustomerBehaviour.cs* and *CustomerGenerator.cs*. The first one defines the behaviour of an individual customer, by means of selecting around 1 to 10 shelves and setting the destination of the path algorithm according to the proximity of each shelve. Different from the automatic cart behaviour, the customers use a package from Unity called *NavMesh*, which computes the path of the supermarket from a baked navigation area by means of using the A* algorithm. This navigation mesh can be observed in Figure 2.16. The second script generates the customers periodically at the entrance of each supermarket. When a customer ends their journey, its *GameObject* is destroyed, preventing the generation of an infinite number of customers.

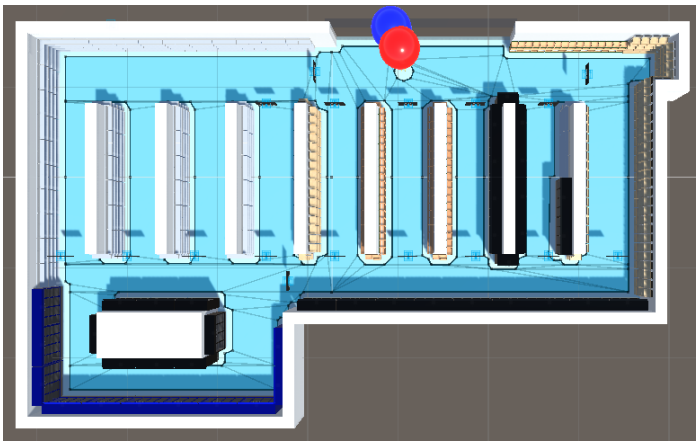


Figure 2.16: Navigation Mesh of the customers.

2.6.5 Simulator Parameters

Object	Parameter	Value	Description
Map	Image Size	300x220	Size of the layout image.
	Map Origin	10x10	Origin of the planner.
	Image Scale	1.2	Scale factor of the map in Unity.
Robot	Max. Velocity	2 m/s	Maximum linear velocity of the robot (6 km/h $\star$).
	Range	2 m	Range of the lidar (1.66 m $\star$).
	User Range	8 m	Distance from the user to stop the robot (6.66 m $\star$).
	Dist. Threshold	0.1 m	Dist. to remove the next node of the path (0.083 m $\star$).
	Dist. Threshold Last	1.4 m	Dist. to remove the last node of the path (1.16 m $\star$).
	Update Rate	1 s	Period to save data to the log files.

Object	Parameter	Value	Description
User	User Speed	1.6 m/s	Linear velocity of the user (4.8 km/h $\star$).
Customer	Shelves	1-10	Number of shelves to visit.
	Range	2 m	Range to remove a shelf from the path (1.66 m $\star$)
	Time	3 s	Waiting time to each shelve.
	Period	2-15 s	Generation period of customers.

Table 2.3: Simulator Parameters. $\star$: removing the scale factor.

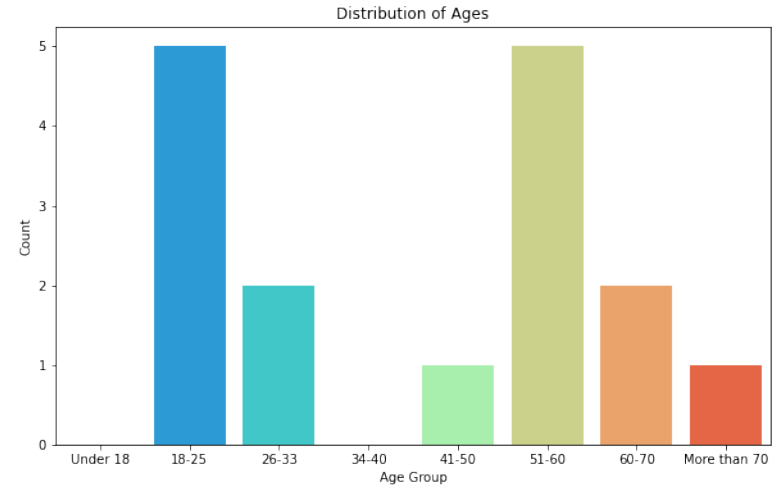
Chapter 3

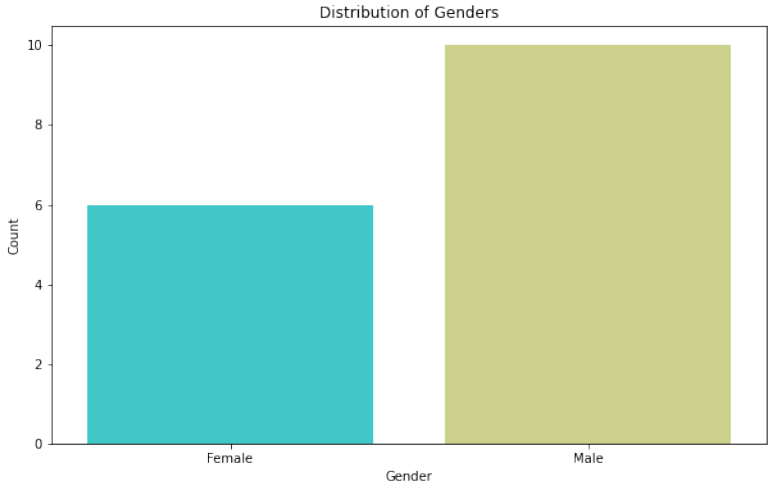
Tests

In Appendix B, one can read all the specific details regarding our testing procedures. The steps we followed during each trial are listed and the questionnaires' links are available there. In this section, the test methods and results are explained.

3.1 Participants

For selecting participants, we used a Convenience Sampling approach. We conducted in total 16 different experiments with our relatives, in 3 different countries, namely Spain, Switzerland and the Netherlands. We tried to have a wide range of ages and a balanced gender distribution but it turned out that the participants were mostly in our age group or our parents' age, and the majority of them were male.





3.2 Method

All participants followed the same procedure described in B and we will describe this procedure from the perspective of the user. For convenience, we divide each test in 4 phases:

- **Introduction Questionnaire** [Introduction Questionnaire ENG], [Introduction Questionnaire CAT]: This questionnaire was designed to collect the anagraphic data of the respondent and their shopping habits. Some questions we were interested in were how the participants usually conducted their shopping, how much they enjoyed it, and how familiar they were with robots

and technology in general. Here are two examples of questions and how the participants responded to them.

