Product: CLYDE

Team: Cloud 9

Abstract

ClYDe is a system that sanitises desk tops and other flat surfaces in high-traffic spaces such as libraries and offices. The team has achieved the main goals set out for this demo: environment mapping, navigation, detection of QR codes and detection of obstruction using computer vision. Additionally, the team was able to start working early on future goals such as attaching the robot arm and the companion app.

Task	Contributors
Navigation/ mapbuilding	Sean, Josh, Janek, Lars
QR code/obstruction detection	Ivan, Jeffrey, Shining
Robot arm integration	Adel, Lars
App	Ivan, Jeffrey, Ryan
Hardware testing	Adel, Lars
Demo video	Ryan, Janek

1. Project plan update

The following milestones were planned for this demo:

- Navigating to a desk - achieved
- Environment and Pathfinding - achieved
- QR code/ obstruction detection - achieved

1.1. Deviations and changes

Due to its importance for the success of the project, we chose to start working on integrating the robot arm and programming cleaning motions earlier than planned. Due to rapid advancement in computer vision we were able to start work on the mobile app as well. Besides this we will not be making any modifications to our goals for the next demo, but we will have a stronger emphasis on integrating the parts we have.

We have used the same organisational tools as described in the demo 1 report. However, this time we have made extensive use of GitHub to structure our development. The team has made effective use of branches to isolate the development of different features, like the robot arm as well as mapping and navigation.

1.2. Budget Spending

Money: £0 as of the 28th of January 2021.
Technician Time: Approximately 3hrs per week, spent mostly on testing out hardware components.

1.3. Individual Contributions

The team structure has predictably changed slightly, as many hardware tasks are completed. The hardware team has been able to assist in other areas like navigation and app development. As a group we feel that there is no need to deviate from our plan, except for continuing to work on the app and robot arm, which again was started earlier than planned. A significant amount of the work for this demo was done to resolve different errors that arose using the various ROS2 software packages that are described later. Different errors arose for different people with very similar computer setups. However most of them were resolved thanks to looking into package details and consulting with experts. Difficulties also arose from lack of documentation in some of the packages used.

2. Technical details

2.1. Hardware

2.1.1. Pincher x100 Robot Arm

We were able to acquire working Webots models for both the arm and the TurtleBot(PROTO files). However integrating them posed some challenges. At first the arm was modelled as an independent robot attached to the Waffle Pi. There is little documentation and few examples with multiple robots in the packages so some experimentation was required. The conclusion was that having the arm as a separate robot attached to the TurtleBot caused interference with navigation. The arm was then converted to a Webots solid, and with a few adjustments in the code we were able to make the arm work with navigation. The end-effector design that we ended up going with resembles squeegee window cleaner. It was modelled in Blender and imported into Webots so that it could be attached to the end of the robot arm. This way it also has longer reach.

2.1.2. 6-12V R385 DC Diaphragm Pump:

After conferring with the technicians we decided that the best approach for distributing sanitation liquid is to use a pump attached to a reservoir on the robot base. The pump will move the liquid up the arm in plastic tubing to the cleaning tool. The activation can be done easily through the Raspberry Pi as demonstrated in the demo video. An air pump was also demonstrated. The R385 was chosen because of its availability, ease of use and low price. Due to the SDP lab rules, we will not be able to run tests with liquids on the robot.

2.2. Software

The main software milestones to be achieved for this demo involved the basic navigation aspects of the system, along with the computer vision required for QR code and Obstruction detection. These were completed as planned.
The approaches taken in the completion of each milestone are explained below.

2.2.1. Core robot control and navigation packages:

The main framework used in software development is ROS2 (Robot Operating System 2). This is due to the abundance of packages available, and the fact that much of the same code can be used in real and simulated scenarios. As opposed to writing controllers directly with Webots, this allows us to much more easily make the robot function in a real scenario. Using just ROS was considered, but ROS2 was chosen due to the amount of good examples relating to Webots. We also wanted to test the capabilities of the new ROS version after having mixed experiences with the older ones in previous courses. Central to the software development is the webots_ros2 package provided by CyberBotics, the developers of Webots itself. This package allows for easy integration between ROS2 and Webots, and contains many examples. These have been used extensively, as they could be easily adapted to our needs. Chief among these is the Webots ROS2 turtlebot example which allowed us to get our core robot simulation running quickly. Examples allowed us to get core components running much easier, and to understand the workings of ROS2. An additional core package is turtlebot3. This would be used to control a physical version of the robot, but also allows our simulation to use the powerful navigation-related packages described below. It also provides a URDF robot description that we used to import the Waffle Pi into Webots.

