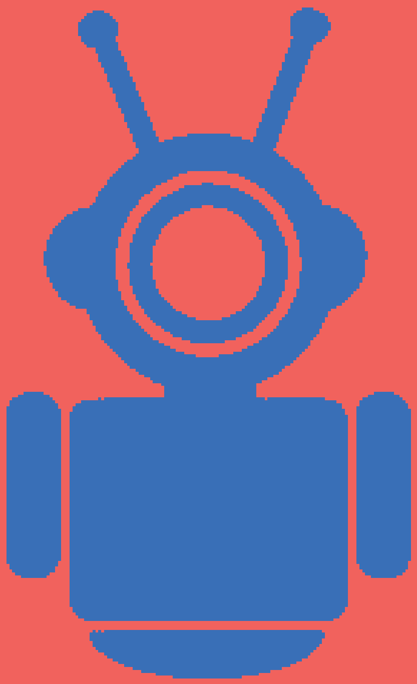
CODE DOCUMENTATION



Automated Kiwifruit Picking

Test Drive

CONTENTS

Table of Contents

- Functionality........................4
- Design Assumptions..............21
- Constraints..........................22
- Design Principles..................23
- Design Decisions..................26
- Generator.............................27

CONTENTS

Table of Figures

Figure 1.0: Beacon System..........................7
Figure 2.0: Picker Finite-State-Machine..........9
Figure 3.0: Carrier Finite-State-Machine........11
Figure 4.0: Obstacle Avoidance Cases............13
Figure 5.0: Obstacle Avoidance Cases............14
Figure 6.0: Obstacle Avoidance
Proceedure Diagram............................16
Figure 7.0: Old Graphical-User-Interface........18
Figure 8.0: New Graphical-User-Interface.......19
Figure 9.0: Wiki Navigation Tab...................21
Figure 10.0: Wiki Navigation Tab.................27

FUNCTIONALILTY

Movement.........................4
Path Planning.......................7
Obstacle Detection &
Avoidance......................12
Animal & Person Behaviour......14
Graphical User Interface........15
Configurable Orchard............16

MOVEMENT

- The movement for all Entity classes are constructed from 6 basic movements. There are essentially two types, forward movement & rotation.

MOVE FORWARD

- For move forward there is only one specific movement which is move forward. It will receive the distance in the positive direction to travel. Eg 5m in North (positive y) will have a distance of 5. Eg 4m in West (negative x) will have a distance of -4.

ROTATION

- For rotation there are 4 preset rotations, face North, East, West, South. These will make the entity rotate in that direction with settable velocity. The last movement for rotation is the configurable rotate. For this movement any desired angle relative to the x axis (CCW) can be specified along with velocity and the entity will rotate to that angle.
- Note: Velocity must always be set to positive.
  Angles are in radians and x,y movements are in meters.

ENTITY MOVEMENT

- The implementation of movement is done by adding sets of basic movements to a movement queue. Movement was chosen to be implemented as a queue. This was due to the fact that movements can be added to the queue and then executed. This allows all subsequent desired movements to be add all at once. The entitiy will move according to the first item in the queue and set the linear or angular velocity accordingly. To confirm that the movement has been executed correctly the queue will convert the distance travelled to the absolute position in Stage(base ground of truth). Next, this value will be compared to the entity's current position (its base ground of truth) to ensure that the entity is in the desired location. If it is then the basic movement is completed and removed from queue so the next basic movement can now be executed.

MOVEMENT

- There is another queue which is the avoidance queue. It essentially does the same thing as a movement queue. However, it runs first over the movement queue and stores the basic movements for avoidance. The main reason for this separate queue is so that it can make checking the status of avoidance movements easier.

- For example currently a robot is at 0,0 facing North. It wants to get to 10,10 North. Then we can first add Face East (which rotates the entity east) then add move forward 10. Now the robot is at 10,0 East. Then add Face North then add move forward 10 to get to the destination of 10,10 North.

- Code specific
  To add to the movement queue the following method can be used.
  addMovement(...) - this takes the type of movement ‘forward_x’, ‘forward_y’ or ‘rotation’, the distance to move and velocity
  FaceNorth(...),FaceWest(..),FaceEast(...),FaceSouth(...) - this takes only velocity and will face the right direction.
  To actually move the entity the move() method must be called in a loop.

MOVEMENT

PICKER SPECIFIC MOVEMENT

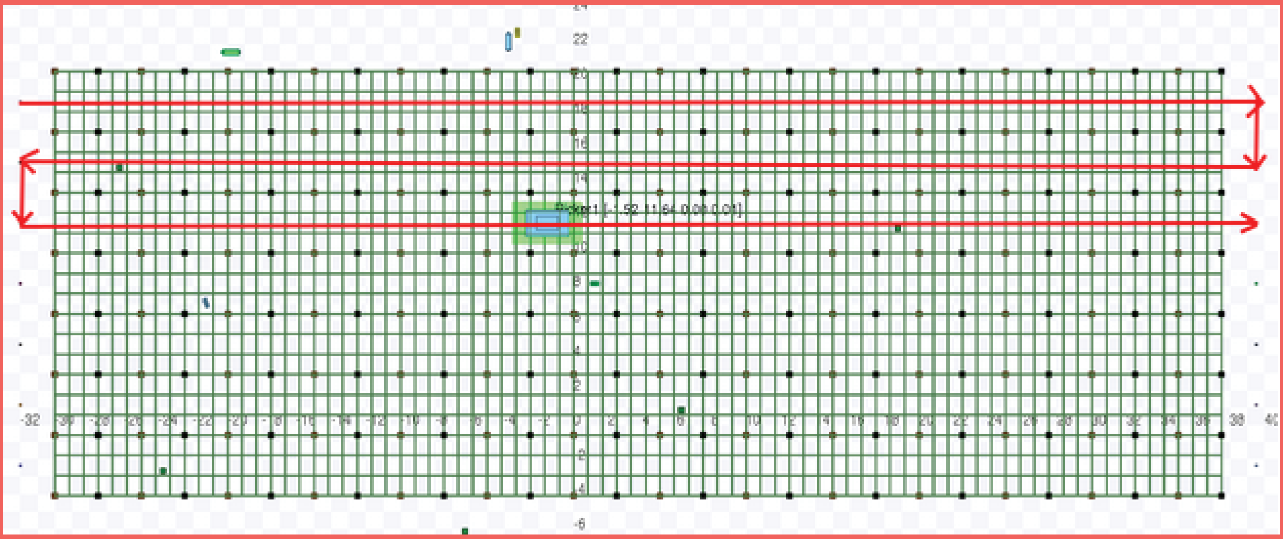


Figure - 1.0

- Two main design options were considered regarding Picker Robot navigation; detection based and beacon based navigation. Detection based navigation would mean the Picker robots would use their laser scanner to find the start of a row of kiwifruit trees. It would then navigate into the row and find the nearest trees to begin picking fruit. Upon reaching the end of a row the Picker would need to calculate which way to turn (into the left row or right row). This would all need to be done using only data from the laser scanner. Hence this method of navigation was deemed too complex for implementation within the time frame of this project. The advantage of this method was that, if implemented correctly, the Picker would not have needed knowledge regarding how wide and long rows were as well as how many rows there were in the orchard.

