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. For this demo, the team has started working on integrating the different subsystems we have demonstrated in the previous demos. In addition, we have continued work on the phone application that was started in demo 2.

1. Project plan update

The following milestones were planned for this demo:

- Successfully disinfecting a desk surface - Achieved.
- Working mobile phone application - Partly Achieved.
- QR code/ obstruction detection - Achieved.

Due to the rapid progress that was made for demo 3, we were able to start working on integrating different subsystems of the robot together earlier than planned. However, developing the app has proved more time-consuming than initially thought and we were not able to implement all of the functionality we desired for demo 3. We are confident that a working version of the app will be completed by demo 4. Since we have been able to start integration early, we can spend more time before demo 4 on polishing individual components and making sure the robot is user friendly. The group has organised goals in the same way as for previous demos: Git branches helped us efficiently integrate the robot's subsystems; Trello allowed us to delegate tasks between us; weekly meetings in Discord meant we could discuss progress and help each other achieve our goals. Unfortunately, several team members experienced significant internet problems which has impeded our progress in many areas. Additionally, a large portion of our work was focused on overcoming insufficient package documentation and compatibility issues.

1.1. Budget

Money: £10 as of the 10th of March 2021. This money was spent on cloud hosting our database so it would be easily accessible to all.
Technician Time: Less than 1hr. The technicians were not needed much for demo 3, but they were consulted for some hardware questions.

1.2. Individual Contributions

See the table below for individual contributions by task.

Task	Contributors
Navigation & database integration	Sean, Josh, Janek, Lars
Robot arm kinematics	Adel, Lars
CV and arm integration	Adel, Lars
App	Ivan, Jeffrey, Ryan, Shining
Demo video	Ryan, Janek

2. Technical details

2.1. Hardware

2.1.1. Zee Lipo Battery

Unfortunately, given the current battery provided with the physical turtlebot and our attached devices the robot has a battery life of about ½ an hour. The battery is 11.1v at 1800mAh. After speaking with technicians, we know that increasing the mAh while keeping the voltage the same would allow us longer battery life. Technicians recommended the "Zeee 3S 11.1V 100C 8000mAh hard case RC Lipo Battery". Depending on usage, this will more than quadruple the battery life of the turtlebot. The battery weighs 500g, which, according to technicians can easily be carried by the turtlebot. The battery is rechargeable. Other batteries were considered, but this was chosen because of its prize and capacity.

2.2. Software

2.2.1. ROS2 actions

Robot arm control and navigation both use ROS2 actions, a combination of the two main ways of sending messages in ROS2, which are topics and services. They are well suited for longer running tasks, as constant feedback about the state of the action is provided, as well as a clear indication of final success or failure. Using ROS2 in general is much preferable to writing code directly in Webots, as it is portable and much of the code can be used as-is in a physical system.

2.2.2. Arm

The milestone for creating the software to control the arm has been completed. The arm is capable of cleaning the desk by swiping the squeegee across the surface of the desk. In order to clean the table, disinfectant would be distributed to the sponge throughout the whole cleaning motion through low volumes by the pump. Since this is not something we can test at all, we cannot say what volume of disinfectant would be pumped at what rate.

The webots_ros2 package(described in demo 2) contains

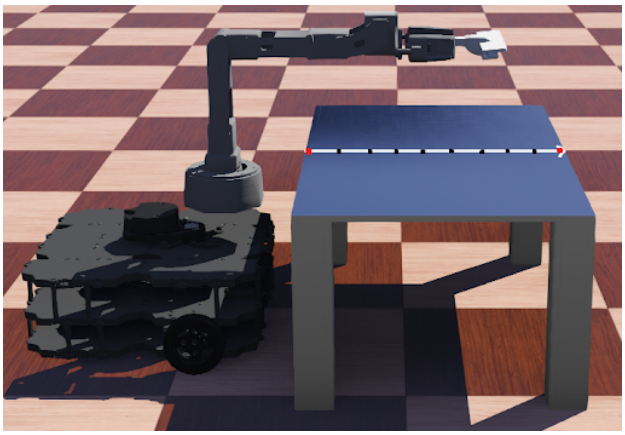

Figure 1. A single swipe across the desk. End points and intermediary points are shown. This image represents a swipe with **noOfPoints** parameter set to 10.

the TrajectoryFollower class that implements the 'follow_joint_trajectory' ROS2 action using this and examples in the package we were able to construct a simple controller for the arm. Using the example code allowed us to easily get the controller running, and it was easily adapted to our needs. The controller takes a list of joint states, velocities, accelerations and time to complete the action. When conferring with the webots expert we were told that this is an acceptable method.