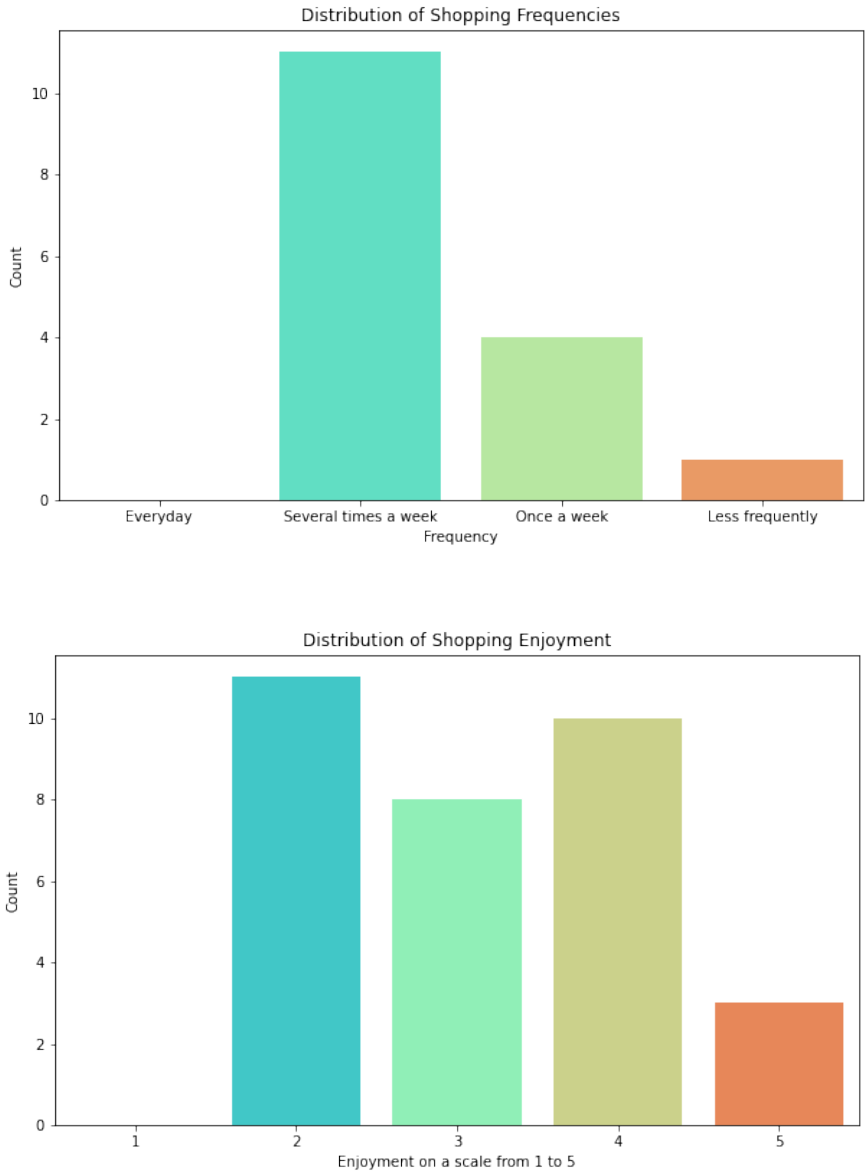


Figure 3.1: The first plot shows the distribution of how often the participants went shopping, as we can see most of them go several times a week. The second plot shows how much the participants enjoy shopping.

- **Simulation 1** and **Simulation 2**: After the questionnaires we introduced the participants to the simulations which they did in succession. In each of the simulations, they first interacted with the supermarket application where

they created their own shopping list. The only restriction we put on them was that they could select at most 5-6 items (due to time limit reasons). Once that was done, the shopping list was sent to Unity. During the first iteration, the participants had the possibility to familiarise themselves with the controls. Afterwards, their job was to simply move the character and collect the items around the supermarket. This is done to simulate a normal supermarket experience in the best way possible. During the second simulation, the users had access to the route page of the application as well, from where they could interact with the robot in the ways described previously.

Here the **Test Mode** parameter comes into play: The first one, called Mode A, will set up the robot in the standard way. Therefore, the path it will follow will be the correct one and the users just have to follow it while ideally only interacting with the application once they have collected each item. The other mode, called Mode B, on the other hand, adds some unpredictability to the robot, which will sometimes lead the shoppers to the wrong places. We made sure that half of the tests we conducted were done with the first mode while the rest with the second one. We will discuss the purpose of this feature in the next section.

- **After Questionnaire** [After Questionnaire ENG], [After Questionnaire CAT]: After the two simulations, the users had to complete a second questionnaire. Here we were interested in gathering feedback from the users, to understand better how the experience was from their point of view and to collect some ideas for improvements or places where they struggled. Here we are specifically interested in understanding how convenient the robot felt and how much faster/slower the experience was between the two simulations. On the second page of the questionnaire, we also asked the standard Godspeed questions about the shopping cart robot regarding likeability, intelligence and safety. We will explore this data in Section 3.3.

A couple of notes regarding the testing paradigm; we avoided helping the participants as much as possible, but for some of them, especially the older ones, we had to step in to help them with the controls and the application interface. Yet, since we kept our influence to a minimum, we believe that the data collected should still be significant and valid. Another possible concern about the validity of the tests is that, as one might note, we always carried the first simulation without the help of

the robot, while in the second one, the robot was always present in one way or another. Therefore, one might assume that the users were faster in the second as they were already familiar with the supermarket and the commands. Although we noticed this issue too late, we think it will not have a big impact on the results. This is because in all the occasions, the users were given the possibility to familiarise themselves with the commands and environment first, and second the shopping list was different between the two runs meaning that the participants had to go to different locations of the supermarket.