2.2.2. Navigation

Creating the floor plan framework
As confirmed in the previous demo, navigation must rely on a precomputed floor plan. This would either be created manually, or by using SLAM (Simultaneous localization and mapping). SLAM is creating or updating a map of an unknown environment while simultaneously keeping track of the robot's location within it.
After speaking with the expert Christopher McGreavy, we investigated two suggested methods for navigation:

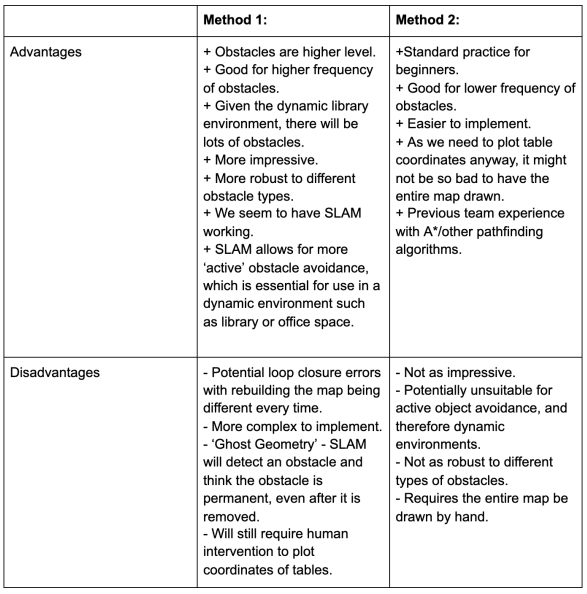
Method 1: using SLAM for real-time map-building and obstacle avoidance.
Method 2: manually 'pre-draw' the map and use naive obstacle avoidance which involves 3 actions: Turn left, turn right, or move backwards.

As a team, we decided on **method 1** as it allowed us to implement a more general solution. The desk coordinates must be hard-coded, due to the difficulty of identifying tables on the map from LIDAR data, which is not a problem as we already need to drive the robot around for mapping the room and can store coordinates in front of tables. With this, we can create a map of any environment. However in the case of failure we are prepared to switch to method 2 for a more basic solution. A short clip of the map building in action is included in the demo video.

Cartographer
Cartographer is used to create a map of the environment using LIDAR data. We can drive a robot around using the turtlebot3_teleop . The map created can then be used by pathfinding. Turtlebot3_cartographer uses the cartographer and allows easy integration with the TurtleBot.

Pathfinding
The Nav2 package uses a map generated by the cartographer, together with SLAM (to avoid new obstacles not present on the map), to compute a path to a goal state. Turtlebot3_navigation2 uses Nav2 to provide pathfinding capabilities specifically tailored for the TurtleBot3. It allows you to simply set the goal pose coordinates within the precomputed map via the command line. The current implementation involves manually specifying the goal pose for the robot. The final step, to be completed for demo 3, is therefore to automate this functionality in order to work alongside the companion app.

Obstacle Avoidance
The Nav2 package includes static and moving obstacle avoidance when pathfinding if provided a precomputed map. Furthermore, it can also avoid objects not present on the map, but are detected by the LIDAR as the robot moves. Again, a short clip is included in the demo video showing this active obstacle avoidance.



	Method 1:	**Method 2:**
Advantages	+ Obstacles are higher level. + Good for higher frequency of obstacles. + Given the dynamic library environment, there will be lots of obstacles. + More impressive. + More robust to different obstacle types. + We seem to have SLAM working. + SLAM allows for more 'active' obstacle avoidance, which is essential for use in a dynamic environment such as library or office space.	+Standard practice for beginners. + Good for lower frequency of obstacles. + Easier to implement. + As we need to plot table coordinates anyway, it might not be so bad to have the entire map drawn. + Previous team experience with A*/other pathfinding algorithms.
Disadvantages	- Potential loop closure errors with rebuilding the map being different every time. - More complex to implement. - 'Ghost Geometry' - SLAM will detect an obstacle and think the obstacle is permanent, even after it is removed. - Will still require human intervention to plot coordinates of tables.	- Not as impressive. - Potentially unsuitable for active object avoidance, and therefore dynamic environments. - Not as robust to different types of obstacles. - Requires the entire map be drawn by hand.

Note
In the images and clips seen in the demo video, the BurgerBot is used for demonstration purposes, rather than the custom WafflePi robot created by the hardware team. This is due to issues with the robot URDF file, which was suggested by expert Thomas Corbères. These issues should be

resolved by demo 3, and converting to the WafflePi should not affect any of the navigation functionality displayed above.