- Beacon based navigation meant placing beacons on the ends of each row of kiwifruit trees within the orchard and then using those to set the fruit picking path of each Picker robot. In the simulation these beacons are visible as nodes on both ends of each row. This solution simplified navigation drastically in comparison with the first option. The fundamental concept of it is that a Picker robot will receive the position of the next beacon on its picking path and then it will set its destination to that position. Depending on the name of the beacon that it is currently moving towards (beacons in the orchard are numbered) the Picker will know if it is moving through a row or if it is moving to a new row.
  - If the Picker is moving through a row it will make use of its laser scanner to identify kiwifruit that it can pick.

MOVEMENT

- Due to the placing of beacon nodes as well as their movement being strictly vertical and horizontal, Picker robots will never collide with trees or poles within the orchard.

- Due to the placing of beacon nodes as well as their movement being strictly vertical and horizontal, Picker robots will never collide with trees or poles within the orchard.

- This solution took into account the possibility of a kiwifruit orchard changing size (much like the detection based solution) because all that is required when expansion occurs, is to place beacons on each new row. Though this is not as elegant as the detection based solution it was the form of system extensibility that was viable for implementation within the project time frame.

- The disadvantage of this solution is that it relies on the beacons being placed correctly. If the beacons are placed too close to a tree then the Picker may collide with fruit trees. If the beacons are placed too far from a horizontal line of fruit trees then as it is moving through that row, the Picker robot’s laser scanner may miss the kiwifruit that are closer to the trunks of fruit trees.

PICKER WORK SPLIT

- When the system is started, an algorithm uses the number of rows and Picker robots within the system to fairly distribute the workload to each Picker. The workload refers to the number of rows of trees a Picker needs to pick kiwifruit from.

- To assign each Picker to the rows that it has been delegated, the Picker is given the starting beacon and finishing beacon of its picking path. With this information the Picker is able to calculate the missing path by listening to the beacons in between the starting beacon and the finishing beacon.

MOVEMENT

PICKER STATE DESIGN

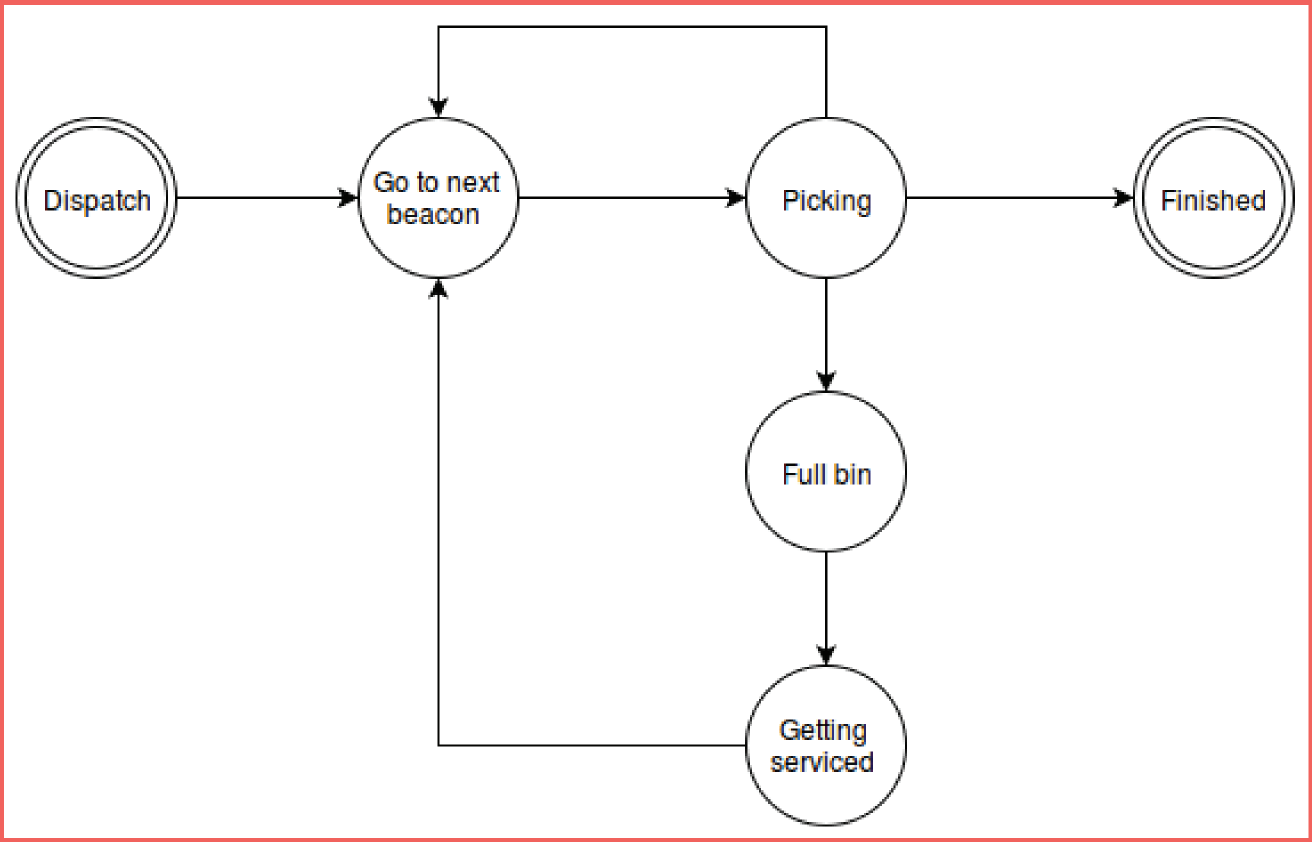


Figure - 2.0

- The Picker robot’s AI was designed to be state-driven. This design was chosen instead of other candidates because by using a finite state machine, this was the most easily modelled and understood. Considering two people were working hand in hand on this mission critical component of the system, it was of utmost importance that they were both on the same page in regards to the design of the Picker’s AI.

- Upon being created, the Picker robot begins in the “Dispatch” state. At this point the Picker robot already knows which kiwifruit picking path it has been assigned. It will subscribe to the robotic beacon at the start of its path and will wait until it begins to received pose messages from that beacon. Once it learns the location of the beacon it will move to it. Once it reaches that beacon it will switch to the “Go to next beacon” state.