3.3 Results

3.3.1 Data Preparation

Before being able to run the statistical tests, we first needed to parse the data that we had in the Log files. This proved to be a difficult challenge. Firstly, because of the amount of data; since we recorded the state of the simulation 50 times a second and at each of those timestamps we recorded data from 5 different sources, a run of 5 minutes can end up generating a Log file of 75.000 lines, and in general some runs where over 100.000 lines or longer. Another challenge for the data preparation was the way that the Logs were stored. This required various handcrafted regular expressions and type conversions in order to have the data stored in 5 different *pandas* [Pandas Documentation] dataframes. The code used to parse the data and to generate all the plots can be found in *test_parse.ipynb* under the LogFiles folder.

3.3.2 Hypotheses

There are two main hypotheses we want to test:
Is the grocery shopping faster when using the robot?:

- Null Hypothesis: No speedups observed.
- Alternative Hypothesis: The application provides a speedup.

We will check this hypothesis from two different perspectives. The first one is from the simulation data that we collected, which will tell us objectively how much the speedup was. The second perspective comes from the perceived speedups from the users and will be extracted from the questionnaire data. For the first perspective, we extracted the duration in seconds between the moment the user moved

outside of a bounding circle from the starting point (i.e. after they finished familiarising themselves with the commands) and the moment when they collected the last item. This value in itself is not enough though. This is because during each run we have a different shopping list. Therefore, to correct the duration, we scale it by the length of the best path generated that one can take since we believe that is a good measure of the difficulty of each shopping list. Now, to measure the speedup, we divide this adjusted time measure of the first run with the one of the second run. This should give us an idea of the speedup between them.

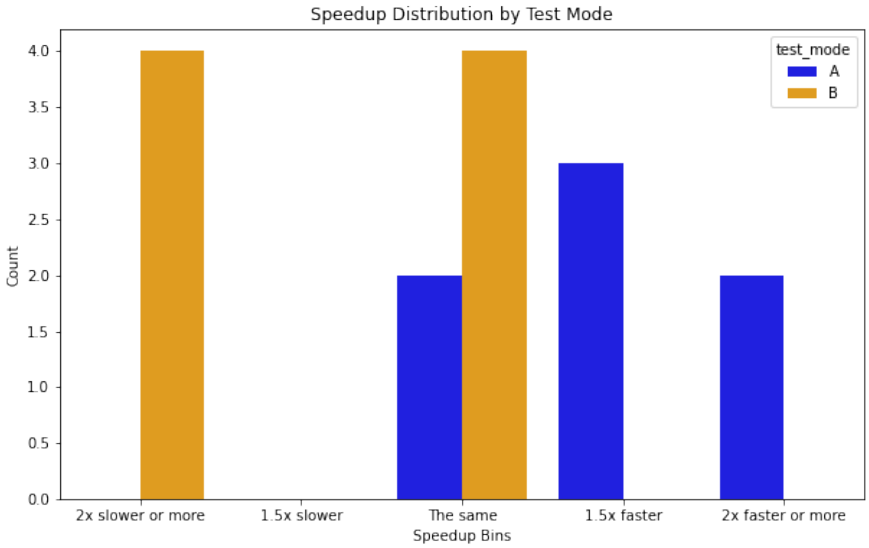


Figure 3.2: Measured speedup for both testing modes

As predicted, most users were on average faster when the test was set to mode A and slower on mode B. But what we are interested in is quantifying the impact of our application when it is working correctly, so we will focus our attention on the tests conducted in test_mode A. For this, we need to use an appropriate statistical test in the paired difference tests family. Unfortunately, we cannot use the paired Student's *t*-test as we do not know for certain if the speedups are Gaussian distributed or follow a distribution that depends on some of our participant pool characteristics. Therefore, we turn our attention to the Wilcoxon signed-rank test, where the first population will be the adjusted time measure for the first simulation and the second one for the second simulation. We can use this test since the data is continuous and we can rank the difference between the two populations.

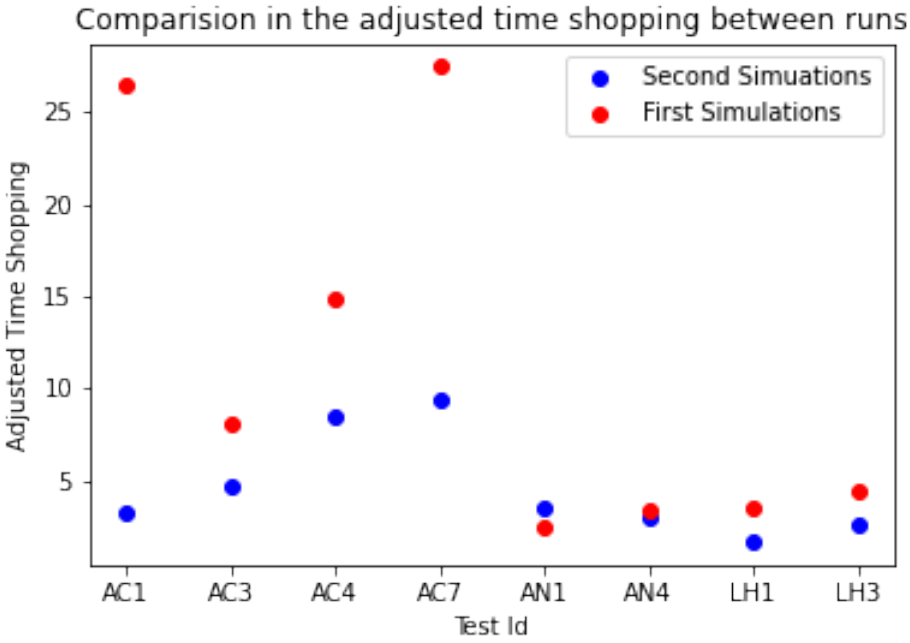


Figure 3.3: Larger time means the participant took more time to complete the shopping

Our null hypothesis posits that the median of the differences ($First - Second$) between the two populations is centered around 0. On the contrary, the alternative hypothesis suggests that the median is greater than 0, indicating that the times for the second simulation are lower. Indeed when running the Wilcoxon signed-rank test with a one-sided alternative hypothesis on our data we obtained a p-value of **0.0117** which with a significance value $\alpha = 0.05$ means we can reject the null hypothesis. We now apply the same reasoning with the data gathered from our questionnaires about the perceived speedup for each user.

