APPENDIX

This appendix only presents the information that is allowed to be shared. Confidential and personal information is not shared here.

MSc. Thesis Report
Stijn Gruben

APPENDIX A: SLAM VISUALIZER DEVLOG

Appendix A will show the documented robot exploration journey towards creating the different interfaces for this project.

Platform: Ubuntu 22.04, ROS2 Humble, Python 3.10, PyQT5, PyQTGraph

Goal: Visualize real-time SLAM data (map, robot, and LiDAR) in a custom GUI environment (like RViz2).

Motivation

The robot's SLAM data could already be visualized in RViz2, but as the end goal of the project is a more intuitive, user-friendly interface, I wanted to see if I could create my own customizable GUI

Goal

Create a custom interface with customizable layout, style, and interaction (e.g., toggling layers, adding extra info panels).
Be able to subscribe to ROS nodes and display the sensor data in real-time

Design Decision #1 – Software Choice

The first decision I needed to make was which platform to use for creating the GUI. Some options I looked into:

Option	Pros	Cons
RViz plugins	Integrates easily with ROS2	Limited styling, C++ heavy
Web UI (rosbridge + JS)	Modern look, remote access	Needs web server, more plumbing
Desktop GUI (PyQt5)	Full control, fast prototyping, Python-native	Local only, manual layout

Choice: PyQt5, because:

- I am already using Python for ROS 2.
- PyQt5 efficiently handles real-time 2D graphics (map, scan points).
- Needs the least initial setup for this first test, I can always change my preferred method later.

Phase 1: Basic Setup (8 Oct)

Start SLAM on Mirte

First step was to make sure we have a reliable connection between my laptop and the Mirte robot. I need to be able to read the ROS topics from my laptop and launch the minimal SLAM code to gain access to all necessary ROS topics. There are two ways to connect to the mirte: using an Ethernet cable (mostly used while writing and editing code) and wireless (used when testing "for real").

Tutorial to connect to Mirte and start SLAM mapping using RViz2

1. Connect Wi-Fi to Mirte-67AFA6 hotspot, OR connect the laptop and robot using an Ethernet cable
2. In VSCode: >remote SSH: Connect to host (For Wireless-> IP: 192.168.42.1) OR (For cable-> IP: 10.42.0.59)
   Password: dev_dev
3. Open Linux terminal (CTRL+SHIFT+~)
4. In Terminal: check if connected to Mirte by using ros2 topic list
   If yes -> move on
   If no -> in VSCode (robot): sudo service mirte-ros restart (alias=restart)
   If still no try on Linux: ros2 daemon stop
5. In VSCode, start SLAM: ros2 launch mirte_navigation minimal_slam_launch.py
6. In Terminal, open RViz2: rviz2
7. In RViz2, add topics:
   /map
   /scan
8. Create your map by riding around in VSCode (new terminal): ros2 launch mirte_teleop teleop_key.launch.py
9. In VSCode, save map: ros2 run nav2_map_server map_saver_cli -f /home/mirte/mirte_ws/src/mirte_navigation/maps/default
10. In VSCode, close SLAM: CTRL+c
11. Before turning off the robot, always use: sudo shutdown now (alias=shutdown)

Basic setup

Now we are able to connect to the Mirte and access the needed ROS topics it's time to see if we can create our own GUI. After some initial troubleshooting, it became clear what main libraries I was going to need.

- rclpy The ROS Client Library for Python, to interact with ros2 using python
- PyQt5and pyqtgraph to create the GUI

Phase 2: Opening up the GUI (13 Oct)

When first starting up a code that should have opened up a window showing me a GUI visualizing a coordinate system with the necessary SLAM topics, I immediately had some troubles. While I didn't get any errors, there was also no GUI window popping up. So I first needed to make sure that I could get `PyQt5` to work before integrating ROS.

Create minimal PyQt5 GUI to see if it works

1. Started from minimal `PyQt5` code:

```
import sys
from PyQt5 import QtWidgets

# Create the Qt Application
app = QtWidgets.QApplication(sys.argv)

# Create a main window
window = QtWidgets.QMainWindow()
window.setWindowTitle("Basic PyQt5 Window")
window.resize(400, 300)
window.show()

# Start the Qt event loop
sys.exit(app.exec_())
```

2. Ran the code and confirmed that a blank GUI window appeared.
3. This confirmed that PyQt5 and Qt bindings were working correctly in the environment.

The initial issue was a mismatch between Qt's platform plugin expectations (Wayland vs X11/XWayland) on my system, which prevented the GUI from opening properly. Now that this issue was eliminated, I could add ROS back into my GUI.

Phase 3: ROS2 Integration GUI (14 Oct)

Goal: Create a basic ROS2 node using `rclpy` that opens a `PyQtGraph` plot and subscribes to SLAM-related topics.

