

# **SPRAWOZDANIE Z PROJEKTU**

Tworzenie systemów robotycznych

Temat: Roboty Taxi

Wykonane przez:

Miłosz Stasiak 240471

Filip Grzymski 240410

# 1 Opis

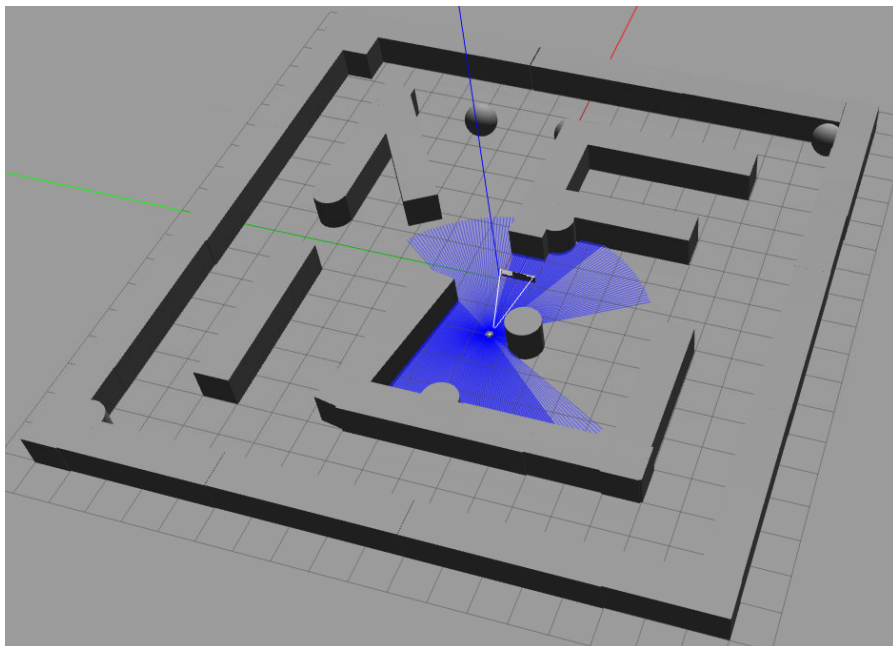
Stworzyliśmy system robotyczny, który zarządza robotyczną taksówką. System wspiera kolejkovanie zadań. Dodatkowo, wykonaliśmy mapę, po której robot porusza się w symulacji. Jej zadaniem jest odwzorowanie miejskiego środowiska, w którym robot mógłby się znaleźć w rzeczywistości.