MOVEMENT

PICKER STATE DESIGN

- In the “Go to next beacon” state the Picker robot checks if it is on the West or East side of an orchard row. Based on this it is able to figure out which beacon is next on its picking path. It then subscribes to this beacon and once it begins receiving the position of the beacon it will move to it. The Picker is in this state when it finishes picking a row and then needs to figure out which beacon to go to so it can enter the next row in its path.
- In the “Picking” state the Picker slows down its speed drastically so it is able to pick kiwifruit without missing any. While in this state, the Picker will be travelling through a row (between two beacons) and hence will be within picking range of kiwifruit pergola structures.
    - During this state the Picker will continuously poll to check if its kiwifruit bin is full (it does so each time it picks a kiwifruit tree). If it is full the Picker will switch its state to “Full bin”.
    - If the Picker reaches the end of a row (when it is at the position of the beacon at the end of that row) it will check to see if it has reached the end of its picking path. If it has not, then it will switch to the “Go to next beacon” state so it can find its next row. If it has reached the end it will switch to the “Finished” state.
- As soon as the Picker is in the “Full bin” state it will contact a Carrier robot to come and switch its full bin for an empty bin so it can continue picking fruit. It will remain stationary until this happens. Once the Picker has been notified by a Carrier that it is coming to help, the Picker will change its state to “Getting serviced”.
- During the “Getting serviced” state the Picker is still stationary. This state is required so other idle Carriers do not also come to the aid of this Picker (they can check to see if the Picker is getting serviced). Once the bin exchange has been made the Picker will switch back to the “Picking” state.
- Once in the “Finished” state the Picker will signal it has finished and will stop at the last beacon on its picking path.

PICKER ROBOT PICKING

- The picker robot picks kiwi fruit. This is done by using laser scanner on the picker robot to see if it go pass kiwi fruit trees. When it goes pass kiwi fruit trees it picks which will increase the amount of kiwi fruit picked in bin.

MOVEMENT

CARRIER ROBOT QUEUEING

- Initially all the carrier robots will queue up at the driveway waiting and the carrier robot at the top of queue will be in idle state waiting to be dispatch. As soon as the robot is dispatch, the rest of the carrier robot in the queue will move up and the next robot in the queue switch to idle state. The carrier will rejoin the queue after it has finish transporting the bin.