Implementation Steps:

1. Created a ROS2 node class `SlamVisualizer(Node)`.
2. Initialized the GUI using:

```
self.app = QtWidgets.QApplication(sys.argv)
self.window = QtWidgets.QMainWindow()
self.view = pg.GraphicsLayoutWidget()
self.plot = self.view.addPlot()
```

3. Added basic PyQtGraph elements:
   `ScatterPlotItem` for LiDAR points (`blue`)
   `ScatterPlotItem` for robot position (`red`)
4. Subscribed to:
   `/scan` (LaserScan)
   `/odom` (Odometry)
   `/map` (OccupancyGrid)
5. Used a `QTimer` running at **20 Hz** to continuously update the GUI with new data.
6. Run code by using: `python3 ~/slam_viz/custom_slam_viz.py`

Result:
Using this code, the GUI launched without errors and displayed a coordinate grid, but no sensor data appeared.

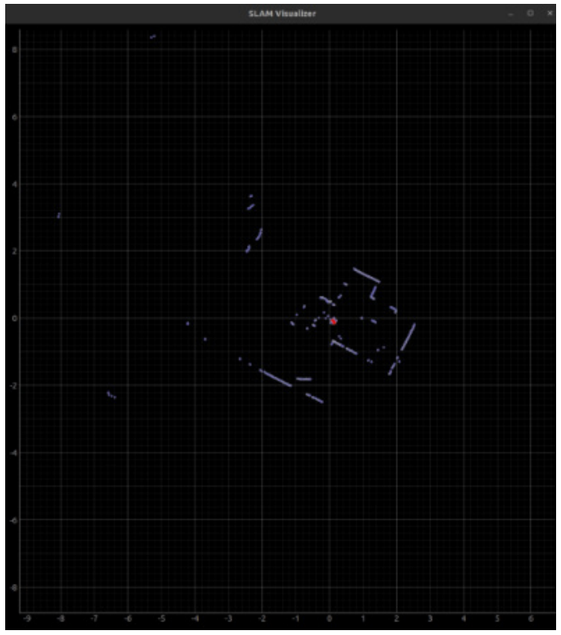
Phase 4: Debugging Missing Data (14 Oct)

Problem: ROS topics were not being visualized, no LiDAR or map updates appeared.

Troubleshooting Steps:

1. Verified active ROS topics: `ros2 topic list`
2. Check specific ROS messages: `ros2 topic echo /scan`
3. Discovered that the Python node wasn't sourcing the workspace where the robot's ROS packages were built, and SLAM needed to be launched for the topics to display properly
4. Fixed by launching minimal SLAM and running: `source /opt/ros/humble/setup.bash`

Result:
LiDAR points began appearing in the GUI, the first visual confirmation that the ROS node and PyQt5 GUI were integrated correctly.



Phase 5: Displaying the Map (15 Oct)

Goal: Render the /map data as in the GUI to represent free, occupied, and unknown cells.

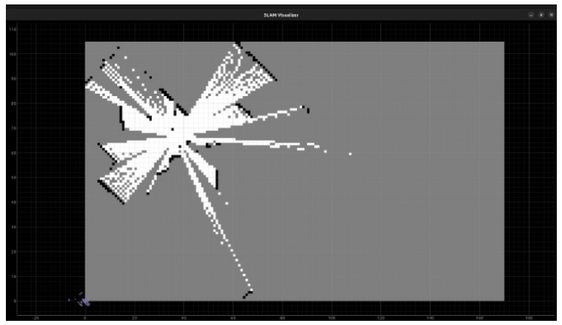
Steps:

1. Converted flat occupancy grid into a 2D numpy array:

data = np.array(msg.data, dtype=np.int8).reshape((msg.info.height, msg.info.width))

2. Translated occupancy values to grayscale:
   - `0` -> White (free)
   - `100` -> Black (occupied)
   - `1` -> Gray (unknown)
3. Displayed map with PyQtGraph's `ImageItem`.
4. Scaled and positioned map using `msg.info.resolution` and `msg.info.origin`.

Result:
As can be seen, a map image has appeared in the GUI; however, it is incorrectly scaled and when zoomed in, the sensor points don't align and are rotated in relation to the map data.