## 2 Mapa

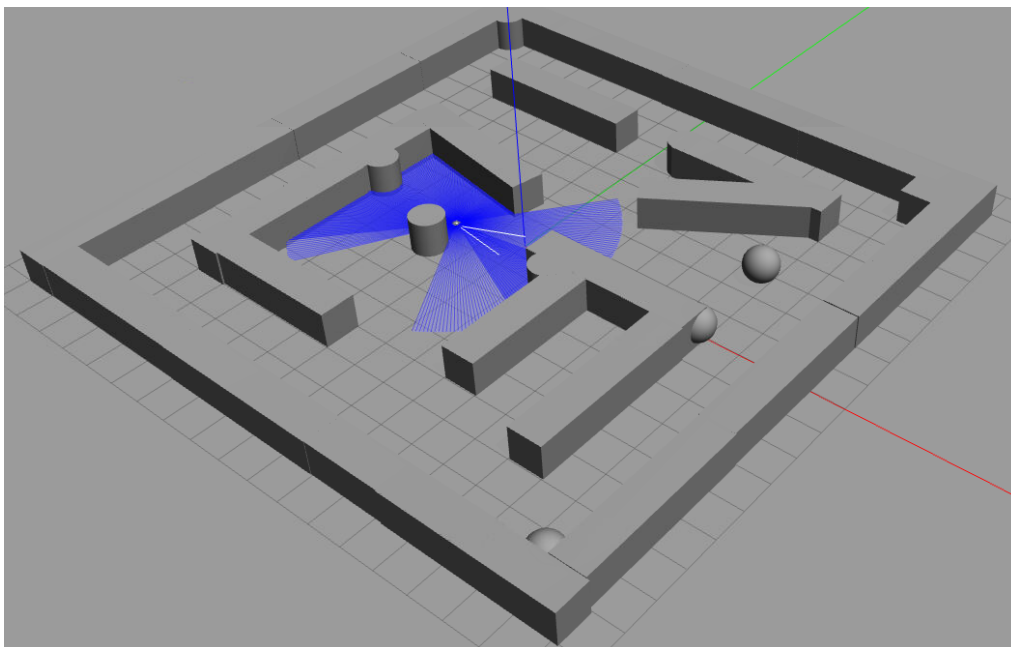
### TODO

---

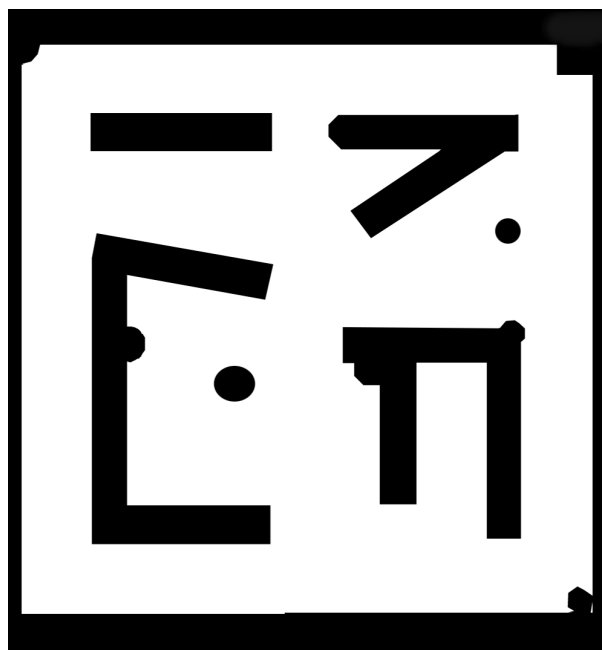
Napisałbym, że 1 iteracja mapy była zbyt powtarzalna i robot się gubił, i że użyliśmy programu cartographer



Ilustracja 1: Widok mapy w rzucie izometrycznego



Ilustracja 2: Inny rzut izometryczny



Ilustracja 3: Model mapy do nawigacji w Rvizz

## 3 Kod

### 3.1 Zasada działania

Nasz system składa się z dwóch programów, które można zaklasyfikować jako back-end i front-end. Pierwszy program komunikuje się bezpośrednio z robotem, a drugi program komunikuje się z użytkownikiem końcowym. Oba programy wymieniają się informacjami za pomocą wspólnej bazy danych w technologii “sqlite3”.

#### Baza danych

Baza danych zawiera jedną tabelę, której definicja w naszym narzędziu ORM jest poniżej. Każdy rekord w tabeli `schedules` odpowiada jednemu rozkazowi, który ma zostać wydany robotowi.

---

```
self.tables = {
    "schedules":
        BaseTable(name="schedules", fields=[
            IntegerFieldSQLite(name="id", primary_key=True),
            BoolFieldSQLite(name="is_completed", null=False, default=0),
            BoolFieldSQLite(name="in_progress", null=False, default=0),
            BoolFieldSQLite(name="timestamp", null=False),
            IntegerFieldSQLite(name="x", null=False),
            IntegerFieldSQLite(name="y", null=False),
            IntegerFieldSQLite(name="z", null=False),
        ]),
}
```

---

Odczytywanie następnego polecenia odpowiada się za pomocą kwerendy

---

```
SELECT * FROM schedules WHERE is_completed=0 AND in_progress=0 ORDER BY timestamp
```

---

## Back-end

Wydawanie komend akcji `navigate_to_pose` zaimplementowaliśmy za pomocą timera, który co jedną sekundę wykonuje zapytanie do bazy danych. Baza danych odpowiada lista zadań, które robot ma wykonać a program decyduje, które zadanie mu przydzielić. Po wykonaniu zadania robot zgłasza koniec akcji, co w programie powoduje wykonanie kodu, który dokonuje odpowiednich zmian w bazie danych.