CARRIER STATE DESIGN

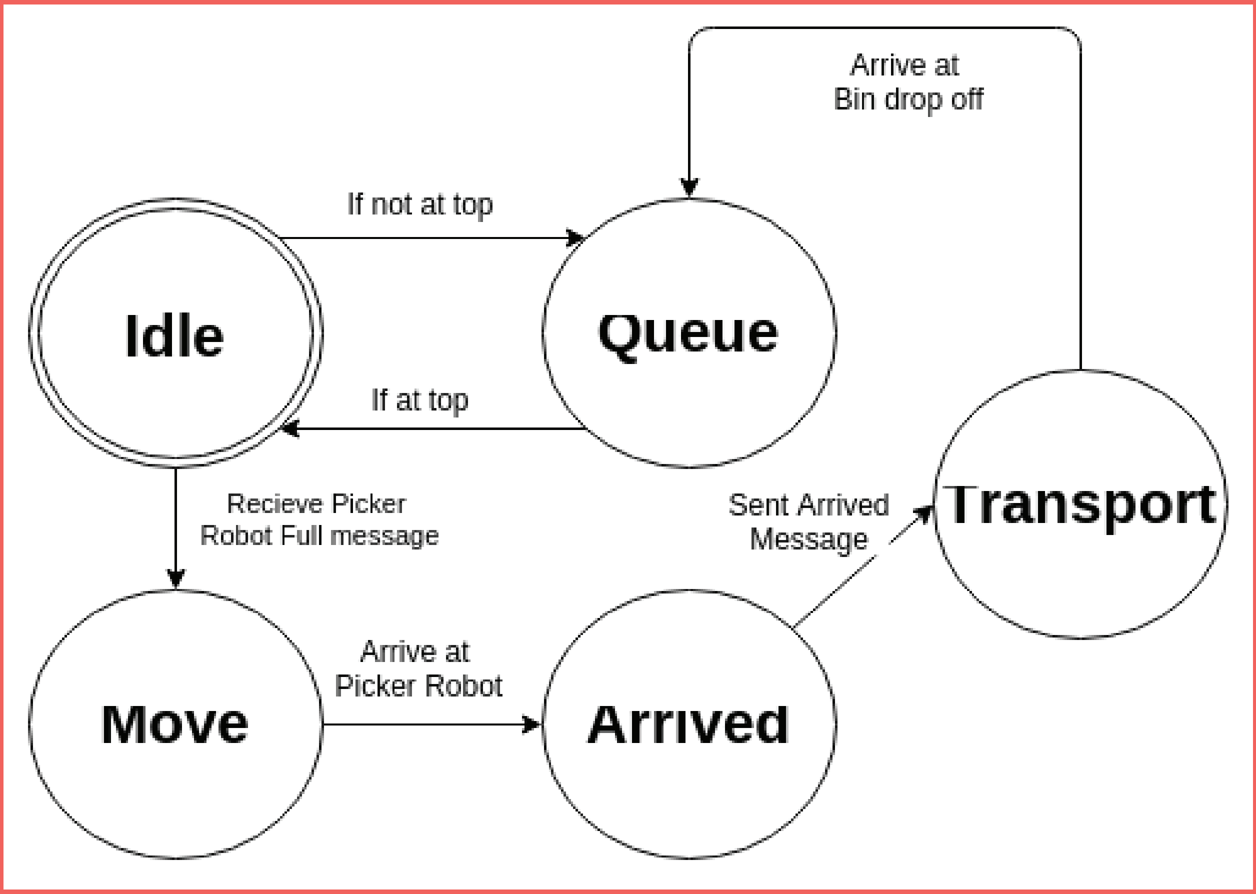


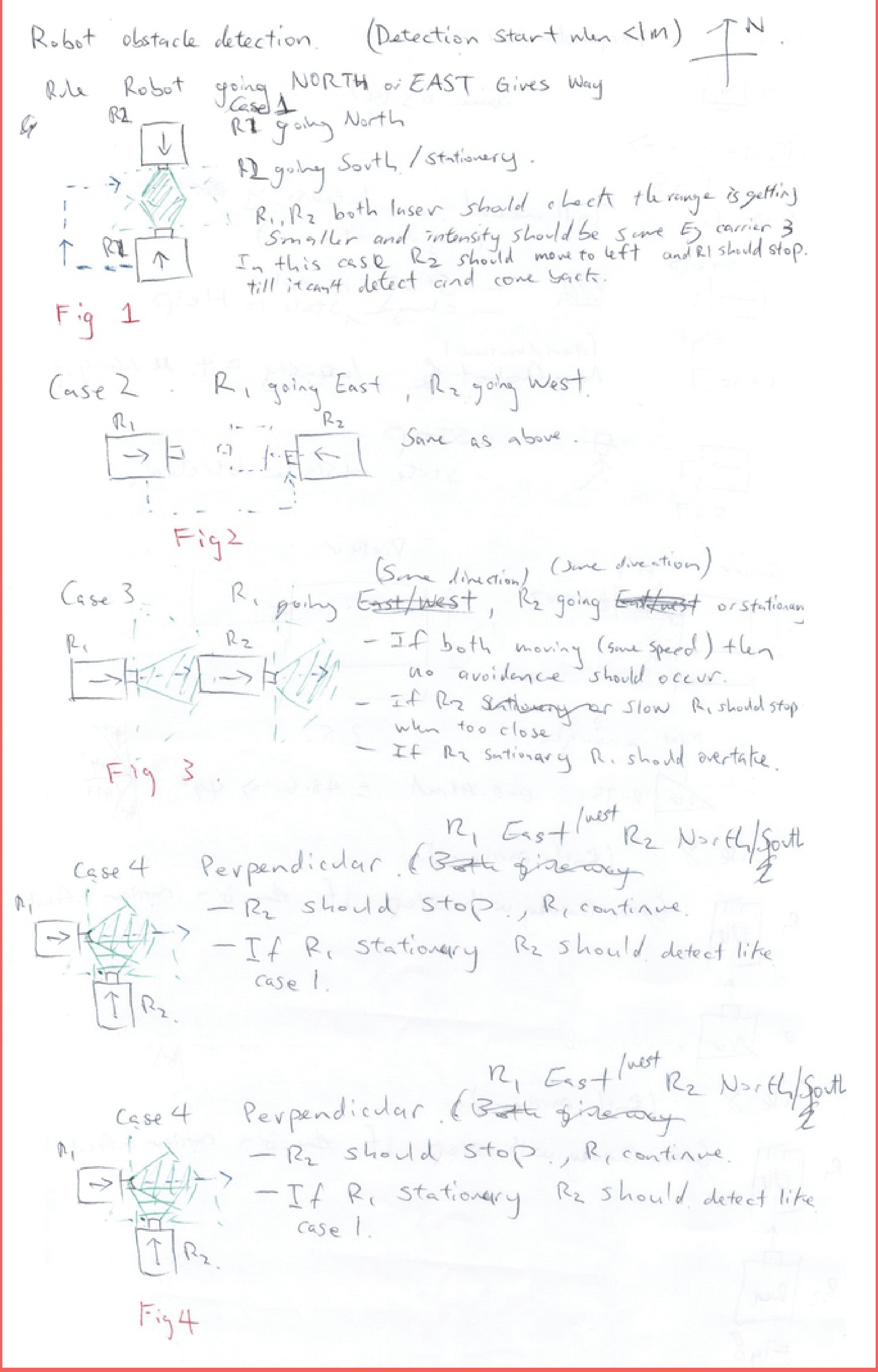
Figure - 3.0

OBSTACLE DETECTION & AVOIDANCE

LASER INTENSITY

- Tree.................................1
- Picker..............................2
- Carrier.............................3
- Human...............................4
- Animals.............................5
- Tall Weed...........................6
- Tractor.............................7
- Blind Person........................9

OBSTACLE DETECTION & AVOIDANCE



Robot obstacle detection. (Detection start when <1m)

Rule Robot going NORTH or EAST Gives Way

Case 1

R1 going North

R2 going South / stationery.

R1, R2 both laser should check the range is getting smaller and intensity should be same 5 carrier 3

In this case R2 should move to left and R1 should stop. till it can't detect and come back.

Fig 1

Case 2. R1 going East, R2 going West.

Same as above.

Fig2

Case 3 R1 going East/west (Same direction), R2 going ~~East/west~~ (Same direction) or stationary

- If both moving (same speed) then no avoidance should occur.
- If R2 ~~Stationery~~ or slow R1 should stop when too close
- If R2 Sationary R1 should overtake.

Fig 3.

Case 4 Perpendicular. (~~Both give way~~ R1 East/west R2 North/South

- R2 should stop., R1 continue.
- If R1 stationary R2 should detect like case 1.

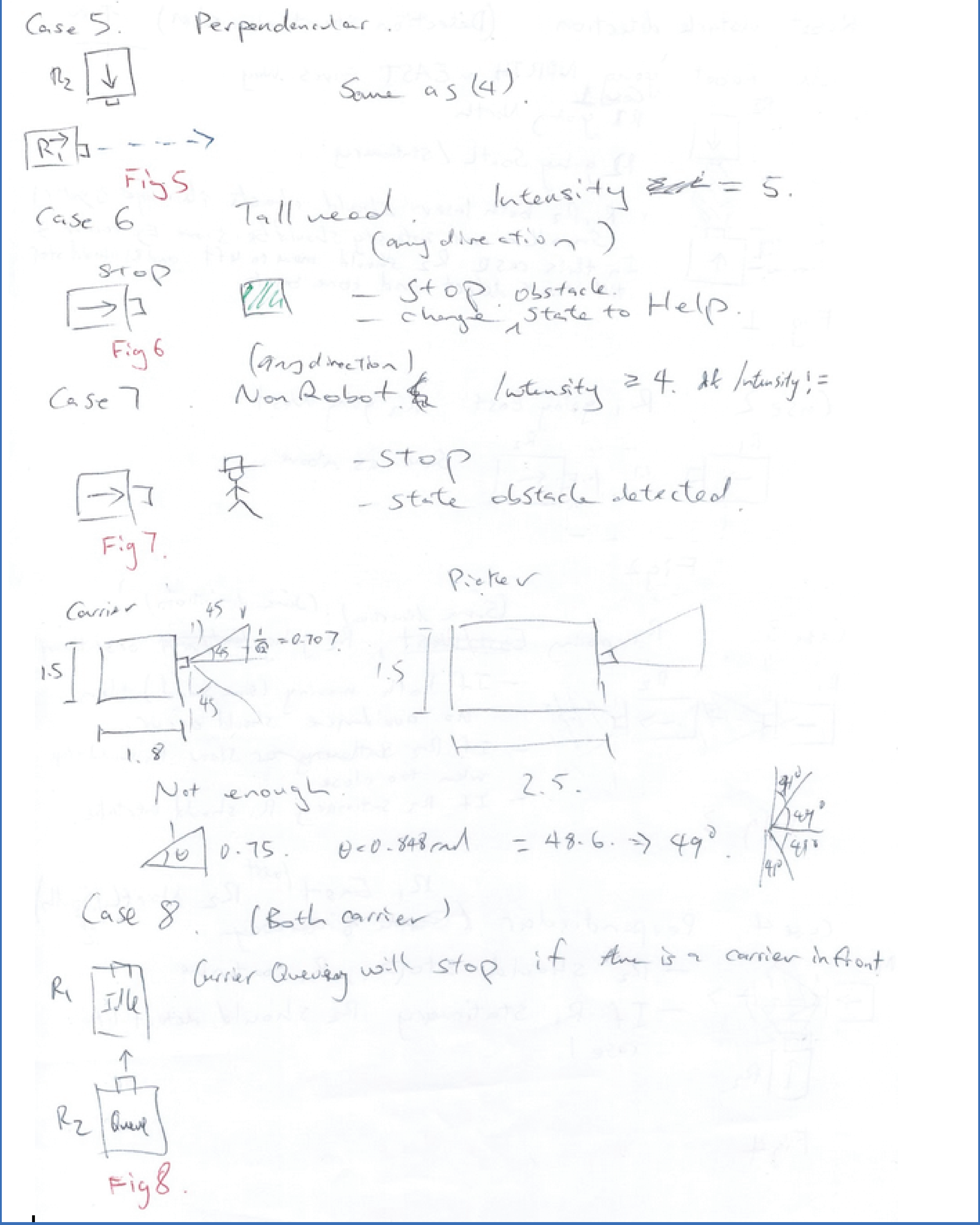
Case 4 Perpendicular. (~~Both give way~~ R1 East/west R2 North/South

- R2 should stop., R1 continue.
- If R1 stationary R2 should detect like case 1.

