POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ

PRACA DYPLOMOWA INŻYNIERSKA
na kierunku Automatyka i Robotyka

Patryk Piotr DŻOŃ
Nr indeksu: 230393

Rok akademicki: 2012/2013
Warszawa, 15.09.2012

WYKORZYSTANIE SYSTEMU ROS WE WSPÓŁPRACY Z SYMULATOREM GAZEBO DO STEROWANIA WIRTUALNYM ROBOTEM MOBILNYM

Zakres pracy:

1. *Wprowadzenie.*
2. *Opis wykorzystywanego oprogramowania.*
3. *Tworzenie środowiska w symulatorze.*
4. *Implementacja pakietów.*
5. *Realizacja wybranych scenariuszy symulacji.*
6. *Podsumowanie i wnioski.*

KIEROWNIK ZAKŁADU STEROWANIA
Dr hab. inż. Bartłomiej Beliczyński

Kierujący pracą: dr inż. Witold Czajewski

Termin wykonania: 09.09.2013

Temat pracy: Wykorzystanie systemu ROS we współpracy z symulatorem Gazebo do sterowania wirtualnym robotem mobilnym

Streszczenie

Celem niniejszej pracy inżynierskiej była symulacja robota mobilnego poruszającego się w środowisku domowym. Robot działał w oparciu o Robot Operating System, a symulacja odbywała się w programie Gazebo, dlatego pierwszym elementem pracy jest opis wykorzystywanego oprogramowania pozwalający na zrozumienie zasad jego funkcjonowania w projekcie.

Druga część pracy opisuje modelowanie środowiska symulacyjnego. Najważniejszym punktem było szczegółowe odtworzenie robota mobilnego Pioneer 3-DX wyposażonego w czujnik laserowy oraz bezprzewodową kamerę. Zadanie to ułatwiła możliwość wykorzystania modeli udostępnionych przez programistów ROS, dzięki czemu nie było konieczne zaczynanie od zera, a jedynie skonfigurowanie plików do własnych potrzeb. Kolejnym zagadnieniem było utworzenie projektu budynku mieszkalnego wraz z realistycznie odwzorowanym wyposażeniem. Wiązało się to głównie z pracą w programie Gazebo, w którym skompletowano trójwymiarowe modele pochodzące z dostępnej bezpłatnie bazy internetowej.

W kolejnym rozdziale przeprowadzono implementację wszystkich pakietów systemu ROS, które były niezbędne do funkcjonowania projektu, łącznie z tymi realizującymi współpracę z symulatorem Gazebo. Każdy pakiet został szczegółowo opisany: od wyjaśnienia sposobu działania, poprzez przetwarzane przez niego informacje oraz dostępne parametry pracy, aż do utworzenia pliku ze wstępną konfiguracją oraz pliku uruchomieniowego.

Ostatnią częścią projektu było przeprowadzenie kilku wybranych scenariuszy symulacji mających na celu zbadanie możliwości sterowania robotem mobilnym w środowisku domowym. Jednym z nich było utworzenie mapy pomieszczeń, która została późnej dołączona do pakietu nawigacyjnego w celu zwiększenia jego efektywności. Robot poprawnie reagował na manualne sterowanie, a po przeprowadzeniu kilku zmian w plikach nawigacyjnych i lepszemu dostosowaniu ich do specyfiki środowiska pracy, pomyślnie realizował autonomiczną nawigację do miejsca w budynku wybranego przez użytkownika, unikając przy tym zderzenia z przeszkodami.

Wszystkie zasadnicze założenia projektu zostały spełnione, a dodatkową funkcjonalnością było utworzenie interfejsu graficznego pozwalającego na wygodne i łatwe sterowanie robotem.

Thesis subject: Application of the Robot Operating System in cooperation with the Gazebo simulator to control a virtual mobile robot

Abstract

The aim of this Bsc thesis was to simulate a mobile robot operating in a household environment. The robot was using the Robot Operating System and the simulation was performed in the Gazebo simulator, so the first element of this project was to describe the software in a way that should help to understand how the project works.

The second part of the thesis describes building of the environment. The most important point was to make a detailed model of the Pioneer 3-DX mobile robot equipped with some accesories: a laser sensor and a wireless camera. Fortunately, thank to the work of many ROS developers, it was not necessary to make a model from scratch, but it was possible to import and configure an existing one. The next task was to build a completely furnished household model. This was done mainly in Gazebo simulator with use of many three-dimensional models, which were available to download for free from an internet database.

In the next chapter all essential ROS packages, including the set of packages for interfacing with Gazebo, were implemented. Each of them has been fully described: from how it will work and which data will be processed, through the explanation of the parameters that can be configured, to creating a tentative configuration file and a useful launch file.

The last part of this project was to carry out a couple of simulations and test the possibilities to control the movement of the mobile robot in the household environment. One of the scenarios was to make a map of the appartment, which can be used by navigation package to improve the efficiency. In the teleoperation mode, the robot model was working properly all the time. The navigation package needed a couple of changes in configuration files. After a better adaptation of the navigation parameters to the characterics of the environment, the mobile robot was able to reach autonomously a given point in the appartment avoiding all of the obstacles.