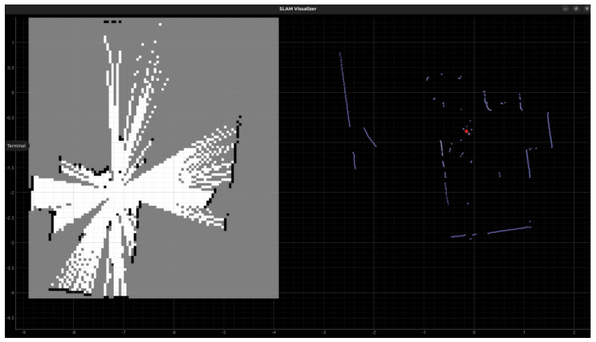


Phase 6: Align map and sensor data

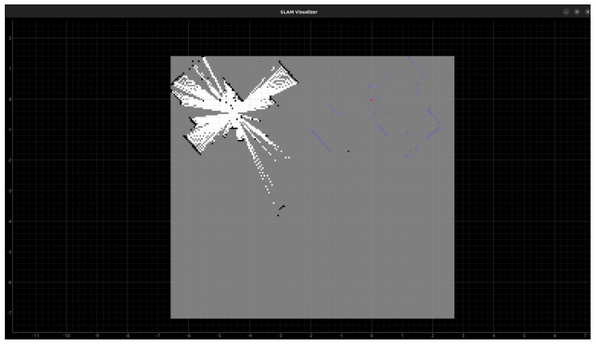
Goal: Align the map and sensor data to get a representation just like rviz has.

Steps:

1. Fix the current scaling issue
   - Currently, I scaled the map using: `self.map_img.scale(self.map_resolution, self.map_resolution)` however I got `TypeError: scale(self): too many arguments`
   - The solution for this is to instead use `setScale()`, which properly handles both X and Y scaling using one number (the `map_resolution`).
   - This gave me the following result: the map and sensor data are correctly scaled, but don't properly align.



2. Initial align approach: manual rotation and translation
   - At first, I tried to align the LIDAR by manually rotating points and shifting the GUI
   - Problem: This solution does not work, as it only applies to some situations, and the real map and LIDAR data are not correctly linked because they use different coordinate systems. The solution for this is hidden in the ROS library called `TF2`, which has the function to manage the relationships between different coordinate frames, allowing us to transform points, vectors, and other data between them over time.



3. Switching from manual to TF2 approach
   - Implemented TF2 into the code to fix the transformation:

```
from tf2_ros import Buffer, TransformListener
self.tf_buffer = Buffer()
self.tf_listener = TransformListener(self.tf_buffer, self)

transform = self.tf_buffer.lookup_transform('map', 'base_link', rclpy.time.Time())
```

- This way, I could get the robot's real position and rotation using the TF tree
- Problem: Map and LIDAR now appeared in the same area, but still didn't align; it seemed like the LIDAR was incorrectly rotated.

4. Fixing the TF tree
   - To check the different relations of the TR tree, we used `ros2 run tf2_tools view_frames`
   - The confirmed output was: `map -> odom -> base_link -> lidar_base -> lidar_link`
   - After using this TR tree, we could find the correct transforms between the map, robot, and LIDAR data.
5. Apply correct rotation using yaw
   - Using the TF, I could extract the yaw (2D rotation) and apply it, so the LIDAR data is now correctly aligned with the map data

```
#extract yaw
yaw = tf_transformations.euler_from_quaternion([rot.x, rot.y, rot.z, rot.w])[2]

#Apply 2D rotation
R = np.array([
    [np.cos(yaw), -np.sin(yaw)],
    [np.sin(yaw),  np.cos(yaw)]
])
points_world = R @ points_lidar.T + np.array([[tx], [ty]])
```

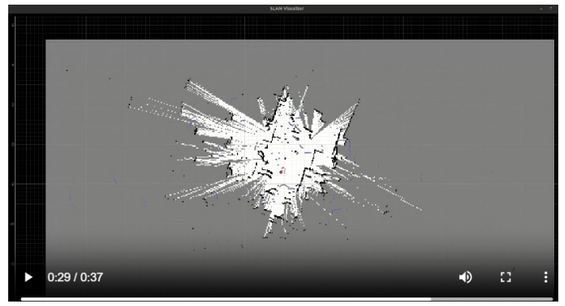
This fixed the rotation as can be seen however I was rotating every LIDAR point one by one which froze the GUI.
6. Clean up the code to fix lag issues
   - To stop rotating every point one by one I used vectorized transformation which helped make the GUI more smooth:

```
ranges = np.array(msg.ranges)
angles = np.linspace(msg.angle_min, msg.angle_max, len(ranges))

# Convert to Cartesian
xs = ranges * np.cos(angles)
ys = ranges * np.sin(angles)
points_lidar = np.vstack((xs, ys))

# Transform all points at once
points_world = R @ points_lidar + np.array([[tx], [ty]])
```

Also, to further simplify the GUI code, I cleaned up the update_plot() to no longer calculate rotations manually but have everything come from TF2:

```
def update_plot(self):
    transform = self.tf_buffer.lookup_transform('map', 'base_link',
    tx = transform.transform.translation.x
    ty = transform.transform.translation.y
    yaw = tf_transformations.euler_from_quaternion(
        [rot.x, rot.y, rot.z, rot.w]
    )[2]

    self.robot_pose = (tx, ty, yaw)
    self.update_lidar_plot()
```

7. Final outcome (for now)
   - Below is a picture showing how the GUI currently looks when mapping begins. At the start, the map and LIDAR data align well. However, once the robot starts moving, the map struggles to keep up with the other data.