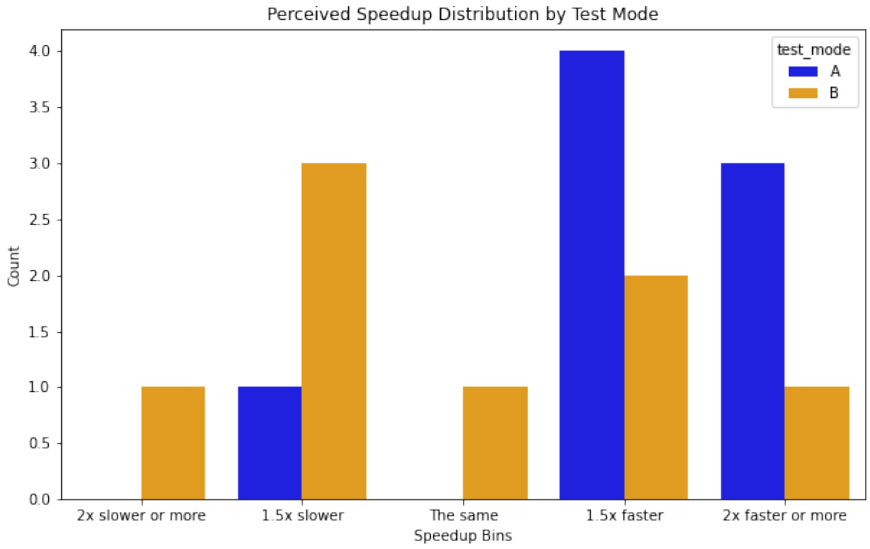


Figure 3.4: Perceived speedup distribution of the users for each test mode.

The first population will be set to 1 as the baseline speed for the first run and the second one will be set to the values reported by the participants. For this test we observe a p-value of **0.0078** which again is below our $\alpha = 0.5$ and therefore also here we can reject the null hypothesis and claim that also there is a perceived speedup.

Is the user experience Convenient?:

- Null Hypothesis: The user experience is "not convenient".
- Alternative Hypothesis: The user experience is "convenient"

Also for this question we will have two different approaches, one more subjective and the other one more objective. For the objective one we will compare the number of interactions the users had with the robot in both test modes, in theory if the interactions with the robot are less during test mode A, then it will mean the user experience is convenient. In this situation we cannot use a paired difference test since the data comes from a set of different users and therefore the variance is not the same.

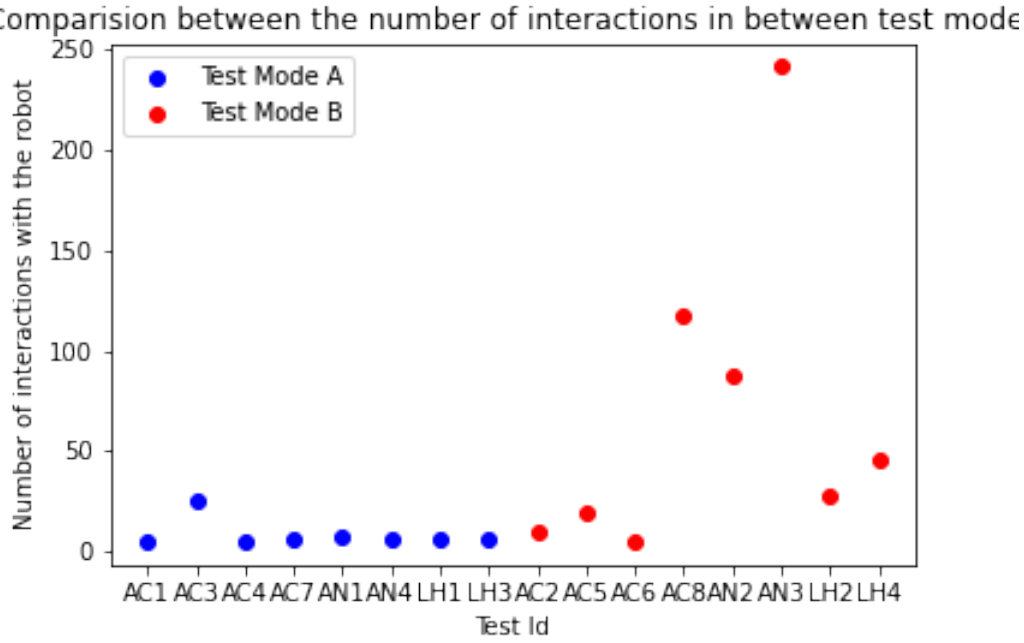


Figure 3.5: Number of interactions with the robot.

In this situation therefore we have to use the Mann–Whitney U test, the null hypothesis in this test is that the probability of any sample X_i from the first population being larger than another sample Y_i from another population is equal to the inverse, we can use this test since the samples between the two populations are independent. Without going into too much detail into the theory of this test, when inputting the data and using again a one-sided alternative hypothesis we obtain a p-value of **0.0084** which allows us to reject the null hypothesis. The same test is then applied to the user feedback we received from the questionnaire, where we ask what was the overall experience with the simulation on a scale from one to five.

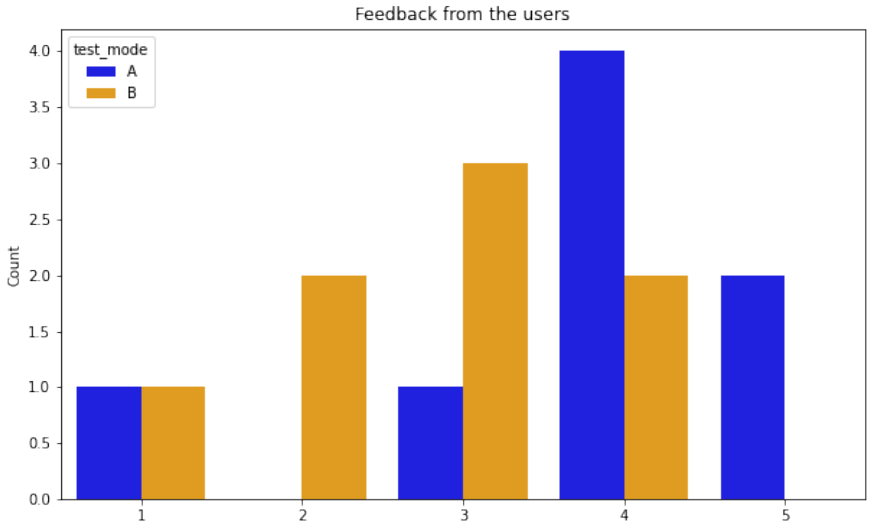


Figure 3.6: Convenience feedboack from the users.