Fig4

Figure - 4.0

OBSTACLE DETECTION & AVOIDANCE



Case 5. Perpendencdar.

R2

Same as (4).

R1

Fig 5

Case 6 Tall weed. Intensity ~~≥k~~ = 5.

(any direction)

stop

– stop obstacle.

– change state to Help.

Fig 6

(any direction)

Case 7. Non Robot. Intensity ≥ 4. && Intensity !=

– stop

– state obstacle detected.

Fig 7.

Carrier

45

1.5

45

$\frac{1}{\sqrt{2}} = 0.707$

45

1.8

Picker

1.5

Not enough. 2.5.

θ 0.75. θ<0.848rad = 48.6. → 49°.

41°

49°

41°

41°

Case 8. (Both carrier)

Carrier Queuing will stop if there is a carrier in front

R1 Idle

R2 Queue

Fig 8.

Figure - 5.0

OBSTACLE DETECTION & AVOIDANCE

- Obstacle detection is done by using two laser scans to gather data which is used to determine the specific type of encounter. Two messages are used to check how the obstacles are moving.
  - NONE: If no object has been detected within the given range then there are no obstacles.
  - HALT: If objects have been detected then the entity must stop.
  - LIVING_OBJ: When the detected object is a living entity, such as a human(worker, garden worker, blind person or neighbour) or animal(cat or dog).
  - WEED: When the detected object is a tall weed.
  - TREE: When the detected object is a tree.
  - FACE_ON: When the detected object is a robot and it is moving towards the entity.
  - PERPENDICULAR: When the detected object is a robot and it is moving perpendicular to the entity.
  - STATIONARY: When the detected object is a robot and it is idle.
  - ROTATE: When the detected object is a robot and it is rotating.

CARRIER ROBOT DETECTIONS

- If in queue stops, also stops when its human, animal or weed. If it is transporting and facing north it will move backwards.

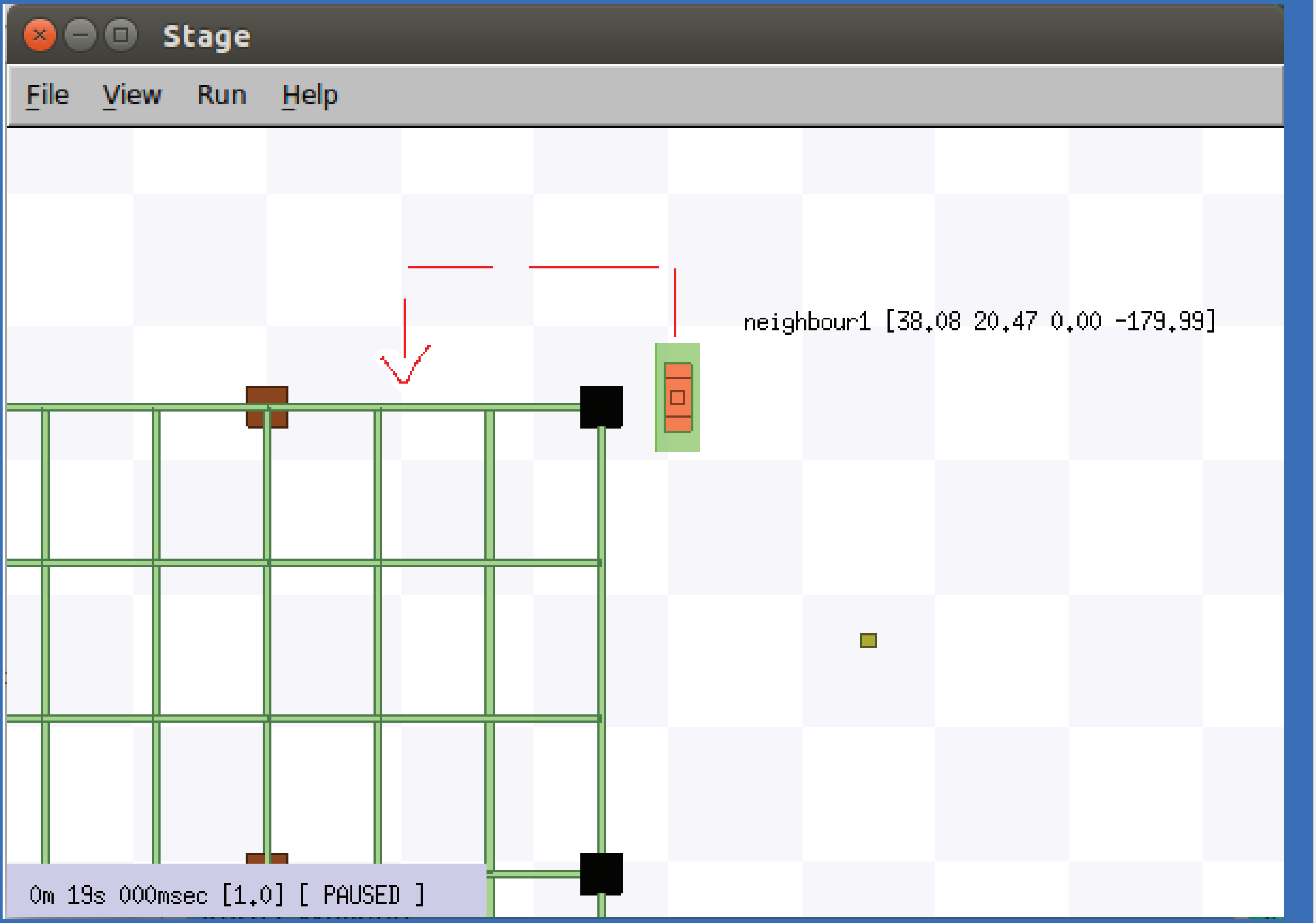
PICKER ROBOT DETECTIONS

- It stops when its human, animal or weed. It also stops if carrier robot is in front when they are picking.

ADDITIONAL DETECTIONS

- Neighbour robot will move in the opposite direction when it has detected a picker robot.

OBSTACLE DETECTION & AVOIDANCE



neighbour1 [38.08 20.47 0.00 -179.99]

0m 19s 000msec [1.0] [PAUSED]

Figure - 6.0

- All animals and humans, with the exception of blind persons, will avoid obstacles as shown in the figure above.