2.2.3. Image Processing and Computer vision

Obstruction detection

For any vision processing, we have chosen to work in Python and use the OpenCV package as its library contains comprehensive functionality for working with image data. The Numpy package is also used to perform any matrix manipulation required to support this. Both of these are de facto standard when working with computer vision, as they have good performance, and are well documented.

Approach:

1. The OTSU thresholding (Otsu, 1979) algorithm that separates the foreground and background of an image, we choose it as it's invariant to an image's brightness and contrast. This also deals with different surface colours, illuminations, shadows and reflections. This is the algorithm used for detecting objects on the desk. The demo video includes images displaying the results of OTSU thresholding in various scenarios.

2. By assuming that the robot can navigate to the same position relative to each table, we can encode the desk surface coordinates in a database that the robot can access. If this assumption is not valid, a 'back-up plan' has also been developed which uses edge detection to calculate the desk boundary. Canny edge detection (Canny, 1986) is used for detecting edges based on gradient of the image. See Figure 1. Using detected edges, we perform Hough line detection (Ballard, 1981) to find the analytic expression of each line. After this, four straight lines representing the edges of the desk can be obtained. Then, we find the intersection point of each line which represent the corners of desk.

3. Finally, using the encoded area and result from OTSU, we perform a Perspective Projection visualization table's surface on a new plane. This helps with any debugging requirements in later stages.

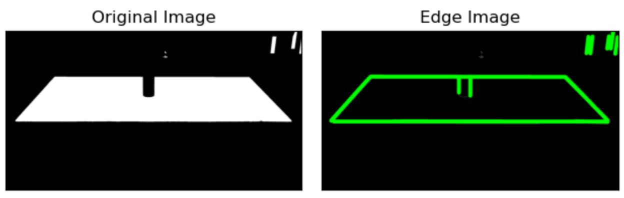


Figure 1. Canny edge detection in use

Dealing with Different Colours of Desk

1. Light surface colour table. Image converted into a grayscale picture and steps 1-3 applied. The table is segmented to be white and obstructions are segmented to be black. Simply checks if there is any black coloured pixel within the encoded area.
2. Dark surface colour table. Split three channels of the original image and steps 1-3 applied. For a proper result, the table is segmented to be black and obstructions are segmented to be white. Checks if there is any white pixel within the encoded area on all three images.

QR code detection

The ZBar python package is an open source software suite for reading barcodes and QR codes. It was chosen due to its ease of use and good performance. The function pyzbar.decode() can detect all QR codes it finds in the image and extracts the information stored in them. The QR code also helps to locate the edge of the table. The code detection will be conducted by both the robot and visitors using the app, using the same code. Therefore, we ensure that the detection should be a stable process.

Mobile App

Work on the mobile app is still in an early phase. Still, the group had decided on a design so each of the sub-teams know what they should do to interface with the app. It will be made for Android phones, in Android Studio. We chose this because the open-source nature of android development makes it accessible, and because Android Studio offers a wealth of tools that help with many steps of the development process. For ease of access, each desk will be kept in a central database accessible by both the robot and the app. When using a desk, the user scans QR code to mark the corresponding desk as occupied. Users will check out the desks when they are done and they are marked as dirty. The robot will make a path between dirty desks, and the path is updated when a new desk becomes dirty. Dirty desks will be cleaned, and their entries will be updated as such.

Database

Using the mysql python package, the database will store information on each desk. The information will be: surface colour, length, width, height, encoded area,goal pose for navigation and status. MySQL was chosen for ease of use and because it is well-tested package. The databases can be remotely connected to. If some connection errors arise during the network connection process, and the remote connection becomes hard to accomplish, we will switch to using cloud databases.

3. Evaluation

3.1. Navigation QA

To test and expose the limitations of the navigation2 package, we repeatedly navigated the robot to each of our tables in the test environment built by Sean which is shown in the video. Table 1 shows the robot's position and orientation (pose) from its starting point. Each table was given a goal pose during the initial mapping by noting the robot's odometry information at each table, these values are noted in the goal pose column. There are 4 values each for position and orientation (w, x, y, z). For position w and z are always zero, and for orientation x and y are always zero, so below the first two values refer only to position's x and y, and the last two values refer only to orientation's z and w. Each table was tested and each test was conducted 5 times. The mean final position of the robot (when successful) for each table also shown.