With this data the test returns a p-value of **0.0361** which again allows us to discard the null hypothesis.

3.3.3 Other interesting insights

We will explore a couple of additional insight from the data that are not related to the hypotheses per se. The first one, not surprisingly is the relationship between the age of the participants and the adjusted time which it took them to navigate the supermarket. As we can see from the plot in fact the two values are correlated to each other.

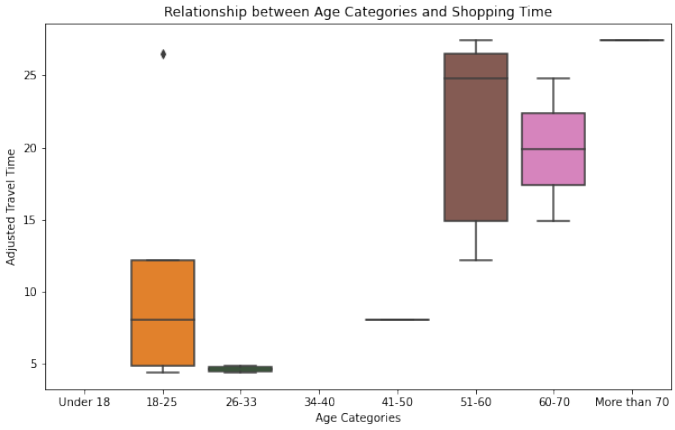


Figure 3.7: Relationship between the advanced travel time and the age of the participants.

Another interesting data to analyze is what were the most popular items. Since the users had free choice over them it gives an insight so supermarkets over which areas they might focus their attention more. Perhaps unsurprisingly these were the most popular items out of our set of 500:

Item	**Count**
White bread	8
Eggs	8
Milk	7
Potatoes	6
Tomatoes	5
Yogurt	5
Chicken breasts	4
Coffee	4
Apples	4

Table 3.1: Most Common Items and Counts

Chapter 4

Conclusions

4.1 Limitations and Future Work

The largest limitation to our work is that we did not have a physical robot at disposal that we could bring outside of school grounds. Therefore, we were limited to only simulations which in general is not ideal. Naturally, the next step would therefore be to implement a real world version and see if supermarket chains would be interested in it. This real world application would probably also include a solution for mapping a supermarket in a more precise way, where the floor plan is more detailed, the items are placed in the correct locations, and the app itself would have the full catalogue of products. This is because currently in our simulations the items where placed randomly and we only included a small subsets of supermarket items with no additional information like pricing. There is a reasonable case for including an intermediate step that involves using only a mobile application. We thought about this option during the project, where users would input their shopping list, and then on the route page, there would be a simple navigation screen for users to follow. This approach is simpler, less costly to implement, yet still likely to enhance the shopping experience by making it faster and more convenient. Another issue we faced was to translate the floor plan from an image format to a 3D format for ROS and Unity, such that the coordinates in 2D were matching the ones in three dimensions. This proved to be challenging since the pixel level was too coarse to build on top of. Therefore the scale was a bit changed and we implemented some functions that translate the coordinates between the different domains.
During our tests with the participants, we also encountered the problem that the older generation never experienced computer video games or anything of the sort,

and therefore for them the interaction proved quite hard and most likely confusing. Maybe an UI more similar to the one of a car could have proved more intuitive and familiar for them.

4.2 Suggestions for improvement

Since our application is in its infancy, there are quite some things that can improve the user experience. Think of making the simulation environment more similar to a real supermarket by adding visible product objects, or letting the user enter a supermarket building and find the robot cart. Also, our participants came up with some great suggestions. One mentioned for example to add the possibility to pay directly via the robot cart. This would mean a user does not have to put the products on a conveyor belt and can put them directly in their bags. Another participant suggested adding an item recommender inside the app to make the application more intelligent. We were working on this, but due to time constraints, we did not implement this. However, it is good to see that it is also of importance for potential end users. As a last suggestion, one let us know that it would have been convenient if the speed of the cart could be adjusted by the user. This of course makes sense, as people differ in the speed at which they want to do their shopping. Obviously, the speed cannot go too high as that would result in accidents. Therefore, around three different speeds could be a great solution. A robot with an adaptive velocity is also a viable possibility.

All in all, one can see that much can still be improved and the project would need quite some evaluation and design iterations to approach the perfect user experience. Also, one can imagine that if this project were translated to a real-life scenario with a physical robot, new issues would emerge. However, this simulation project can function as a good base for a physical automatic shopping cart.

Appendix A

ROS Implementation

Initially, the project was designed to simulate the robot in ROS, exporting the Blender models to .dae files and communicating the robot and the application with publisher and subscriber topics. However, after several attempts trying to fix several issues encountered during the project, we decided to change the robot environment to Unity in order to avoid some of these issues and speed up the environment design. In the following sections, we will describe the work done in ROS and the reason why we changed to Unity.

A.1 URDF Design

The design of the ROS robot was built in a URDF file, using a pair of .dae files for the base and basket of the robot. The robot wheels were included separately using the cylinder models from URDF. In a first approach, we designed a robot with 3 wheels. However, after seeing the robot shaking in the simulation, we decided to apply a 4-wheel design. A prismatic (not-used) joint was added for the basket and a set of continuous joints were included on the wheels. Some other plugins, such as gazeo_ros_control, imu and ray were included for the robot control, IMU and Lidar. Figure A.1 shows the robot design in RViz.