ANIMAL & PERSON BEHAVIOUR

- The class structure contains an Entity superclass which is subclassed by the Animal and Person class. Specific animals and people such as dogs, cats, workers, gardeners, and neighbours are subclassed from Animal or Person. This allows generic behaviour specific to all entities, people or animals and does not need to be repeated. The subclasses only contain behaviours that are specific to them.
- The workers and gardeners model the orchard workers, so they will move around the orchard regularly and trim the trees or remove weeds. The worker trims all the trees in a single column instead of trimming any tree in the orchard to reduce the complexity of the implementation. This allows us to spawn multiple workers and assign each worker with a column without having to worry about multiple workers trimming the same tree.
- The neighbour and blind person are not trained to be around robots so they may move around more randomly. The blind person may follow the dog, which models the use of a guide dog .
- Animals such as dogs and cats have specific behaviours such as moving in circles and lying close to trees, though they may get out of the path of other robots if it sees the robot.
- Garden workers are responsible for “pulling” the weed placed around the orchard during the launch file generation. Initially, the garden workers are idle in their spawn area. It is only until one of the carrier robot/picker robot has encountered a weed does the garden worker move to pull the weed. The implementation of garden workers closely follow a FSM. The states for the garden workers are Idle, Communicating, Moving, Pulling Weed, Done. Each state in the FSM represent the current situation the garden worker is in. All garden workers are initially Idle until a weed removal request is made (picker robot/carrier robot issues request when a weed obstacle is detected).
- Depending on the number of garden workers in the orchard, the garden worker will move to pull the weed or communicate with the other garden workers to determine the closest garden worker to the weed. If the garden workers enter the communicating state, all garden workers exchange messages between each other, each passing in their distance to the weed. Once all messages have been received, the closest robot to weed can be determined. Pulling weed involves a timer. Pulling weed was decided to take 3 seconds. When the garden worker has approached the weed, it will remain stationary for 3 seconds. After the 3 seconds have completed, it has finished its job and will return back to its original location.

GRAPHICAL USER INTERFACE

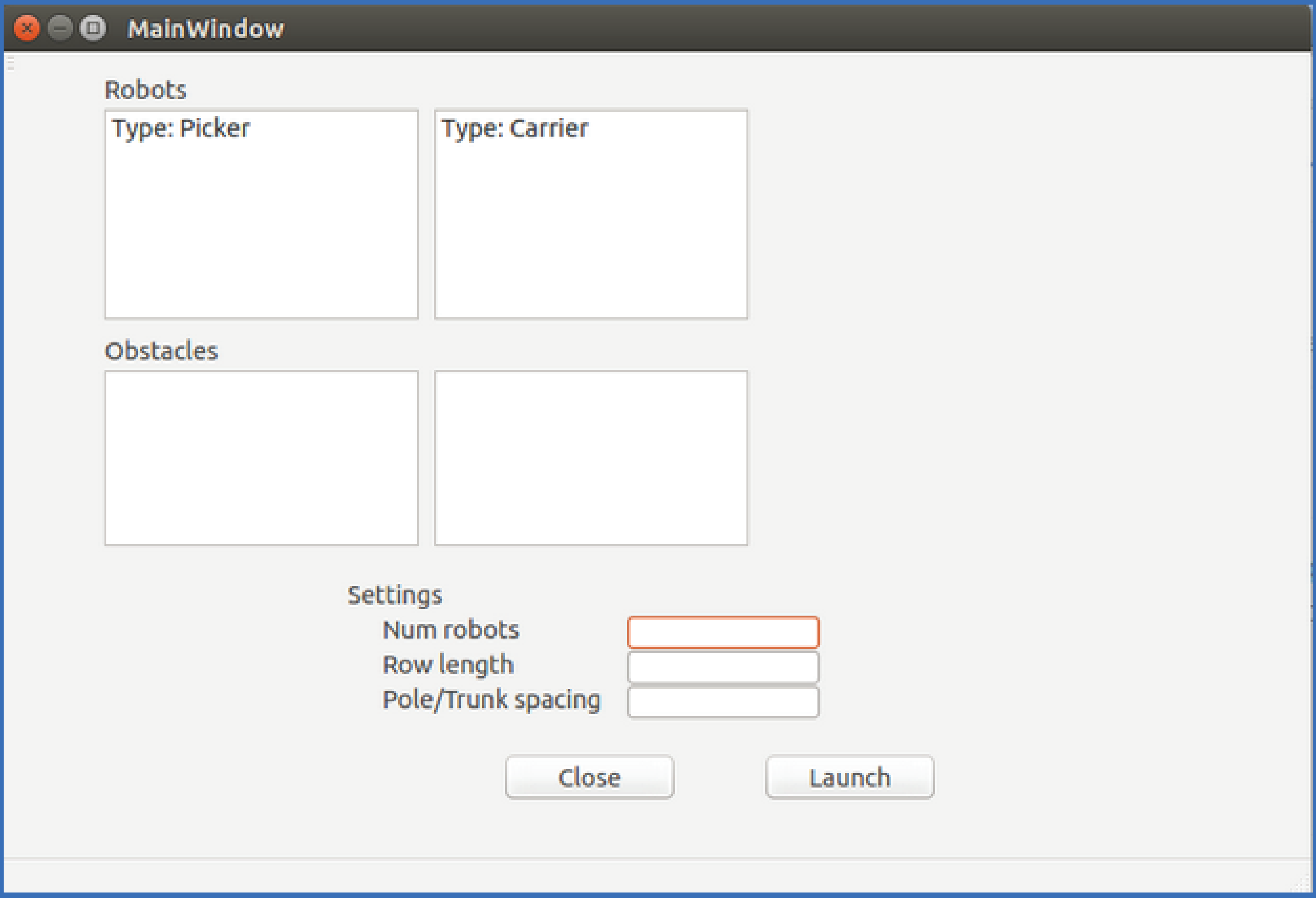
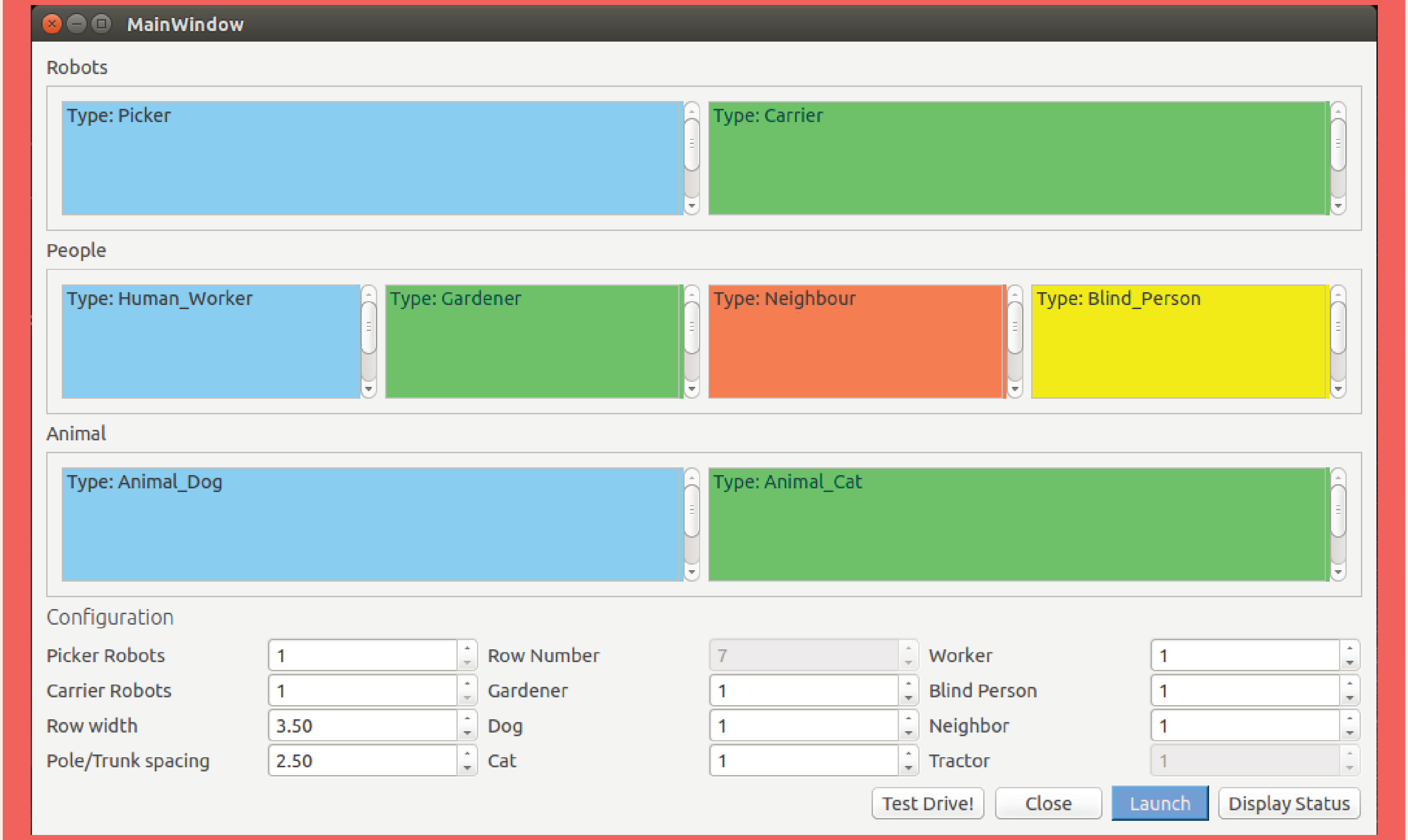