---

```
class Taxi(Node):
    def __init__(self):
        super().__init__('taxi')

        self.is_robot_busy = False

        self.publisher_ = self.create_publisher(Twist,
                                                'cmd_vel',
                                                10)

        self.timer = self.create_timer(
            1.0,
            self.read_db)

[...]

def read_db(self):
    navigate_action = ActionClient(
        taxi,
        NavigateToPose,
        '/navigate_to_pose')

    BASE_DIR = Path("/home/ubuntu/turtlebot3_ws/src/jupyter_notebooks")
    storage_obj = BaseStorage(BASE_DIR)
    with storage_obj as s:
        cursor = s.conn.cursor()
        rows = cursor.execute("SELECT * FROM schedules WHERE is_completed=0 AND
                               in_progress=0 ORDER BY timestamp").fetchall()

        print(rows)

    if len(rows) > 0 and not self.is_robot_busy:
        navigate_pose_goal = NavigateToPose.Goal()
        goal_pose = PoseStamped()
        goal_pose.header.stamp = self.get_clock().now().to_msg()
        goal_pose.pose.position.x = float(rows[0]["x"])
        goal_pose.pose.position.y = float(rows[0]["y"])
```

```

goal_pose.pose.orientation.z = 0.0
goal_pose.pose.orientation.w = 0.0
navigate_pose_goal.pose = goal_pose

self.goal_id = rows[0]["id"]

goal_future_pose =
    navigate_action.send_goal_async(navigate_pose_goal, feedback_callback=self.print_feedback)
goal_future_pose.add_done_callback(self.received_task)

```

[...]

```

def history_success(self, future):
    BASE_DIR = Path("/home/ubuntu/turtlebot3_ws/src/jupyter_notebooks")

    storage_obj = BaseStorage(BASE_DIR)
    with storage_obj as s:
        cursor = s.conn.cursor()
        rows = cursor.execute(f"UPDATE schedules SET is_completed=1, in_progress=0 WHERE
                                id={self.goal_id}")
        s.conn.commit()

    self.is_robot_busy = False

```

---

## Front-end

Aplikacja front-end pozwala użytkownikowi komunikować się z robotem. Aplikacja opiera się na interaktywnych komórkach Jupyter Notebook, w których użytkownik musi wpisać miejsce do, którego robot ma się udać. W rzeczywistości mogłoby to wyglądać w następujący sposób. Użytkownik w aplikacji mobilnej wpisuje, gdzie się aktualnie znajduje, robot do niego przyjeżdża. Następnie po wejściu do taksówki użytkownik wpisuje w aplikacji dokąd chce jechać, a robot taxi go tam zabiera.

---

```
def send_robot(x="0", y="0"):
    BASE_DIR = Path("/home/ubuntu/turtlebot3_ws/src/jupyter_notebooks")

    timestamp = int(time.time()*1000)

    storage_obj = BaseStorage(BASE_DIR)
    with storage_obj as s:
        timestamp = int(time.time()*1000)
        s.insert_row({"is_completed":False, "in_progress":False, "x":float(x),
                    "y":float(y), "z":0, "timestamp": timestamp}, "schedules")
```

---



```
def send_robot(x="0", y="0"):
    BASE_DIR = Path("/home/ubuntu/turtlebot3_ws/src/jupyter_notebooks")

    timestamp = int(time.time()*1000)

    storage_obj = BaseStorage(BASE_DIR)
    with storage_obj as s:
        timestamp = int(time.time()*1000)
        s.insert_row({"is_completed":False, "in_progress":False, "x":float(x), "y":float(y), "z":0, "timestamp": timestamp}, "schedules")

im = interact_manual(send_robot)
im.widget.children[2].description = 'Przywołaj robota'
display(im)
```

The screenshot shows the Jupyter Notebook interface. The top part displays the Python code for the `send_robot` function and its interactive widget. The widget consists of two input fields labeled 'x' and 'y', both containing the value '0'. Below these fields is a button labeled 'Przywołaj robota'.

Ilustracja 4: Interfejs użytkownika