TABLE	GOAL POSE	MEAN ACHIEVED POSE	SUCCESS
1	0.35 0.90 0.70 0.70	0.46 0.99 0.80 0.61	5/5
2	-1.20 0.50 0.70 0.70	-1.33 0.53 0.50 0.87	4/5
3	0.3 -4.7 0.0 1.0	0.24 -4.5 5 0.10 0.96	4/5
4	0.3 -3.7 0.0 1.0	0.23 -3.62 0.21 0.95	5/5
5	0.3 -2.7 0.0 1.0	0.18 -2.81 0.09 0.99	5/5
6	1.8 -4.7 0.0 1.0	1.77 -4.88 -0.09 1.11	3/5
7	1.8 -3.7 0.0 1.0	1.64 -3.58 -0.03 1.06	4/5
8	1.8 -2.7 0.0 1.0	1.73 -2.64 0.05 0.91	3/5

Table 1. QA for navigation

The results of the tests are favourable with a 82.5% success rate, but there are some failures to note, the most predominant of which were: the navigation2 package crashing unexpectedly, failure of the navigation2 recovery system when encountering an obstacle, and the robot not seeing and crashing into some obstacles that are too small such as a table leg. We also captured some video of the robot successfully handling an obstacle that is not present on the provided map.

3.2. QR code detection QA

We analyzed the performance of the detection algorithm by holding the camera at different angles and from different distances to QR code. The result of the detection can simply be classified as 0 for failure and 1 for success.

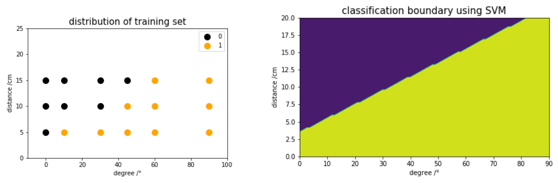


Figure 2. Support Vector Machine

To show stability of the detection algorithm, we predict the performance of the QR code detection using support vector machine(a machine learning algorithm which helps decide the decision boundary). The yellow part is the area where the detection algorithm succeeds and the purple part is when it fails. As shown in Figure 2, the detection algorithm is stable as long as the camera is not held at a small angle or far from the QR code. The data and SVM boundary show a clear linear relationship. The QR code requires a full exposure to the camera. In our user scenarios, the visitors will be able to remove any obstructions on QR codes manually. A precondition for robot to clean is that there is no obstructions on the table. This also means a blocked QR code is unlikely.

3.3. Obstruction Detection QA

For obstruction detection, we vary different factors that might affect performance of the algorithm as shown in Table 2. Due to the limitation of computer vision, the algorithm fails to detect when there is an obstruction with same color as table or there is a dark shadow on light-colored table. Our back up plan is making use of LIDAR to solve this.

Table 2. QA for Obstruction Detection

Attr. to change ↓	Bright-ness/cd	Obs. pos	Obs. size	Obs. color	Table color	Reflect light	Shadow (Gradi-ent)	Out-come
Illumina tion	0-20	middle	middle	light	brown	None	None	Success
	20-40	middle	middle	green	brown	None	None	Success
	40-60	middle	middle	green	brown	None	None	Success
	60-80	middle	middle	green	brown	None	None	Success
Position	40-60	left	middle	green	brown	None	None	Success
	40-60	right	middle	green	brown	None	None	Success
	40-60	top	middle	green	brown	None	None	Success
	40-60	bottom	middle	green	brown	None	None	Success
Size	40-60	middle	small	green	brown	None	None	Success
	40-60	middle	large	green	brown	None	None	Success
Obs. color	40-60	middle	middle	brown	brown	None	None	Success
	40-60	middle	middle	black	black	None	None	Fail
Table color	40-60	middle	middle	brown	dark	None	None	Success
Reflect light	40-60	middle	middle	green	brown	Yes	None	Success
Shadow	40-60	middle	middle	green	brown	None	low	Success
	40-60	middle	middle	green	brown	None	high	Fail

4. Budget

Money: Currently the robot contains the parts specified in demo report 1, in addition to the 6-12V R385 DC Diaphragm Pump, that costs £5 and plastic tubing for £10. This brings the total construction cost of the robot to £2,030.4.

Technician Time: In week 4 the hardware team met with Garry Ellard to discuss designing and manufacturing the robot's cleaning tool. During Reading Week and the last week we have spent additional time with the technicians testing out various hardware components, specifically using an arduino to remotely activate LEDs, motors and a pump. This took up approximately 3 hours of technician time each week, due to the major delays that come with working with hardware remotely.

5. Video

Please follow the link below to watch our demo video: https://uoe.sharepoint.com/:v:/s/SDP2021-Group-9/ERzHRZk3MqhFj7kgTJioZv0BH5vX79cmTg5gPqO9r-t7OQ?e=QjG5ks

References

Ballard, Dana H. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

Canny, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986. doi: 10.1109/TPAMI.1986.4767851.

Otsu, N. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979. doi: 10.1109/TSMC.1979.4310076.