The final code for this exploration is called First_CustomSlamViz.py and can be found along with all final code used for this project in the github: https://github.com/stijngruben/MscGraduationReport.git

APPENDIX B: WORKING SLAM VISUALIZER

Custom SLAM GUI using only native QT

Initial Context and Motivation

Motivation

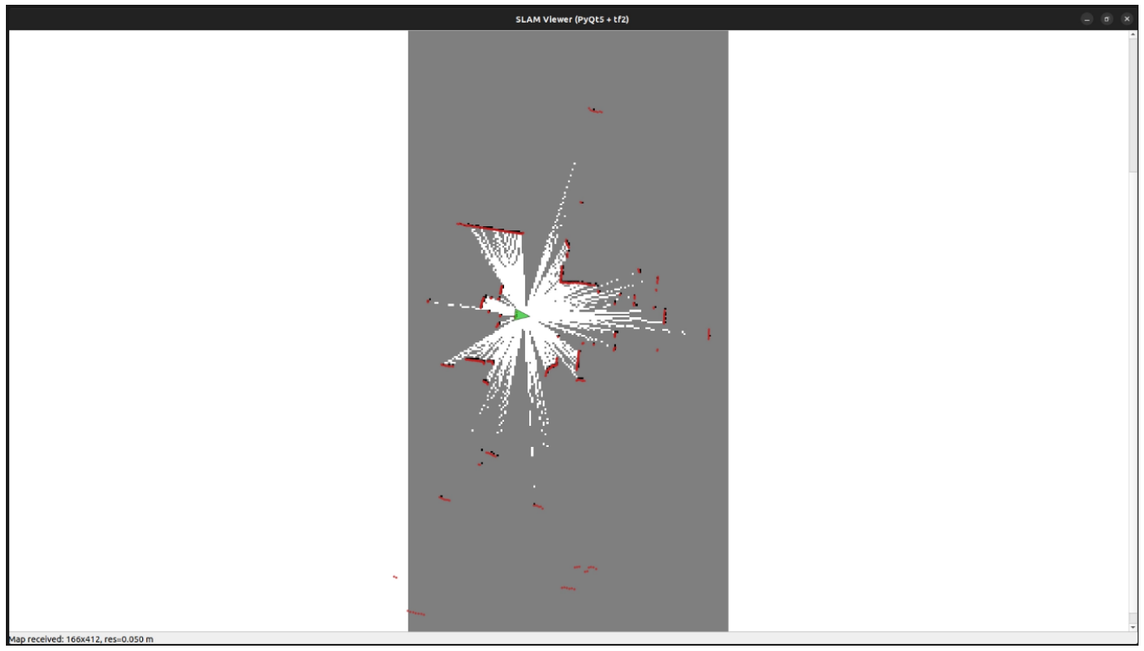
My previous GUI used PyQtGraph to visualize ROS data. It worked fine at first, but my supervisor and I noticed some issues; it seemed to interfere with the ROS nodes and have issues with the speed it was displaying sensor and map data. Furthermore, adding new features later on would be trickier with having PyQTGraph involved, so we decided it would be better to take out this part and try to rebuild the code without it. Rebuilding it with native Qt makes things cleaner, more stable, and easier to expand.

Goal

- Create a custom GUI that can handle the ROS data well and display it in real-time
- Only use native QT for this visualisation

Phase 1: Rebuilding from scratch (12 nov)

I found out that the issue was qt uses other coordinate system than ros (mirrored). So a proper working system was built quite quickly due to the extensive exploration done before and the right base we already made.



The final code called: WorkingSlamViz.py can be found in the github:
https://github.com/stijngruben/MscGraduationReport.git

APPENDIX C: WORKING METHOD TUTORIAL

Current work method for mapping, saving, cleaning the map and navigating with the newly clean map.

1. Map using SLAM on the LINUX pc.

Step 1: Make sure the terminal + VS Code (robot) are properly aligned using `ros2 topic list`. (if not use `restart` to restart the robot)

Step 2: Launch SLAM on the robot: `ros2 launch mirte_navigation minimal_slam_launch.py`

Step 3: Launch personal SLAM visualisation `python3 ~/slam_viz/workingsave.py` `(/bin/python3 /home/robohouse/slam_viz/workingmapsave.py`)

Step 4: Map the area using: `ros2 launch mirte_teleop teleop_key.launch.py`

Step 5: Once correctly mapped, save the map in the interface (you get a `.png` and a `.yaml`) to save you have to input a file name and add .png at the end