All of the goals of this thesis have been met. An additional achievement was a graphical user interface created to make the control of the robot easier and more convenient.

Warszawa, dnia 10 września 2014 roku.

Politechnika Warszawska
Wydział Elektryczny

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska pt. **„Wykorzystanie systemu ROS we współpracy z symulatorem Gazebo do sterowania wirtualnym robotem mobilnym"**

- została napisana przeze mnie samodzielnie
- nie narusza niczyich praw autorskich
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej.

Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Imię i Nazwisko dyplomanta: **Patryk Dżoń**

Podpis dyplomanta: Patryk Dżoń

Spis treści

1. Wprowadzenie

Robotyka mobilna to szybko rozwijająca się gałąź nauki łącząca wiedzę z zakresu mechaniki, elektroniki i informatyki. Skupia się na urządzeniach posiadających zdolność przemieszczania się względem otoczenia, których podstawowym układem wykonawczym jest układ lokomocyjny. Bez względu na to, czy ruch ten jest wynikiem manualnego sterowania przez człowieka, czy następuje w sposób mniej lub bardziej autonomiczny, urządzenia te nazywamy robotami mobilnymi. Działają one na lądzie, w wodzie, w powietrzu, a nawet w kosmosie wykonując szereg działań związanych z eksploracją terenu. Wśród robotów mobilnych poruszających się po powierzchni ziemi najpopularniejsze są systemy kołowe, najbardziej efektowne systemy kroczące (na przykład humanoidalne), ale nie można też pominąć napędów gąsienicowych czy skaczących.

Duża część robotów (zwłaszcza mobilnych) to nadal systemy sterowane przez człowieka. Obecnie robotyka dąży do rozwiązania zagadnienia ich pełnej autonomii, która dla platform mobilnych oznacza głównie posiadanie sprawnego systemu nawigacyjnego. Składa się on po pierwsze z oprogramowania umożliwiającego lokalizację (umiejscowienie względem układu odniesienia) robota, po drugie z metod planowania ruchu pozwalających na osiągnięcie przez robota założonego celu. System działa dzięki nieustannemu przetwarzaniu danych o otaczającym środowisku pochodzących z zewnętrznych czujników. Projektowanie i udoskonalanie tak złożonych procesów wymaga przeprowadzenia wielu badań, które dzięki komputerowym symulatorom stały się tańsze, łatwiejsze, a przez to bardziej dostępne – obecnie nawet dla studentów i pasjonatów, którzy nie posiadają przecież specjalistycznego sprzętu.

1.1. Cele i założenia pracy

Głównym celem pracy jest przeprowadzenie symulacji robota mobilnego działającego w oparciu o Robot Operating System. Zasadniczo symulacje służą do testowania tworzonego oprogramowania, ale bez konieczności narażania realnego sprzętu na uszkodzenia mogące powstać w wyniku błędów. W tym wypadku posłużą one do poznania: zasad działania systemu ROS, sposobów na manualne kontrolowanie robota, a także pakietów umożliwiających autonomiczną nawigację robota w nieznanym środowisku.

Ważnym elementem każdego robota są zewnętrzne czujniki, dzięki którym może on zdobywać informacje o otaczającym go środowisku. Dlatego jednym z celów pracy jest dołączenie do robota kilku akcesoriów i ich implementacja w ROS. W przypadku symulacji istnieje oczywiście możliwość wykorzystania czujników skomplikowanych i kosztownych, jednak dla zachowania

realizmu zdecydowano się na symulację takich, które są dostępne, relatywnie tanie i współpracują z danym robotem - zgodnie z danymi producenta. Dzięki temu projekt po niewielkich zmianach nadawał się będzie do pracy z użyciem sprzętu. Dodatkowym zadaniem będzie umożliwienie sterowania robotem za pomocą wygodnego interfejsu graficznego.

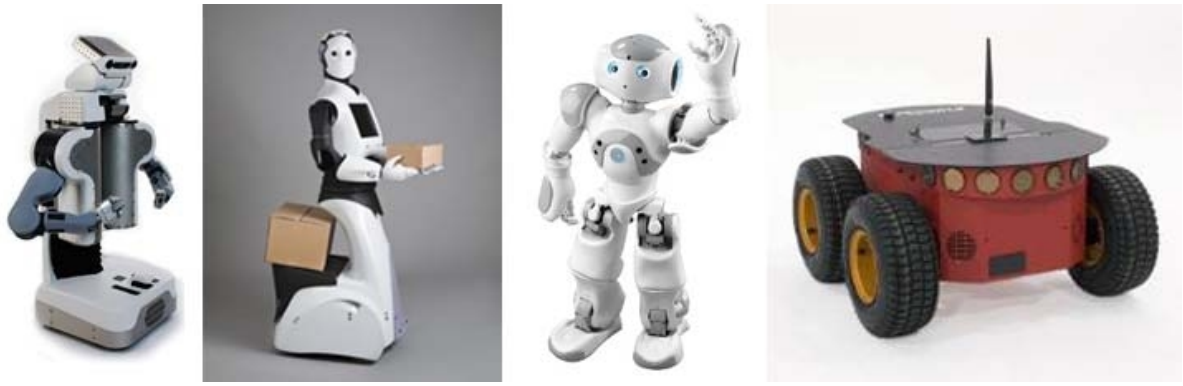
Symulacja będzie odbywać się z programie Gazebo wykorzystując szczegółowo odwzorowany model robota oraz fikcyjne środowisko domowe i zachowując zasady fizyki. Robot mobilny powinien reagować na manualne sterowanie, a także samodzielnie przemieścić się do wybranego punktu unikając kolizji z innymi obiektami. Dane pochodzące z czujników powinny być poprawnie wizualizowane.