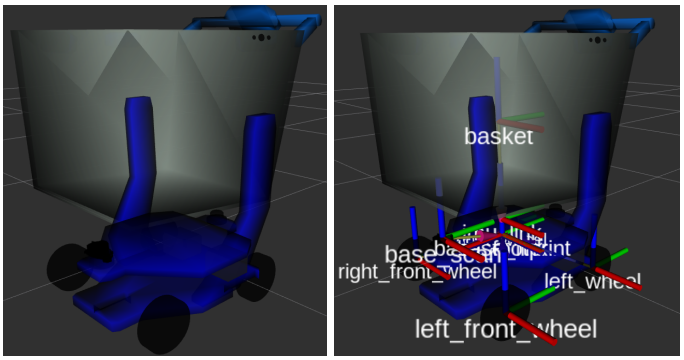


Figure A.1: Robot design in RViz. Accessible with the following command: *roslaunch automatic_cart display.launch model:='$(find automatic_cart)/models/shopping_cart/automatic_cart.urdf.xacro'*

In the robot's launch file, some other configuration files were launched to control the robot such as:

- *diffdrive.yaml:* it describes the configuration of the robot as a differential drive. The linear and angular velocity of the robot as well as the wheel distribution are described there. Therefore, it has a huge impact in the odometry values.
- *joints.yaml:* initialises the joint state controller.
- *control_params.yaml:* adjust the "inertia" of the robot behaviour and improves its physics on the simulation. There, some PID (Proportional - Integral - Differential) parameters are defined.

Several launch files initialise these configuration files since the robot could be initialised differently depending on the map.

A.2 Telemetry

To add telemetry on the robot, we created another launch file that calls the package *rqt_robot_steering* on a new ROS node. This package modifies the topic */cart_diff_drive_controller/cmd_vel* , which defines the linear and angular velocity of the robot. The robot telemetry is controlled in a graphical interface, as shown in Figure A.2. This example can be loaded with the commands: *roslaunch automatic_cart cart_house.launch* and *roslaunch automatic_cart cart_telemetry.launch*.

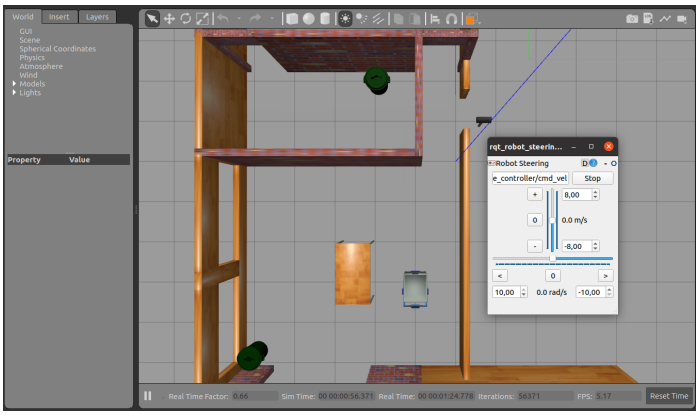


Figure A.2: Example of robot's telemetry.

A.3 Odometry methods and SLAM

The odometry of the robot is one of the most important components of our ROS project and serves to indicate which is the real location of the robot. Usually, the odometry is given by a topic published by the difference drive controller, called */cart_diff_drive_controller/odom*. However, there are other methods to get the odometry. In our project, we tested three ways to get the odometry, with an already integrated robot to the application. However, most of them offered us deviated values, resulting in a robot that collides to the walls of the supermarket. These methods were:

- **Difference drive controller:** the odometry method commented before. It computes the robot's orientation and displacement from its wheels rotation. As it depends on the configuration given in the *diffdrive.yaml*, it requires certain parameter adjustment, such as controlling the wheel radius and the wheel separation. In our system, it gaves correct displacement values when the robot moves linearly. However, when the robot rotates, the difference drive controller gets lost. We performed several tests adjusting the wheel separation. Nonetheless, the drifting observed was not completely proportional to the wheel separation, so the robot loses its orientation after several rotations. This controller also controls the velocity of the robot by subscribing itself to the topic */cart_diff_drive_controller/cmd_vel*.

- **Inertial Measurement Unit**: a device added on the robot that uses accelerometers and gyroscopes to compute the angular and linear velocity of the robot. It also publishes a ROS topic with the position and orientation of the robot,

called */cart_imu*. However, due to the behaviour of the IMU's gyroscope, the IMU also provides certain degree of drifting. Technically, the previous method also includes the IMU of the robot, but we did not notice any improvement.

- **SLAM/Gmapping**: the most promising method, it uses the odometry given by the previous methods and corrects it by adding the data given by the Lidar. To do so, it generates a map of the supermarket and adjusts the location and rotation of the robot according to the data receives by the Lidar and its expected location on that map.

This last method requires much more modification since the data is not directly accessible by the *rostopic* command. First, we should include SLAM on the project to create the map of the robot. In our project, we included the same method used by the TurtleBotV3 robot, generating a new launch file called *cart_slam.launch*. This launch file also shows the map generated with RViz, as observed in Figure A.3.

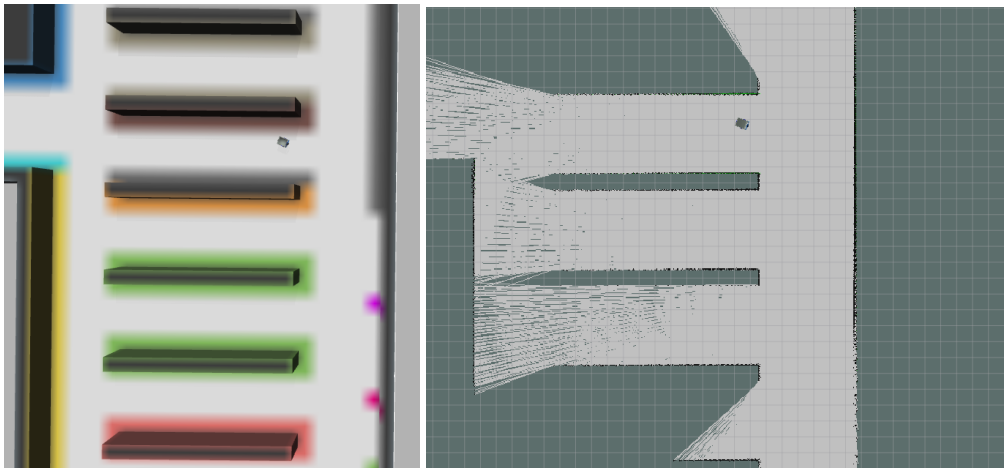


Figure A.3: Example of the SLAM map in the LIDL store (without appliances).