The ikpy python package was used to calculate inverse kinematics for the arm. We used it because it provides very useful utilities for kinematics of a robot arm which would have been time consuming and error-prone to implement ourselves. Two features were used from this package. The first is the ability to create a model of an arm from a given Unified Robot Description Format(URDF) file. In order to use this feature, we modified the URDF of the arm such that it included the squeegee on the end and passed that file to ikpy. The second feature is a function which uses the model to calculate the joint states of the arm needed to reach a given 3D point along with an orientation. These are then passed to the controller, along with some predefined velocities, accelerations and times.

The software was created in a way such that the height of the cleaning motion as well as the limits of the waist movement is generalised. This means the arm cleaning motions can be adapted to any size table, given that the dimensions are suitable for the arm's reach.

The cleaning motion consists of 'swipes'. A swipe is defined as the act of the arm rubbing against the table along the whole length in one direction. The swipe has a set start and end point, seen in Figure 1 below as the red dots. Several points are then generated in between the red dots, so the arm goes across the table while keeping it's end effector against the surface the whole time. The number of these intermediary points (shown as the black dots in Figure 1) is generalised and can be changed as a parameter. In Figure 1 there are 10 points in total (meaning 8 intermediary points), which is how the robot is currently set up.

The joint angles are only computed for one swipe in practice. This essentially means the inverse kinematics only really have to work on a 2D plane, since the y value is constant. The joint angles that are calculated for this swipe are then re-used with different waist values in order to swipe across the whole surface. This minimises the number of inverse kinematics calculations that have to be performed.

Parameters of the arm software are as follows:

- **timeToDoOneSwipe** - this is the amount of time the arm takes to swipe across one length of the table (as seen above, from red dot to red dot). The value we settled on is around 10 seconds for now. Bringing the time too low reduces the stability as the in-built ros2 controller for the arm struggles to control the inertia when carrying out the trajectory.
- **noOfPoints** - this is the total number of points a swipe will have (including the red dots). As previously mentioned, we set this to 10 to get an accurate trajectory across the surface.
- **Height** - this is the height at which the arm will perform the cleaning motion at. The height of the table can be plugged into this parameter and the arm will work with it.
- **waistRange** - this is the range of values that the waist will take when carrying out the swipes. By default we have set the range as -1 to 1 radians, with 6 swipes equally spaced in that range. This range cleans the desk we have all the way, and 6 swipes ensures the squeegee makes contact with everywhere on the desk. See the Figure 2 for a visualisation of how the limits work.

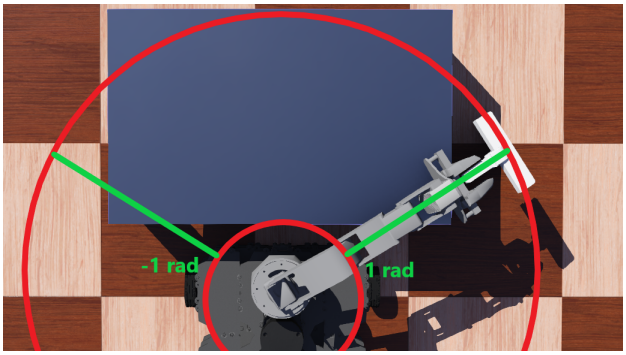

Figure 2. A visualisation of the range of motion the arm can move in to clean the desk. This image represents a trajectory with **waistRange** parameter set to (-1, 1, N/A). The number of swipes is undefined since it is not shown in this image.