2. Opis wykorzystywanego oprogramowania

Projekt zakłada istnienie ścisłej integracji Robot Operating System z symulatorem Gazebo. Dlatego wyjaśnienie zasad tej współpracy stanowi ważną część bieżącego rozdziału. Dodatkowo streszczono tutaj ogólną charakterystykę użytego oprogramowania, z naciskiem na wyjaśnienie ważnych terminów oraz opisanie procesu instalacji i konfiguracji środowiska pracy.

2.1. ROS

Robot Operating System jest rozbudowaną platformą programistyczną szeroko używaną w robotyce. Składa się z zestawu narzędzi, bibliotek i konwencji, których celem jest ułatwianie tworzenia skomplikowanych zachowań robotów. Główną filozofią przyświecającą temu systemowi jest tworzenie oprogramowania, które może być wykorzystywane przez wiele rodzajów robotów (rys. 2.1) przy jedynie niewielkich zmianach w kodzie. Dzięki temu rozmaite funkcjonalności mogą być udostępniane przez programistów i łatwo implementowane przez innych użytkowników.



Rys. 2.1. Przykłady robotów w pełni kompatybilnych z ROS [4]

Koncept ROS powstał w 2007 roku jako owoc wielu wcześniejszych badań prowadzonych nad oprogramowaniem przeznaczonym dla robotów. Jego realizacja była możliwa dzięki współpracy Uniwersytetu Stanforda w Kalifornii z prywatnym przedsiębiorstwem o nazwie Willow Garage, które jest laboratorium naukowym i inkubatorem technologicznym zajmującym się rozwijaniem wolnego oprogramowania oraz sprzętu (roboty PR2 i Turtlebot). Pierwsza stabilna wersja ROS ujrzała światło dzienne na początku 2010 roku. Od tego czasu system doczekał się kilku dystrybucji i kilku tysięcy pakietów z oprogramowaniem. 22 lipca 2014 roku wydano najnowszą wersję o nazwie Indigo Igloo.

Robot Operating System to duży projekt działający na zasadach wolnego oprogramowania,

dlatego jego obecny kształt jest rezultatem pracy międzynarodowego środowiska programistów, którzy przyczynili się do powstania lub udoskonalania pojedynczych pakietów. Społeczność ROS składa się obecnie z tysięcy użytkowników na całym świecie: hobbystów, studentów, naukowców, a nawet inżynierów projektujących manipulatory przemysłowe [6].

2.1.1. Instalacja systemu

Pomimo starań programistów ROS o kompatybilność z wieloma systemami operacyjnymi, najlepiej jest korzystać z systemu Linux Ubuntu w wersji LTS (*Long Time Support* – długi okres wsparcia). Niniejszy projekt został opracowany w ROS Hydro zainstalowanym w systemie Ubuntu 12.04 LTS (*Precise Pangolin*). Ten zestaw ma wsparcie producenta do 2017 roku.

Rekomendowaną, pełną instalację ROS w systemie Ubuntu 12.04 Precise za pomocą pakietów Debiana możemy przeprowadzić w kilku prostych krokach:

1. Uruchamiamy terminal (np. z klawiatury kombinacją przycisków Ctrl + Alt + T).

2. Dodajemy do systemu repozytorium, z którego zostanie pobrany ROS (w przypadku posiadania innej wersji systemu Linux wpisujemy jego nazwę kodową w miejsce *precise*) :

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. Dodajemy klucze repozytorium do zaufanej listy systemu:

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -
```

4. Aktualizujemy posiadane oprogramowanie, aby uniknąć potencjalnych problemów z bibliotekami:

```
sudo apt-get update
```

5. Instalujemy pełny pakiet ROS (w miejsce *hydro* możemy wstawić nazwę dowolnej obsługiwanej przez nas system dystrybucji); jeśli podczas instalacji pojawi się pytanie o 'hddtemp', wybieramy <Nie>:

```
sudo apt-get install ros-hydro-desktop-full
```

6. Inicjalizujemy *rosdep* – jest to wymagane do uruchomienia głównych komponentów ROS:

```
sudo rosdep init
```

7. Następnie zgodnie z rekomendacją uruchamiamy:

```
rosdep update
```

8. Na końcu ustawiamy zmienne środowiskowe ROS:

```
echo "source /opt/ros/hydro/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