Figure - 7.0

- The role of the GUI in our program is to show the robot node statuses, allow configuration from user input and starting our ROS application. The biggest advantage of having GUI is usability because it replaces the need to run multiple bash commands to execute the program therefore user interface design was prioritized. The changes from the alpha version to the final version is noticeable, however, the key elements are similar. The list widgets show the statuses of each robot. The final version introduced coloured backgrounds which corresponds to the robots on the stage as this makes it easier to identify the robot for debugging purposes. The user input are received through spinners.

GRAPHICAL USER INTERFACE



Picker Robots	1	Row Number	7	Worker	1
Carrier Robots	1	Gardener	1	Blind Person	1
Row width	3.50	Dog	1	Neighbor	1
Pole/Trunk spacing	2.50	Cat	1	Tractor	1

Figure - 8.0

- The design includes a graphical user interface which is designed to show the status of the system in a clear way to the users. It uses the C++ QT4 library which is cross-platform and built into the lab image of Ubuntu.
- Each dynamic object shows its current position, velocity, and current state. This will update dynamically using background threads so that the interface can continue to be responsive.
- A colour key is used which links the background colour of the panels with the colour of the objects, so it is easy for the user to tell which panel refers to which object.

CONFIGURABLE ORCHARD

- Due to the specifications in the brief mentioning the possibility of configurable dimensions in the orchard, it was decided that a world and launch file generator would be used which would generate the orchard based on the values that the user enters into the user interface.

- The user interface can also be used to set the number of robots, animals, and people in the orchard, to allow the user to easily customise the simulation.

- Restrictions are placed on the number of robots that the user can create to represent a real world scenario, maintain performance and prevent overloading Stage.

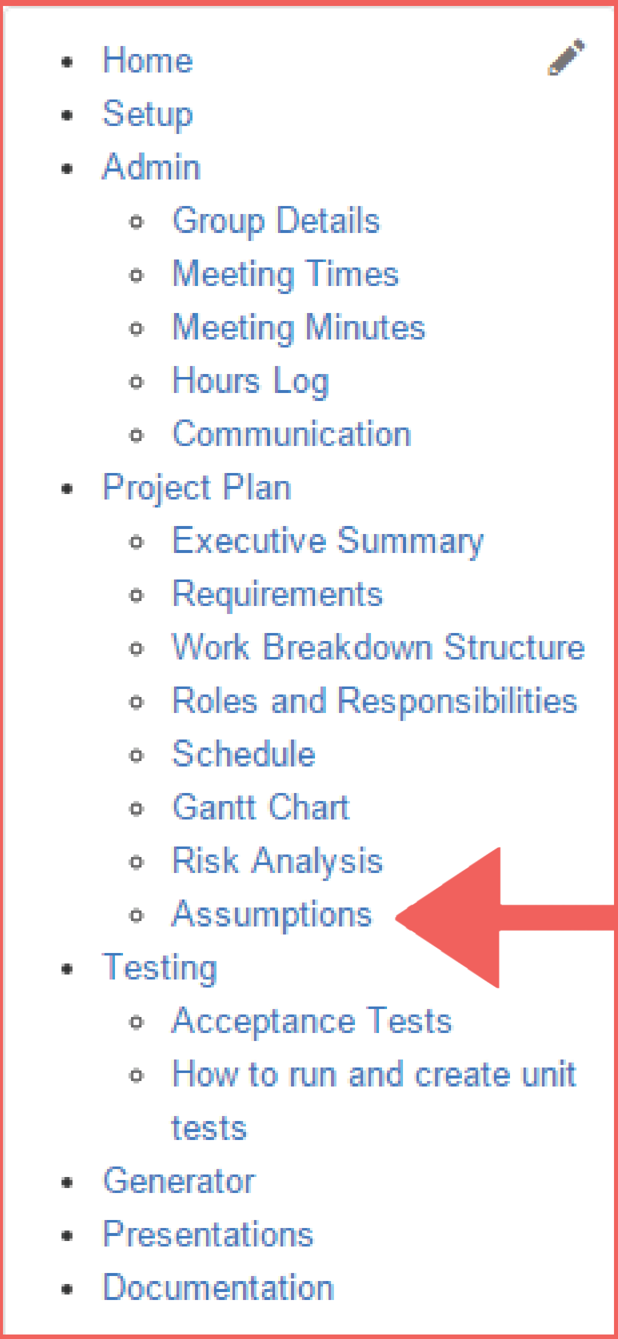
Picker	1 - 7
Carrier	1 - 7
Cat	0 - 3
Dog	0 - 5
Worker	0 - 4
Garden Worker	0 - 4
Blind Person	0 - 3
Neighbour	0 -3
Row Width	3 - 6m
Pole Spacing	1 - 10m

- The world components specified by the user are randomly generated inside the orchard. All components, other than the tall weeds, are generated inside the kiwi-fruit farming area, not the entire orchard. To ensure that these components do not spawn inside or too close to the kiwifruit trees they're spawn points are ensured to be equal distance apart from the trees surrounding them.

- The tall weeds spawn randomly around the orchard. However, they are not allowed to spawn on or near the driveway. While weeds can be removed on the simulation; this decision was made as weeds spawning on these areas hindered the simulation's speed unnecessarily.

- The trees that surround the perimeter of the orchard have their colours randomly picked from an array of colours.