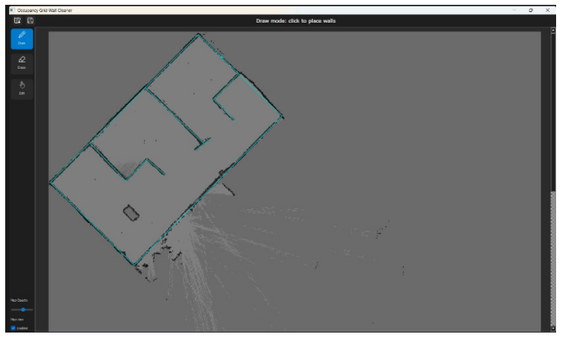
2. Clean the map using the drawing interface

Step 1: Launch the drawing GUI python file.

Step 2: Load the map (`.png` & `.yaml`) that you just created using SLAM

Step 3: Create your clean map on top of the messy SLAM map using the drawing interface

Step 4: Save the clean map as a `.pgm` & `.yaml`



3. Use the new map for navigating

Step 1: Put the clean map (.pgm & .yaml) in the `/home/mirte/mirte_ws/src/mirte_navigation/maps` directory

Step 2: open the file `~/mirte_ws/src/mirte_navigation/params/minimal_nav2_params.yaml` and change the map name in the line
`"/home/mirte/mirte_ws/src/mirte_navigation/maps/default.yaml"` of `default.yaml` into your `.yaml` file

Step 3: Make sure the .yaml file is referencing the correct map file (so `.pgm` and not `.png`)

Step 4: Start the navigation using `ros2 launch mirte_navigation minimal_navigation_launch.py`

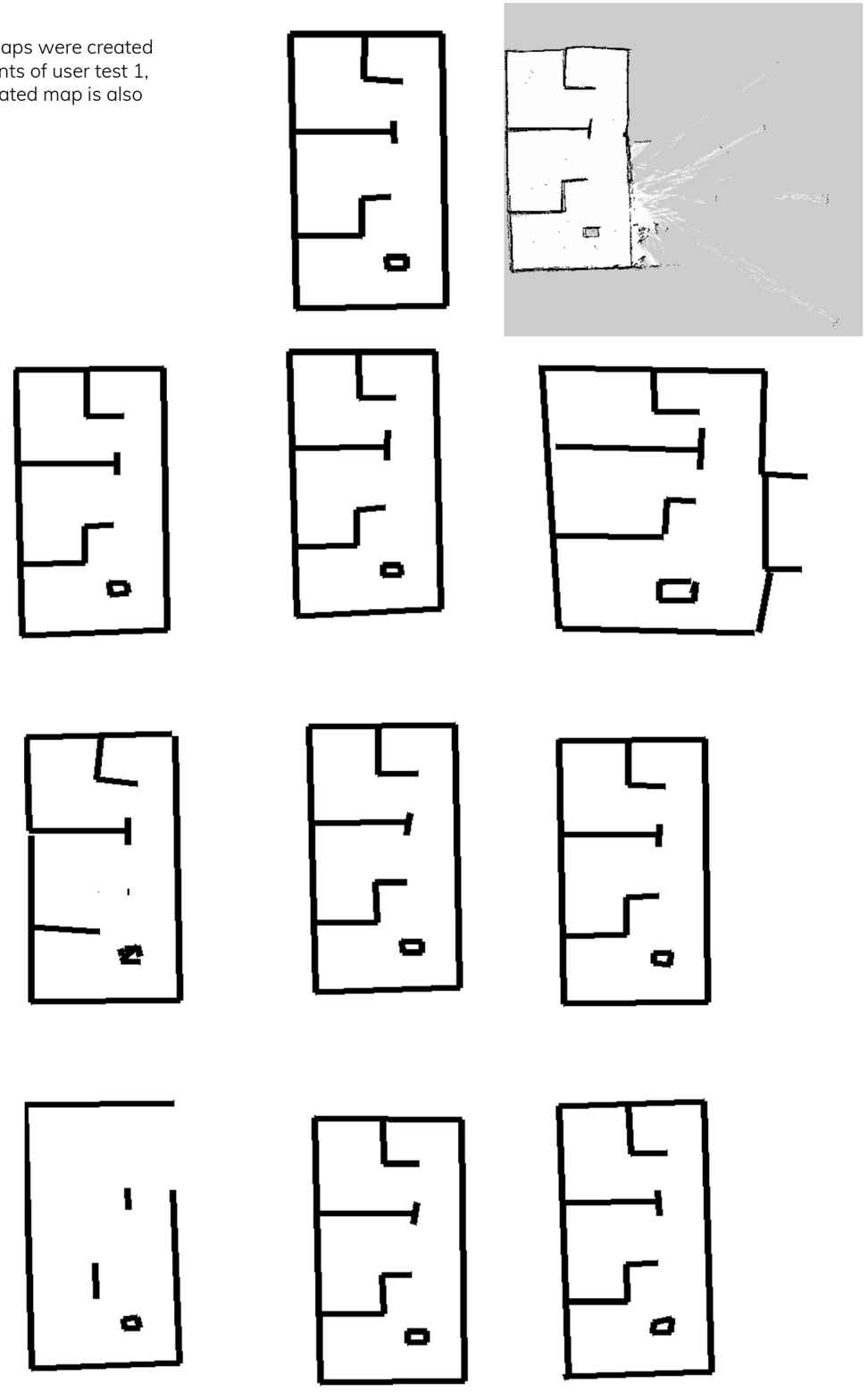
Step 5: Open RViz in the terminal using `rviz2`