2.1.2. System plików ROS i przestrzeń robocza *Catkin*

Spośród zasobów, które znajdują się na dysku twardym, możemy wyróżnić:

a) PAKIETY – główne jednostki organizacji oprogramowania w ROS. Pakiet może zawierać procesy (węzły), biblioteki, zbiory danych, pliki konfiguracyjne, czyli wszystkie pliki, które korzystnie jest mieć zebrane w jednym miejscu. Pakiety to najmniejsze jednostki, które można skompilować i udostępnić - z nich składa się cały system ROS;

b) META-PAKIETY – służą do reprezentowania grupy połączonych ze sobą pakietów;

c) REPOZYTORIA – kolekcja pakietów, które łączy wspólny system VCS (kontroli wersji), dlatego mogą być one udostępniane razem korzystając z pomocy automatycznych narzędzi;

d) MANIFESTY PAKIETÓW – (*package.xml*) każdy manifest zawiera dane na temat pakietu: nazwę, wersję, opis, informacje o licencji, zależności, i tym podobne;

e) PLIKI *CMakeList.txt* – zawierają informacje dla kompilatora (m.in. nazwę pakietu, listę innych pakietów oraz bibliotek potrzebnych do kompilacji, zasady kompilacji).

Catkin to nazwa oficjalnego systemu kompilacji ROS, następcy pierwotnego systemu *rosbuild*. Łączy on makra *CMake* ze skryptami *Python* w taki sposób, aby zapewnić funkcjonalność standardowego kompilatora *CMake*, jednocześnie dodając wsparcie dla automatycznego wyszukiwania pakietów i tworzenia wielu rozbudowanych (oraz często zależnych od siebie) projektów w tym samym czasie.

Tworzenie przykładowej przestrzeni roboczej C*atkin* przebiega następująco:

```
mkdir -p ~/projekt/src
cd ~/projekt/src
catkin_init_workspace
```

Każdy pakiet wykorzystywany w bieżącym projekcie (poza oficjalnymi pakietami ROS pochodzącymi z repozytoriów) musi znajdować się w folderze /src/, zgodnie z regułą przedstawioną poniżej:

```
projekt/                – PRZESTRZEŃ ROBOCZA
  src/                  – FOLDER ZAWIERAJĄCY PAKIETY
    CMakeLists.txt      – główny plik CMake, dostarczany przez Catkin, umieszczany za pomocą
                          komendy 'catkin_init_workspace'
    pakiet_1/
      CMakeLists.txt    – plik CMakeLists.txt dla pakietu_1
      package.xml       – manifest dla pakietu_1
    ...
    pakiet_n/
      CMakeLists.txt    – plik CMakeLists.txt dla pakietu_n
      package.xml       – manifest dla pakietu_n
```

Przed uruchomieniem projektu należy skompilować przestrzeń roboczą. Czynność tę trzeba powtarzać po każdym dodaniu pakietów lub przeprowadzeniu zmian w plikach wykonywalnych. Jest to możliwe do wykonania także wtedy, gdy w folderze /src/ nie ma żadnych pakietów. W tym celu należy uruchomić konsolę, przejść do głównego folderu przestrzeni roboczej i skorzystać z komendy *catkin_make*:

```
cd ~/projekt/
catkin_make
```

Spowoduje to utworzenie plików i folderów, które zorganizowane są w następujący sposób:

```
projekt/                – PRZESTRZEŃ ROBOCZA
  src/                  – FOLDER Z PAKIETAMI
  build/                – FOLDER Z PLIKAMI SKOMPILOWANYMI
    catkin_generated/
    CmakeFiles/
    ...
  devel/                – FOLDER Z PLIKAMI ZAINSTALOWANYMI
    etc/
    include/
    lib/                – pliki wykonywalne
    share/
    setup.bash
    ...
```

Przed uruchomieniem projektu konieczne jest także umieszczenie przestrzeni roboczej na szczycie środowiska, to znaczy umożliwienie wyszukiwania pakietów i uruchamiania węzłów znajdujących się w tej przestrzeni z poziomu terminala systemowego. Po uruchomieniu okna konsoli należy każdorazowo uruchomić komendę *source*, w której wskazujemy plik *setup.bash*

znajdujący się w odpowiedniej przestrzeni roboczej. W tym wypadku będzie to:

```
$ source ~/projekt/devel/setup.bash
```