Apart from that, we must ensure a correct connection between the base_footprint frame and the map frame on the ROS system in order to provide the correct transformation between the predicted position and orientation from the driff drive controller and the one predicted in the generated map. With the *view_frames.py* function, we can observe the hierarchy between the frames (Figure A.4).

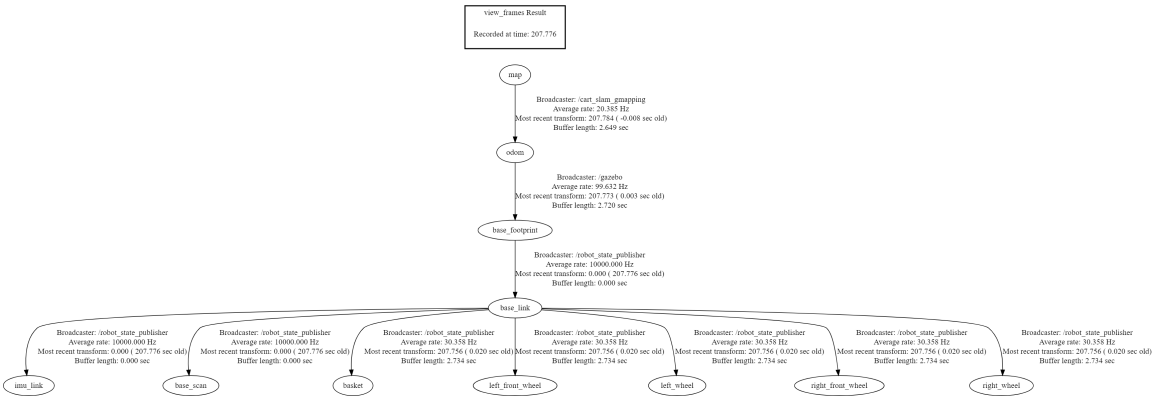


Figure A.4: ROS transform tree of the shopping cart robot.

Having that, we can obtain the odometry from the *tf2_ros.TransformListener*. In our test, we used a simpler python version of the robot behaviour shown in Figure2.11. However, although we observed some improvement on the values given by the odometry, the robot was not working at all yet. So, after seeing other possible issues (which will be commented on section A.5), we decided to re-design the environment with Unity.

A.4 Robot-Application Integration

The integration of the robot with the application was similar to the one shown in Section 2.5, but using ROS nodes and topics. Each component have their respective launch and Python files, being *cart_backend.launch* and *backend.py* for the robot, and *cart_app.launch* and *cart_app_ros.py* for the application. In this mechanism, we created two different topics: one for the communication from the app to the robot, called */cart_app2backend*, and the other from the robot to the app, called */cart_backend2app*.

Both topics also work with specialized messages called *App.msg* and *Backend.msg*. The *App.msg* contains the status requested and the (x,y) coordinates of the path generated by the application. The *Backend.msg*, on the other hand, contains the status of the robot and its x and y coordinates.

The whole project is initialised with the following launch files:

Launch File	Description
cart_shop.launch cart_shop3.launch	Model of the supermarket as the world of the simulation.
cart_robot.launch cart_robot3.launch	Model of the robot initialised at the start position of the supermarket.
cart_slam.launch	SLAM map with the gmapping mechanism to determine the location of the robot.
cart_backend.launch	Behaviour of the robot.
cart_app.launch	Application to give orders and send the path to the robot.

Table A.1: Launch files required to launch the ROS project.

A.5 Issues and Reasons why we switched to Unity

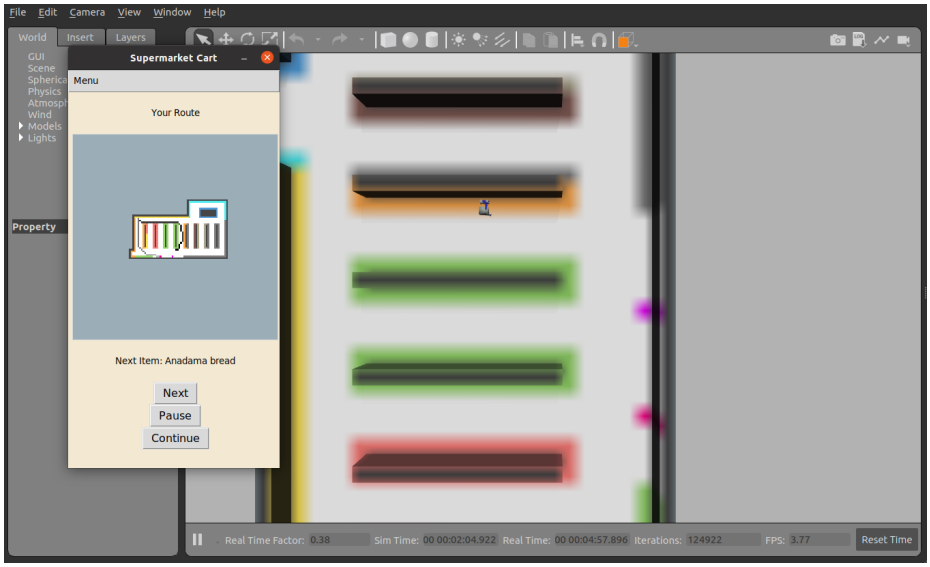


Figure A.5: Environment of ROS with a cart robot failing its mission.

In Section A.3, we highlighted one of the reasons for transitioning to Unity, which was the occurrence of drifting and errors within the odometry measurements. However, it is important to note that this was not the sole factor motivating our decision to make this transition. Some other components also demonstrated to have compatibility issues, or lacked intuitiveness for our potential users. These include:

- **Odometry errors:** Some location and orientation variables from odometry demonstrated to have biases after several adjustments. In certain cases, the

robot deviates so much that no path is recognizable from its behaviour, being difficult to determine if this issue is produced by other components of the robot.