Step 6: If `/map` is not showing up, you have to go to topic and set durability policy to transient local.

The final interface code can be found on the github:
https://github.com/stijngruben/MscGraduationReport.git

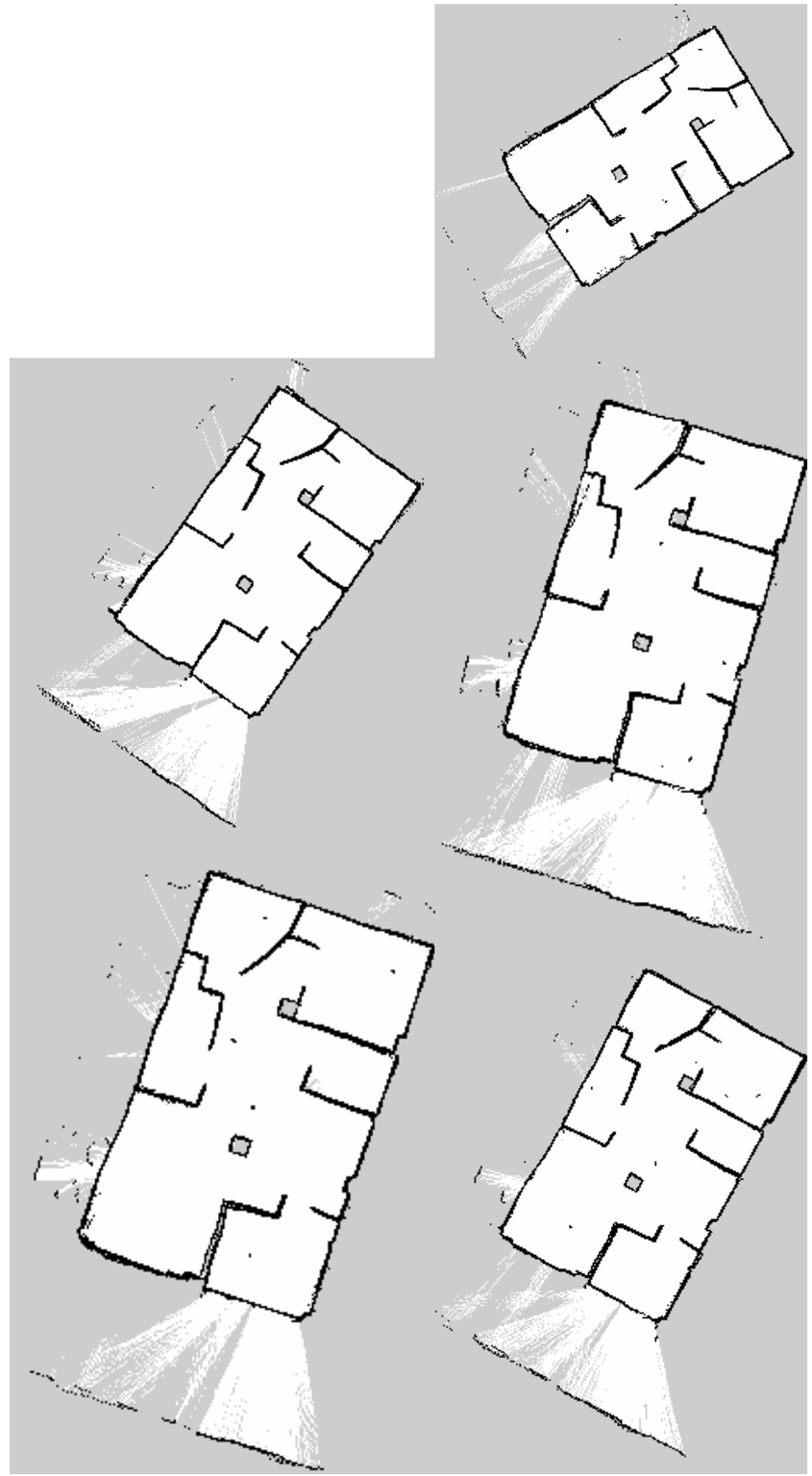
APPENDIX D: USER TEST 1 RESULTS

The following maps were created by the participants of user test 1, the robot-generated map is also shown.