2.2.3. Mobile App

IDE : We have chosen Android Studio 2021 as our developing IDE. Using Android studio, we can easily convert

written code to kotlin (Android Studio's default language) and deploy the app into a real Android Phone for testing purposes. It also contains a wealth of tools that make development significantly easier, from database access, UI creation and emulating different phones to test the app on. The open-source nature of Android makes it easy to find good resources for development. In addition, it is the OS with the largest market share (statcounter, 2021). Below is an overview of the app's functions and design.

Purpose: This app will allow library visitors to book desks so the robot can efficiently clean by only visiting 'dirty' and empty desks.

System Configuration: The main function of this app is to check in and check out a certain desk for a certain time by scanning the QR code located on the desk. After checking in, the user can begin to use the desk. When leaving, the user must 're-scan' the QR code to check out of the desk. The app will then connect to the database and change the status of the desk to "dirty".

Data Flows: As of now, this app will not store any personal information. (No login needed to access the app). Data flows only occur when users use the app to check in and check out the desk. The app will send the request to the database to change the status of the desk.

User Access Levels: Any Android phone user will have access to downloading this app.

Contingencies and Alternate Modes of Operation: This app serves as a very basic booking system and there should be no alternate modes of operation. The movement of the app is linear when used correctly. More detailed analysis is demonstrated in the Usability Analysis section.

Logging on: User clicks on the app and enters the log-on screen, where the logo is attached to the page along with a button which reads "Request". Clicking on this button, will direct the user to the QR code scanning interface. Right action: User clicks the "Request" button and jumps to the scanning interface.

QR code scanning checking in & checking out: When initially opening the QR Code scanning interface, the app will ask for the approval for using the camera on the user's phone. The interface shows "Please scan QR code on desk" as an instruction. Below the instructions there will be two buttons. One is "CHECK IN", the other is "CHECK OUT". The interface will contain a square image which showing the camera live feed. The user must give the app permission to use the camera in order to perform OR code scanning. Right action: User clicks the right button and scan the QR code.

Time slot choosing: If the user clicks the "CHECK IN" button and successfully scan the QR code. There will be three buttons at the bottom of the screen which write "1", "2", and "3" respectively. The numbers mean the duration in hours the users want to use the table for. Right action: User chooses the time duration they would like.

Exit System: Users can exit the app in any of the previous steps. There will be no blocking during the whole process.

2.2.4. Navigation

The main navigation goal for this demo was to automate the process of moving the robot. Before, it was necessary for a person to command the robot to navigate to each table manually; this conflicts with our final goal of having a fully autonomous robot. To achieve this automation we needed to fulfill the following requirements: Set the initial pose of the robot; query the database for the goal location of a desk; send the goal to the robot. We successfully implemented all of these requirements with the creation of a package of python scripts that interact with navigation2 to move the robot. These scripts can be found on our github. By completing these goals, we also completed the integration of navigation and the database, a major step towards creating a finished product.

To query the database, python scripts were created with the use of the mysql-connector-python packages. The database was then updated to include the goal poses of each desk. In practice, the database would be completed by the professional that initially drives the robot around the space while it maps using the cartographer package, as noted in the previous demo report. This professional would drive the robot to an approximate goal pose for each table and insert a new row to the database with the table number and the robot's current position. Later, when the robot is in operation and cleaning desks, it can read this position off the database and navigate to it. This was the exact process we used for our test environment. These scripts include queries to select all 'dirty' desks and the ability to update a desk 'state' from 'dirty' to 'clean'. To combine all of the package together, a 'main' script was created to implement the overall control flow of the robot. This control flow can be seen in the diagram below.

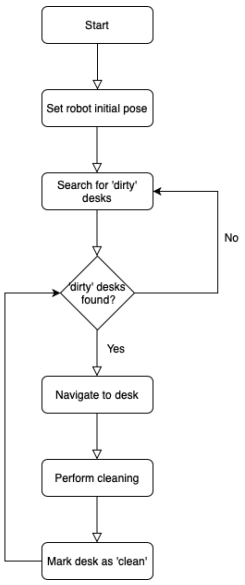

Figure 3. A flowchart visualising the current control flow of the robot, provided by the 'main' script.