- **Some models are not properly rendered in Gazebo:** After creating a whole supermarket in Blender, we realized that Gazebo cannot launch certain models of the map and most of the shelves added were directly omitted. That would force us to re-design the entire supermarket in a .world file by adding 1-by-1 all the appliances.

- **User design:** After looking for several ways to design the user, we did not find any intuitive to control a First-Person View user in the simulator. The most intuitive way was to create a new robot, add a camera plugin and link this camera to a RViz monitor, and then, control the robot by telemetry. In this system, the bystander could lose the control of the user robot by not stopping the robot at the correct time, making it collide to a wall and dropping it on the floor. The First-Person View camera will serve as a distraction instead of a tool for looking for products.

- **Memory-consumption of Gazebo:** During our Gazebo testing, we observed the program's lack of smooth performance, attributed to its consumption of the entire virtual machine's RAM. Despite attempts to increase memory capacity, the program persisted in running sluggishly.

- **Hard-to-Debug:** Due to ROS depends on several configuration files, it is hard to determine the origin of some of the errors encountered in the project, or what is possible to perform on each file.

- **Uncertainty:** Our limited expertise with ROS and its lack of intuitiveness make it challenging to evaluate the ease of implementing certain components on the robot or rectifying specific errors. So it is possible that if we had not switched to Unity, we would still be fixing errors.

Appendix B

Testing Documentation

Steps for executing a test:

1. Select the test mode, we have 2 test modes: A (everything works fine), and B(the robot could go to wrong places). Go to the cart_app_unity.py code and in its initialization change the parameter test_mode before each test.

2. Call the tester and request him to perform the prephase questionnaire... Here you have to introduce the test mode.

3. Open Unity, distribute the Unity Window in order to let space for the App window, select Full HD (1920x1080) aspect ratio (you can scale it later), do double click to the "Game" window to make it bigger, and press to the play button.

4. Request the user to test the controls: mouse movement, keys AWSD; in order that the user gets used to the environment controls.

5. Execute the APP and select the first map (the second map is not tested enough to determine if the robot works fine there, so you will have problems with the second map).

6. Go to Shopping list and request the tester to select between 5 and 6 products.

7. Request the user to click the Unity window (still played), and request him to look for the 5-6 products... Comment him that the products are purple spheres in the map that he could collect by gettting closer. Also comment him that the product list are checked on the Unity app.

8. After the tester perform the first phase, close Unity and the app.

9. Open Unity (first) and then the App, select the first map and request him 5-6 products again.
10. Go to Route and request the tester to click Next or Continue to run the Cart. Comment him that the image shows the path.
11. Request him to click Unity screen and follow the robot.
12. Let him collect the products with the robot until he ends.
13. Close Unity and the App.
14. Perform the second questionnaire.
15. Go to LogFiles, create a test ANX folder, and move the log files of your test to that folder. You should have 2 per run: Log (Unity operations) and LogApp (App communication)... If you perform 2 runs, you will have 4 documents. I normally store only the tester files and move all of them after each test.
16. Repeat, try to have the test modes balanced

Questionnaires:

- Pre-Test questionnaire link
- Post-Test questionnaire link
- Pre-Test questionnaire in Catalan link
- Post-Test questionnaire in Catalan link

Appendix C

Code

- Unity version https://github.com/Limax-cs/Automatic-Cart-Unity
- ROS version: https://github.com/Aure20/Automatic-Cart

Bibliography

A. Addison. How to launch the turtlebot3 simulation with ros, 2020. https://automaticaddison.com/how-to-launch-the-turtlebot3-simulation-with-ros/.

After Questionnaire (CAT). After questionnaire (cat). URL https://forms.gle/Jy8x2yDHQeuu6HPT9.

After Questionnaire (ENG). After questionnaire (eng). URL https://forms.gle/PmihXyZbbpamS5yP8.

A. Arnaud, A. Kollmann, and A. Berndt. Generation: The development and use of shopping lists. *Advances in Social Sciences Journal*, 2015.

Blender. Download blender, 2023. https://www.blender.org/download/.

CharlotteUA. Vegetable free 3d model, 2023. https://www.cgtrader.com/free-3d-models/food/vegetable/vegetable-ddeef747-828b-42af-98c6-72846514e2b3.

Enozone. Fresh shelving, 2023. https://assetstore.unity.com/packages/3d/props/furniture/fresh-shelving-267101.

B. Y. Games. Banana man, 2022. https://assetstore.unity.com/packages/3d/characters/humanoids/banana-man-196830.

gTTS Documentation. *gTTS Documentation*. https://gtts.readthedocs.io/en/latest/.

K. Iglesias. Basic motion free, 2023. https://assetstore.unity.com/packages/3d/animations/basic-motions-free-154271.

Introduction Questionnaire (CAT). Introduction questionnaire (cat). URL https://forms.gle/yMRs4Wtr6VafBkT7A.

Introduction Questionnaire (ENG). Introduction questionnaire (eng). URL https://forms.gle/xWBw7MaakLWgBvdW8.

katarina842. Industrial shelves free 3d model, 2023. https://www.cgtrader.com/free-3d-models/industrial/other/industrial-shelves-77ce91d6-fc8a-44e1-9858-0747b4c671b9.

M. Klaphek. Vergelijking: 6 supermarkt-apps langs de meetlat, 2020. https://www.androidplanet.nl/apps/supermarkt-apps-vergelijking/[12-2023)].

M. Luqman. Installing and configuring your ros environment, 2022. https://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment.

NetworkX Documentation. *NetworkX Documentation*. URL https://networkx.org/.

Oracle. Download virtual box, 2023. https://www.virtualbox.org/wiki/Downloads.

Pandas Documentation. *Pandas Documentation*. URL https://pandas.pydata.org/.

Python Documentation. *Python Documentation*. https://docs.python.org/3/index.html.

pyttsx3 Documentation. *pyttsx3 Documentation*. https://pyttsx3.readthedocs.io/en/latest/.

Ubuntu. Ubuntu 20.04.6 lts (focal fossa) image, 2023. https://releases.ubuntu.com/focal/.

Unity. Unity version 2022.3.10f1, 2023a. https://unity.com/releases/editor/whats-new/2022.3.10.

Unity. Unity hub, 2023b. https://unity.com/es/download.