APPENDIX E: USER TEST 2 ROBOT MAPS

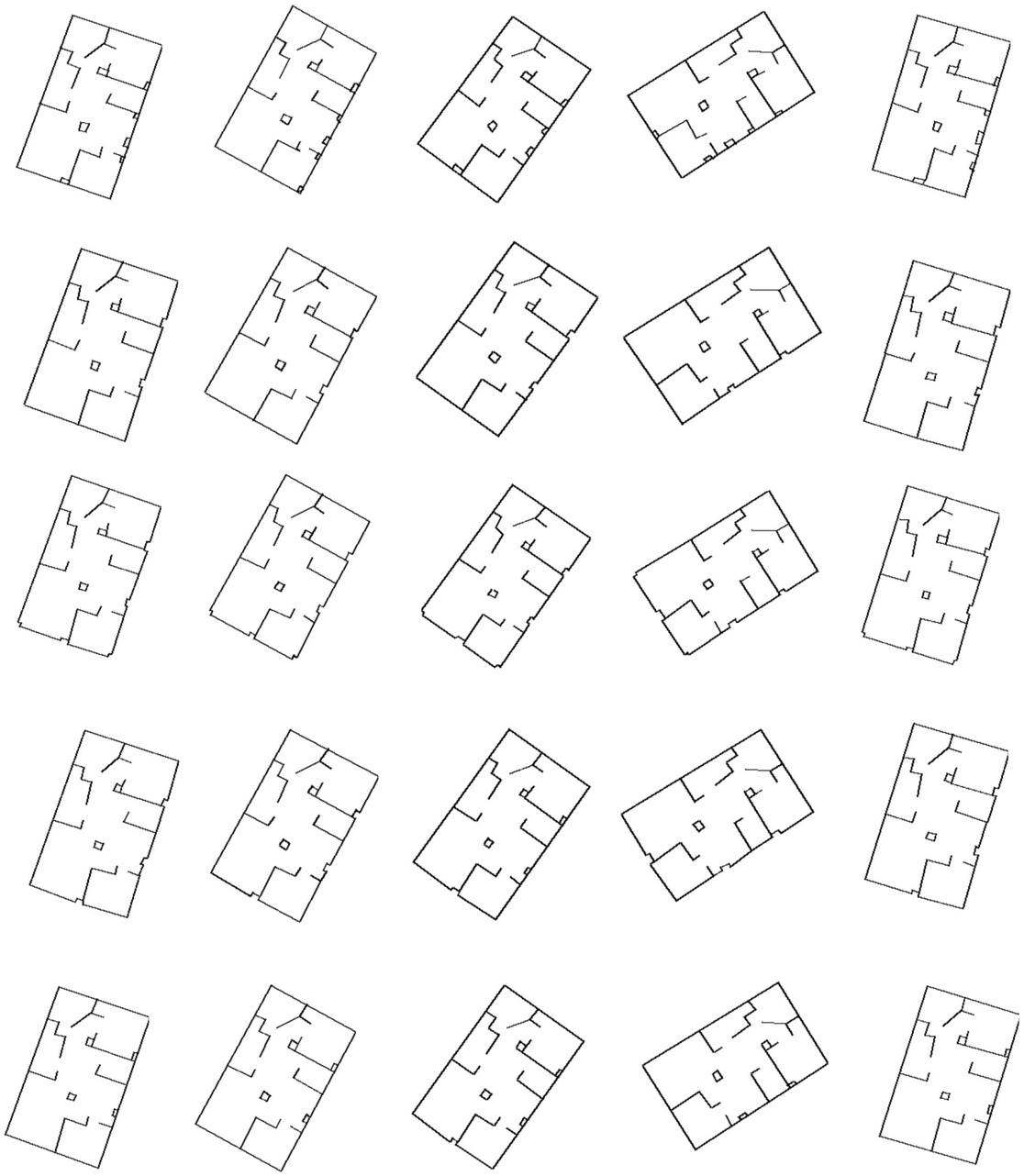
The five robot-generated maps used in the second user test are displayed here.