The demo video also includes a clip of the robot successfully navigating between multiple desks through the use of our scripts to demonstrate this autonomy.

3. Evaluation

In demo 2 the team provided QA for several major components, demonstrating that they functioned to satisfaction. For this demo, the team did not develop any new features suitable for quantitative analysis. We chose not to spend significant time doing QA on the robot arm. This is because high precision is not needed for this aspect of the project, and the arm functions correctly, as demonstrated in the video. A large amount of work was put into integration, however since the integration included previously tested parts, we felt our time was better spent elsewhere. We found that the app was not suitable for QA and have instead conducted a usability analysis.

3.1. Usability Analysis

This section contains a usability analysis (self-reflection) on the UI of the app. We decided to implement the app on Android, but if we were to continue working on the project beyond the scope of this course, we would extend the app to operate on more platforms. We understand the restrictions of creating the service only for smart phones. This might cause issues to users who do not have or refuse to use smart phones. A solution to that would be, taking University of Edinburgh Main Library as an example, implementing another interface on the computers of the Library Help Desk. The following subsections reflect on our design based on Norman's Action Cycle, as described in the week 8 Usability Workshop.

3.1.1. Low-Level: Perceiving the state of the world & Execution of the action sequence

We have identified two issues that could arise in the low level, execution lapses (failure in users' memory) and misperceptions. In our design, execution lapses could happen when the user forgets to check in or check out the desk they are using in the library. This means that there will be a contradiction between the database and the real world. This could be solved by fetching all tables that have not been cleaned in a certain amount of time, and cleaning them at regular intervals. Misperceptions happen when the users are not aware of the state of the desk they have chosen to use. For instance, users may not be aware of the fact that the desk they want to use was just marked as dirty by the previous user, but still has not been cleaned. This could be resolved by enforcing the fact that dirty desks can not be checked out, and displaying a warning to the user. Both usability failures could result in system failures.

3.1.2. Mid-Level: Interpreting the perception & Sequence of actions

The issues that could happen within the middle level are problems in translation to machine I/O. The first potential problem is that the users might not be able to produce input that the app understands, or produce input that is misinterpreted. This might happen if the user presents some other QR code or barcode to the camera than it expects. Since the app is designed to only respond to specific codes, it will receive some information it cannot process, which could lead to the system crashing. This could be mitigated by doing basic input checks on the QR codes. If they don't represent an expected input, they should be ignored and a note should be issued to the user. Another potential problem is that users may not understand or misinterpret the information from the app. This is unlikely to happen since the app is quite linear, it only generates very simple outputs and instructions.

3.1.3. High-Level: Evaluation of interpretations & Intention to act

An issue that could happen at the high level could be that users may be unable to recognise goal satisfaction. The first potential problem is that users may not determine the correct thing to do, or make the wrong choice when interacting with the app. This is unlikely to happen since the goal should be pretty straightforward to all users. Another problem is the uncertainty about outcome or mistaken belief that have made progress towards the goal. To mitigate this, we could communicate the state of the app and any issues/errors very explicitly to users. In the future, when the booking system is implemented in more platforms, different bookings might crash with each other but that issue would be resolved when they are implemented.

4. Budget

Currently the robot consists of the following parts.

Part	Cost(£)
TurtleBot 3 Waffle Pi	1,276
PincherX 100 Robotic Arm	505
Raspberry Pi kit	210
Zeee Lipo Battery	60
C270 Logitech USB Webcam	25
6-12V R385 DC Diaphragm Pump	5
Total	2,081

Technician Time: Less than 1hr. Technicians have not been needed much for this demo.

5. Video

Please follow the link below to watch our demo video: https://uoe.sharepoint.com/:v:/s/SDP2021-Group-9/EbC3zhH8fDNDi-WfWayAKMEBKheFRc0p3lc3zPjB3W2I2Q?e=nwwAD1

References

statcounter. Operating system market share worldwide. 2021. URL https://gs.statcounter.com/os-market-share.