DESIGN ASSUMPTIONS



- Home
- Setup
- Admin
    - Group Details
    - Meeting Times
    - Meeting Minutes
    - Hours Log
    - Communication
- Project Plan
    - Executive Summary
    - Requirements
    - Work Breakdown Structure
    - Roles and Responsibilities
    - Schedule
    - Gantt Chart
    - Risk Analysis
    - Assumptions
- Testing
    - Acceptance Tests
    - How to run and create unit tests
- Generator
- Presentations
- Documentation

Figure - 9.0

- **The project's design assumptions can be found under the "Assumptions" category in the GitHub Wiki.**

CONSTRAINTS

ENVIRONMENT

- The project had to be completed in the Linux operating environment. This is due to the fact that ROS is only compatible with the Linux operating system. Therefore, any programs or libraries used in the development of the project also had to be compatible with Linux.

TECHNICAL

- The number of world components that could be added to the simulation had to be limited. This was due to both performance and overloading issues. If too many components were added the simulation would slow down considerably; additionally, Stage would eventually overload.

- Stage components cannot be moved in the z direction dynamically. Initially, it was planned to remove weeds by moving them underground to remove them from view. However, due to this constraint the weeds had to be moved to the nearest tree instead.

- When idle components are spawned in Stage. ROS occasionally renders their laser intensities incorrectly so that it the object did not render. Components in the simulation use these laser intensities to determine various scenarios. One such scenario is when a robot uses its laser to determine whether there is an obstacle in front of it. Due to this bug the robot will behave differently, such as moving at the wrong speed as the idle obstacle component will fluctuate between transparent (not detectable) and solidified (detectable).

DESIGN PRINCIPLES

Flexibility........................20

Maintainability/ Extensibility........................20

Robustness........................21

Usability........................21

DESIGN PRINCIPLES

FLEXIBILITY

- The design is flexible as it allows for further people, robots, or animals to be added without needing to modify existing entity classes, and duplicating movement code wouldn't be necessary since these are included in parent classes.

- Behaviours are kept separate from low level movement details, so changing the behaviour of one entity will be completely separate from other entities

- The use of callbacks and publisher/subscribers means that specific dependencies between entities are not needed, for example if any other robot also wants to subscribe to the picker robot, they can simply subscribe to the topic.

MAINTAINABILITY/ EXTENSIBILITY

- Movement queue allows for easy addition of new behaviours.

- Generator class means that new entities can be generated automatically into the world file and launch file.

- Use of beacons to guide robots means that further parameters such as row length and number of rows could be added if time allowed.

- GUI showing the status is good for maintenance as it makes the system state obvious.

DESIGN PRINCIPLES

ROBUSTNESS

- The use of spinners in the GUI ensure that no illegal arguments are passed, as these allow for maximum and minimum values to be set. Additionally, non-numeric values cannot be passed.

USABILITY

- The GUI ensures that configuring the simulation is done quickly & easily. There is a label for each component of the simulation that can be configured and a spinner to adjust the value for each component. Additionally, there are buttons to launch & close the simulation as well as starting the innovative feature, controlling the tractor. Furthermore, the GUI easily allows tracking of each world component as their colour on the GUI corresponds to the component's colour in the world. These panels also allow the user to easily track the component's position, orientation & status.

DESIGN DECISIONS

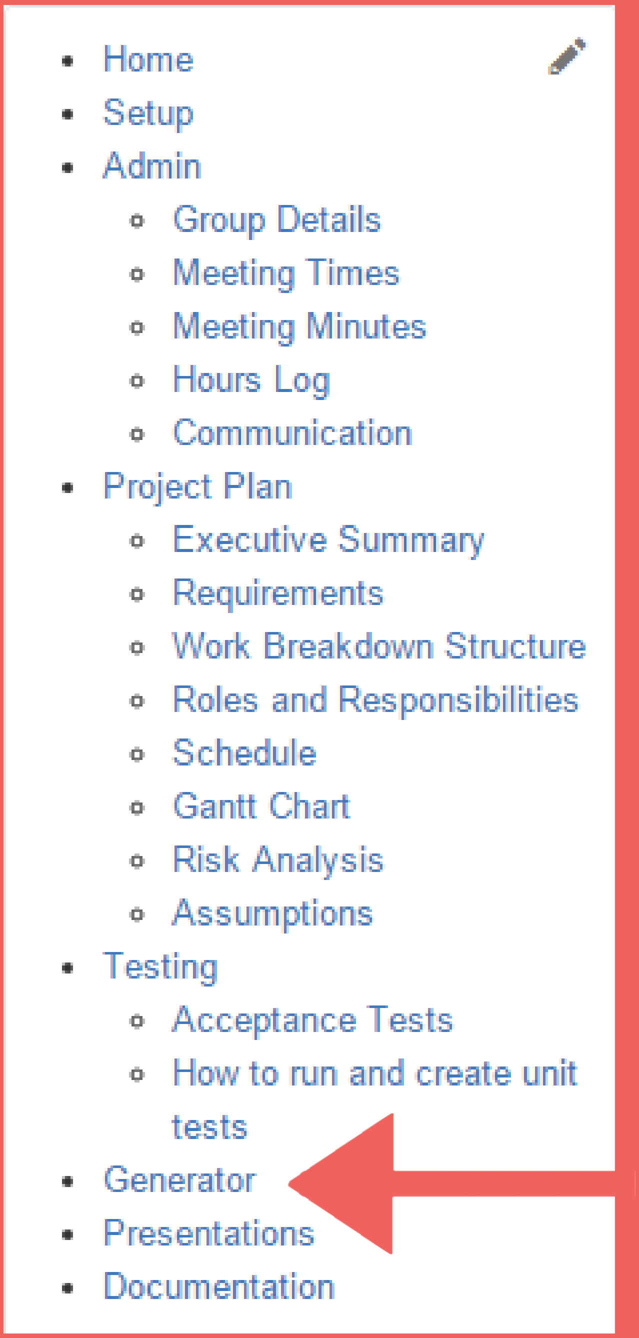
PATTERNS USED

- The design uses a publisher/subscriber pattern as this is how ROS was designed to work, and it mimics the behaviour of swarm bots. There is no “master” node and instead the robots communicate with others through publishing and subscribing to topics.

C++ vs Python

- C++ was chosen instead of python because it is more widely used in the robotics industry so it was thought that using C++ would be a more valuable experience. It also has the QT4 gui library built in to the lab computers, which includes the ability to use background threads to process information and sends signals to update the user interface, which is what was required for the status panels.

GENERATOR



- Home
- Setup
- Admin
  - Group Details
  - Meeting Times
  - Meeting Minutes
  - Hours Log
  - Communication
- Project Plan
  - Executive Summary
  - Requirements
  - Work Breakdown Structure
  - Roles and Responsibilities
  - Schedule
  - Gantt Chart
  - Risk Analysis
  - Assumptions
- Testing
  - Acceptance Tests
  - How to run and create unit tests
- Generator
- Presentations
- Documentation

Figure - 10.0

- The documentation on the “Generator” class can be found under the “Generator” category in the GitHub Wiki.