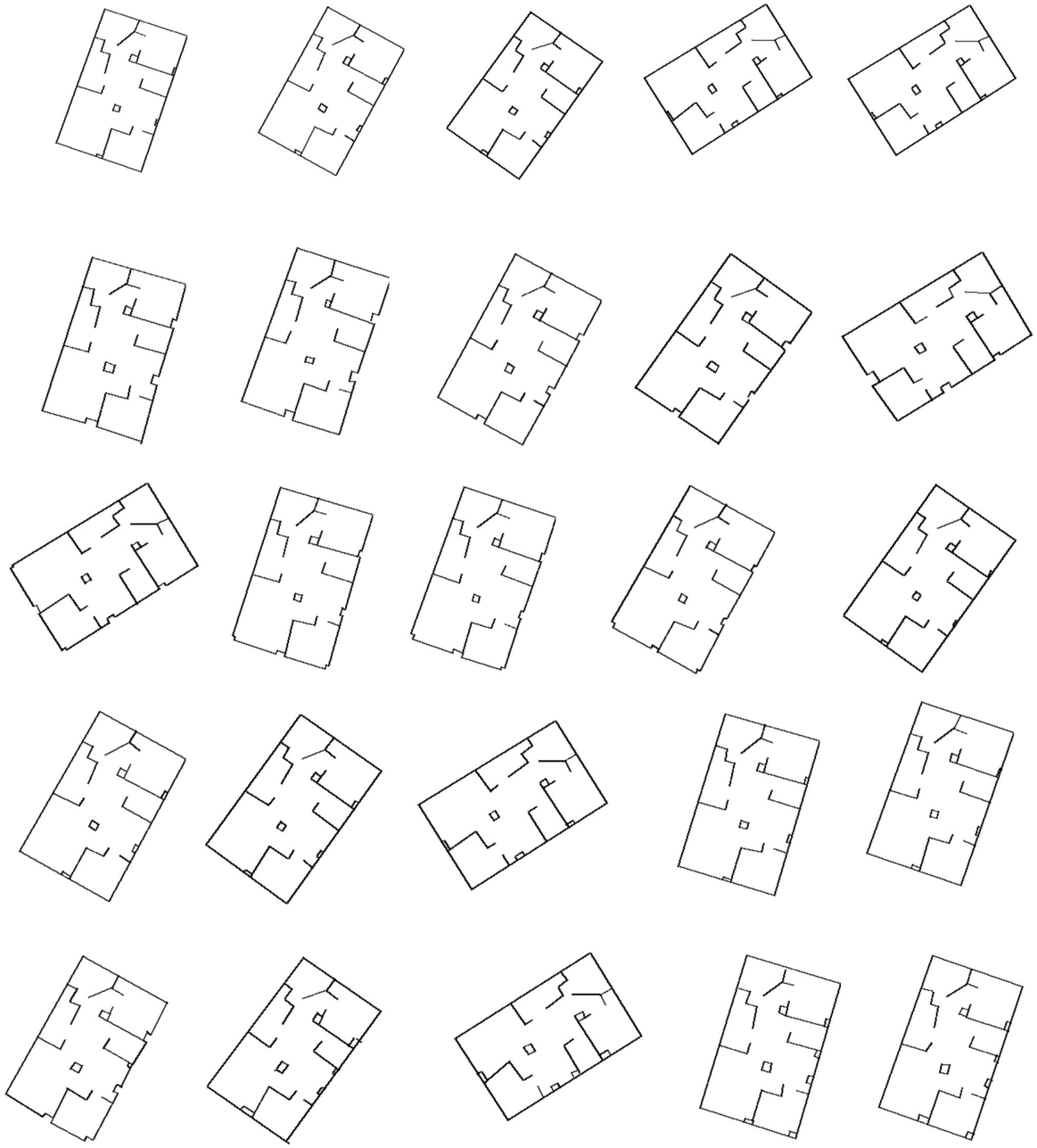


APPENDIX

APPENDIX F: USER TEST 2 USER-REFINED MAPS

All 50 user-refined maps of the second user test are shown on these pages.





APPENDIX G: STATISTICAL TESTS

Fisher's exact test used in the statistical analysis:

FisherExactTest.py > ...

```python
from scipy.stats import fisher_exact

# Contingency table
# [[refined_success, refined_failure],
#  [robot_success, robot_failure]]

table = [[7, 2],
         [4, 5]]

# Two-sided test (standard)
odds_ratio, p_value = fisher_exact(table, alternative='two-sided')

print("Odds ratio:", odds_ratio)
print("P-value:", p_value)


```

Mann Whitney U test used in the statistical analysis:

MannWhitneyUTest.py > ...

```python
from scipy.stats import mannwhitneyu

# A → B
robot_AB = [55.4, 59.3]
refined_AB = [56.9, 54.9]

u_stat, p_AB = mannwhitneyu(robot_AB, refined_AB, alternative='two-sided')

# C → D
robot_CD = [48.7]
refined_CD = [46.4, 47.5, 48.2]

u_stat, p_CD = mannwhitneyu(robot_CD, refined_CD, alternative='two-sided')

# E → F
robot_EF = [81.1]
refined_EF = [76.4, 78.4]

u_stat, p_EF = mannwhitneyu(robot_EF, refined_EF, alternative='two-sided')

print("A→B p-value:", p_AB)
print("C→D p-value:", p_CD)
print("E→F p-value:", p_EF)

```