W podstawowych zastosowaniach, kiedy pracujemy w jednej przestrzeni roboczej *Catkin*, wygodne jest dodanie ścieżki do pliku *setup.bash* na stałe do zmiennych środowiskowych:

```
echo "source ~/projekt/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

2.1.3. Koncept ROS na poziomie procesów

Graf obliczeń (rys. 2.2) to sieć peer-to-peer procesów ROS, które wspólnie przetwarzają dane. Składają się na niego następujące procesy:

Rys. 2.2. Graf obliczeń systemu ROS [1]

a) MASTER – stanowi trzon systemu ROS. Rejestruje nazwy wszystkich procesów składających się na graf obliczeń, dzięki niemu węzły są w stanie się odnaleźć i skomunikować, np. wysyłając wiadomość.

b) WĘZEŁ (NODE) – to proces wykonujący obliczenia. ROS jest zaprojektowany jako system modułowy, dlatego każdy system obsługujący robota składa się w wielu węzłów: jeden kontroluje napęd, drugi czujnik laserowy, kolejny zajmuje się nawigacją. Każdy węzeł jest kodem napisanym w języku C++ lub Python z użyciem odpowiedniej biblioteki - *roscpp* lub *rospy*.

c) WIADOMOŚĆ (MESSAGE) – węzły komunikują się między sobą przekazując wiadomości. Wiadomość to prosta struktura danych typu *integer*, *floating point*, *boolean* lub *array*.

d) TEMAT (TOPIC) – to nazwa skupiająca wiadomości jednego typu. Przesyłanie wiadomości w ROS odbywa się na zasadzie publikacji i subskrybcji. Każdy węzeł wysyła wiadomość poprzez publikowanie jej w odpowiednim temacie. Węzeł, który potrzebuje określonego rodzaju danych, subskrybuje odpowiedni temat. Pojedynczy węzeł może publikować i subskrybować wiele tematów.

e) SERWER PARAMETRÓW – jest częścią procesu MASTER, umożliwia przechowywanie danych, do których dostęp mają wszystkie pracujące węzły.

f) SERWIS (SERVICE) – proces zajmujący się interakcjami typu żądanie-odpowiedź. Składa się z pary wiadomości, z których jedna opisuje strukturę danych wysyłanego żądania, a druga oczekiwanej odpowiedzi.

2.1.4. Przydatne narzędzia

ROS oddaje do dyspozycji użytkownika wiele narzędzi ułatwiających pracę nad projektem. Są to między innymi:

a) WIZUALIZATOR RVIZ – umożliwia trójwymiarową wizualizację modelu URDF robota i danych pochodzących z wielu czujników, takich jak: skany laserowe, trójwymiarowe chmury punktów, obraz z kamery. Korzysta z informacji zawartych w bibliotece *tf*, dzięki czemu wyświetla dane z perspektywy wybranego układu współrzędnych. Dzięki wizualizacji użytkownik może zobaczyć wszystko to, co „widzi” robot, i szybko identyfikować problemy wynikające z błędnego działania czujników.

b) RQT – program służący do tworzenia graficznych interfejsów użytkownika, które pomagają w diagnostyce i kontroli nad robotem. Zawiera bibliotekę wbudowanych wtyczek, które możemy dodawać do okna programu w dowolnych miejscach i rozmiarach, możliwe jest również tworzenie własnych pluginów. Jedną z wartych uwagi wtyczek jest *rqt_graph*, która pokazuje wszystkie węzły działające aktualnie w systemie i połączenia między nimi, co pozwala łatwiej zrozumieć strukturę pojedynczych procesów oraz całego systemu ROS.

2.2. Gazebo

Gazebo to program umożliwiający trójwymiarową symulację wielu złożonych obiektów, w szczególności robotów, czujników i ich środowisk pracy. Jest udostępniany dla systemu Linux Ubuntu na licencji *Apache 2.0*, czyli bezpłatnie zarówno dla zastosowań prywatnych, jak i komercyjnych.

Powstał na Uniwersytecie Południowej Kalifornii jako część projektu Player, który już od 2000 roku ma na celu tworzenie darmowego oprogramowania z zakresu robotyki. Jednak prawdziwą popularność zyskał dopiero w 2011 roku, kiedy to został zintegrowany z bibliotekami Robot Operating System, jak również stał się niezależnym projektem. Od tamtej pory rozwojem Gazebo

zajmuje się Open Source Robotic Foundation – organizacja non-profit wspierająca rozwój robotyki, wywodząca się z przedsiębiorstwa Willow Garage. W sierpniu 2014 roku wydano wersję 4.0 programu i zapowiedziano kolejne aktualizacje, które mają się ukazywać co 6 miesięcy.

Gazebo jest obecnie rozbudowanym narzędziem wykorzystywanym głównie przez społeczność ROS oraz uczestników konkursu DARPA Robotics Challenge. Składa się z silnika graficznego OGRE *(Object – Oriented Graphics Rendering Engine)*, który odpowiada za realistyczny wygląd modelowanych obiektów, oświetlenia i cieni. Dokładne odwzorowanie fizyki, w tym zaawansowany model kolizji, zapewnia silnik ODE *(Open Dynamics Engine)*.

Symulacja w programie Gazebo składa się z dwóch podstawowych części:

- world – plik lub zbiór plików w formacie SDF *(Standard Description Format)* napisanych z użyciem języka XML; zawiera wszystkie elementy tworzące świat w symulacji: roboty, czujniki, światła oraz obiekty statyczne;

- plugin – plik napisany w języku C++ lub Python, zapewnia możliwość interakcji, np. poruszanie obiektów lub zaprogramowanie odpowiedzi na zdarzenie. Jest 5 rodzajów pluginów: *World*, *Model*, *Sensor, System*, *Visual*; nazwanych zgodnie z elementem symulacji, na który mogą oddziaływać.

2.2.1. Instalacja i uruchomienie

Aby pobrać najnowszą wersję programu wystarczy wejść na stronę internetową Gazebo [2] i postępować zgodnie z instrukcjami. W wypadku, kiedy korzystamy również z pełnej instalacji Robot Operating System, odpowiednia wersja Gazebo została już z nim zintegrowana.

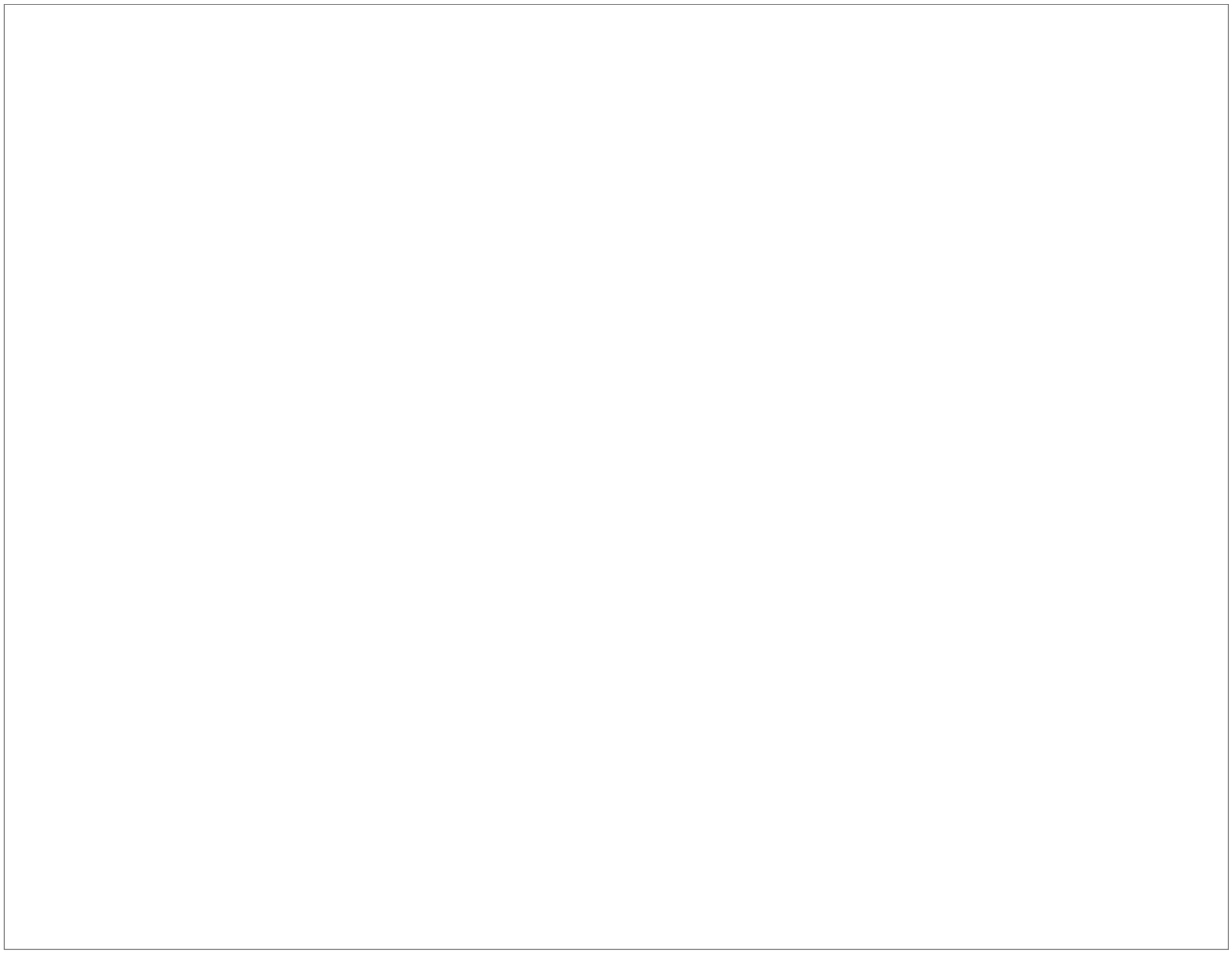
W celu uruchomienia programu wystarczy wpisać w terminalu komendę:

```
gazebo
```

Uruchamia ona dwa procesy. Pierwszym jest *gzserver*, który jest rdzeniem całego programu – wczytuje symulowany świat, odpowiada za fizykę i generowanie danych z czujników oraz może działać niezależnie od interfejsu graficznego. Drugi to *gzclient* – klient graficzny, który podłącza się do serwera i uruchamia interfejs użytkownika, pozwalając na wizualizację symulacji i kontrolę nad jej właściwościami.

2.3. Integracja

We wcześniejszych wersjach oprogramowania symulator Gazebo był integralną częścią ROS i znajdował się w stosie *simulator_gazebo*. Od momentu wydania wersji 1.9 i pojawienia się ROS Hydro nie istnieją między nimi bezpośrednie zależności, a Gazebo jest instalowany jako samodzielny pakiet Ubuntu. Aby osiągnąć potrzebną integrację został stworzony nowy zestaw pakietów ROS o nazwie *gazebo_ros_pkgs* (rys. 2.3). Zapewniają one niezbędne interfejsy do symulacji robota w Gazebo za pomocą wiadomości i serwisów ROS. Korzystają z nowego budowania *Catkin*, mają zredukowane duplikowanie kodu z Gazebo, starają się także traktować pliki URDF i SDF tak równoważnie, jak to możliwe.



Rys. 2.3. Zawartość meta-pakietu realizującego integrację ROS i Gazebo [2]

Meta-pakiet *gazebo_ros_pkgs* w każdej dystrybucji ROS jest zoptymalizowany pod kątem odpowiedniej wersji Gazebo, dlatego dla w pełni zintegrowanego systemu taka wersja jest zalecana i instalowana wraz z pełną instalacją ROS. Dla ROS Hydro jest to Gazebo 1.9.x., dla ROS Indigo – Gazebo 2.x.

2.3.1. Używanie plików URDF w Gazebo

Universal Robot Description Format to format pliku używany w ROS do opisu wszystkich elementów robota. Definiuje on kinematyczne i dynamiczne właściwości pojedynczego, odizolowanego robota. Dlatego, pomimo uniwersalności w nazwie, nie nadaje się do użycia w symulatorze, gdyż nie określa wielu niezbędnych parametrów (np. inercji, tarcia). Jednak posiadając model URDF robota wystarczy utworzyć dodatkowy plik zawierający brakujące elementy, a Gazebo połączy je i skonwertuje do używanego przez siebie formatu SDF.

2.3.2. Uruchamianie zintegrowanych środowisk ROS i Gazebo

Jeśli ROS i Gazebo zostały ze sobą poprawnie zintegrowane, istnieje możliwość uruchomienia symulatora za pomocą komendy *rosrun*, poprzedzonej komendą *roscore*:

```
roscore
rosrun gazebo_ros gazebo
```

W ten sposób uruchomiony zostaje pusty świat w Gazebo. To, że jest on połączony ze środowiskiem ROS, można sprawdzić komendą:

```
rostopic list
```

Zwróci nam ona listę subskrybowanych i publikowanych przez Gazebo tematów ROS:

```
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_description
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
```

2.3.3. Korzystanie z narzędzia *roslaunch*

Narzędzie *roslaunch* jest główną metodą łatwego uruchamiania węzłów w ROS. Korzysta z plików konfiguracyjnych z rozszerzeniem *.launch* (w formacie XML), które definiują węzły do uruchomienia wraz z ich parametrami. Użycie tej metody wymaga podania nazwy pliku *.launch* oraz pakietu, w którym ów plik się znajduje, zgodnie z poniższym wzorem:

```
roslaunch package_name file.launch
```

Składnia przykładowego pliku *file.launch* zawierającego jeden węzeł wygląda następująco:

```
<launch>
 <node name="name" pkg="package_name" type="package_name” output="screen"/>
</launch>
```

Narzędzie *roslaunch* dzięki plikom konfiguracyjnym pozwala na korzystanie z bardziej zaawansowanych opcji uruchamiania symulacji. Najważniejsza jest oczywiście możliwość łatwego uruchamiania modeli utworzonych światów oraz wczytywania do nich modeli robotów. I tak, komenda otwierająca pusty świat w symulatorze to:

```
roslaunch gazebo_ros empty_world.launch
```

Plik konfiguracyjny *empty_world.launch* zapewnia uruchomienie programu Gazebo poprzez odpowiednie węzły (rozdział 2.2.1.). Definiuje też logikę symulowanego świata poprzez parametry, dlatego stanowi punkt wyjściowy do tworzenia własnego pliku na potrzeby projektu, który może wyglądać następująco:

```
<launch>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find package_name)/world_name.world" />
    <arg name="debug" value="false" />
    <arg name="gui" value="true" />
    <arg name="paused" value="false" />
    <arg name="use_sim_time" value="true" />
    <arg name="headless" value="false" />
  </include>
</launch>
```

Znacznik <include> pozwala załączać zawartość jednych plików *.launch* w drugich. Tutaj dołączono plik *empty_world.launch*, a następnie jako wartość parametru *world_name* podano ścieżkę do pliku zawierającego zaprojektowane środowisko symulacyjne. Ten sposób daje dostęp do łatwej zmiany parametrów, na przykład: *gui* – decydujemy czy wyświetlić interfejs graficzny Gazebo; *paused* – wybieramy stan w jakim wystartuje symulacja.

W pliku *.launch* mamy także możliwość załadowania informacji na serwer parametrów, dzięki czemu będą one dostępne dla każdego węzła uruchomionego podczas sesji. Przydatne jest na przykład zapisanie modelu URDF robota na serwerze jako parametr *robot_description*. Jeśli posiadamy model w pliku z makrami *Xacro* musimy wcześniej (za pomocą skryptu *xacro.py*) skonwertować go do „czystego” *XML*:

```
<param name="robot_description" command="$(find xacro)/xacro.py '$(find
package_name)/my_robot.xacro'" />
```

Wczytywanie robotów do świata w Gazebo jest przeprowadzane przez węzeł *urdf_spawner* metodą *ROS Service Spawn*. Korzysta ona ze skryptu w języku Python, który wysyła żądanie wczytania modelu robota (znajdującego się już na serwerze parametrów) w określone miejsce środowiska (tutaj o współrzędnych [3,1]):

```
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen" args="-urdf -x 3 -y 1 -z 0 -model my_robot -param robot_description" />
```

Po wczytaniu modelu robota węzeł *urdf_spawner* wyłącza się automatycznie.

3. Tworzenie środowiska w symulatorze

Każda symulacja to w zasadzie modelowanie – przechwytywanie właściwości fizycznych obiektu, procesu lub systemu w zbiór zasad, relacji i parametrów, które mogą być zapisane i „rozumiane” przez komputer. Duża część pracy z Gazebo jest więc związana z budowaniem, przechowywaniem, uruchamianiem i używaniem modeli.

Dlatego pierwszym etapem projektu jest stworzenie środowiska, które będzie wykorzystywane w dalszych częściach pracy. Niezbędny jest zatem model robota mobilnego oraz model otoczenia, w którym ów robot będzie wykonywał nałożone mu zadania.

3.1. Robot

Za radą promotora zdecydowano się na użycie w symulacji robota mobilnego Pioneer 3-DX firmy *Adept MobileRobots*. Sprzęt ten jest dostępny w laboratorium instytutu, dzięki czemu projekt będzie mógł być wykorzystywany i rozwijany w przyszłości.

3.1.1. Opis rzeczywistego robota

Pioneer 3-DX jest jednym z najpopularniejszych na świecie robotów mobilnych wykorzystywanych w celach naukowych i badawczych. To kompaktowa i wytrzymała platforma, która mimo relatywnie niewielkich rozmiarów posiada aż 23 kilogramową ładowność (rys. 3.1). Dzięki zastosowaniu dużych, 19 centymetrowych kół, mocnych silników i napędu różnicowego osiąga prędkość do 1,2 m/s, pokonuje niskie progi oraz wspina się na rampy dla niepełnosprawnych.

Rys. 3.1. Robot mobilny Pioneer 3-DX [3]

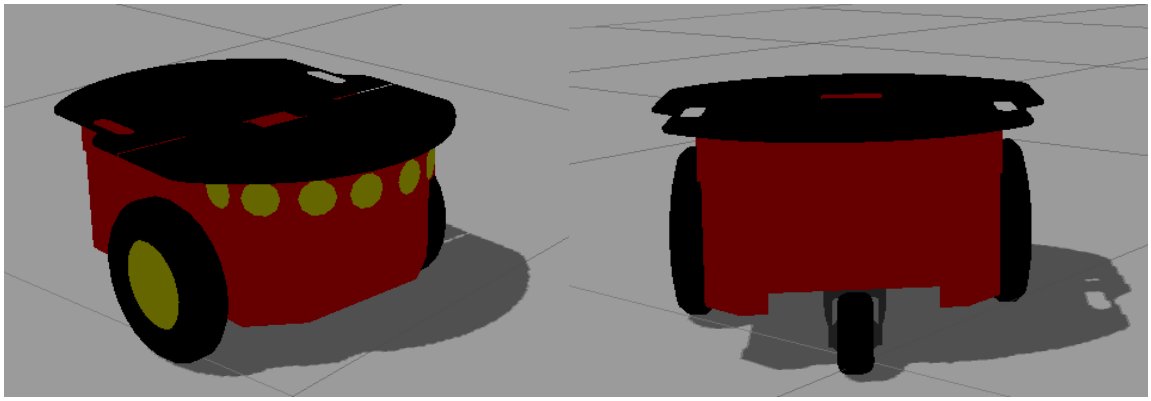
Ponadto robot posiada wbudowany, dedykowany kontroler ruchu, który automatycznie przeprowadza kontrolę prędkości i zapewnia informacje o stanie robota: położenie w przestrzeni (x, y, theta), poziom naładowania baterii (3 akumulatory 9 Ah typu hot-swap) oraz zakres czułości

ultradźwiękowych sonarów (8 sztuk z przodu). Robot jest gotowy do pracy po podłączeniu zewnętrznego laptopa przez złącze USB lub wbudowaniu w bazę opcjonalnego, wewnętrznego komputera.

Uniwersalność platformy oraz dostępność wielu dodatkowych akcesoriów sprawia, że Pioneer 3-DX jest wykorzystywany do różnorakich zadań, takich jak: mapowanie (np. magazynów, hangarów), lokalizacja, monitoring, rekonesans w terenie, autonomiczna nawigacja, i wielu innych.

3.1.2. Wirtualny model

Tworzenie modelu robota od podstaw jest skomplikowane i, jeśli chcemy czegoś więcej od prostych brył geometrycznych, wymaga korzystania z zewnętrznych programów graficznych. Na szczęście można tego uniknąć, a to dzięki wsparciu producentów oprogramowania i ogromnej społeczności ROS. Modele popularnych robotów znajdują się w bibliotece programu Gazebo, skąd można je z łatwością pobrać i następnie używać w symulatorze. Ich minusem jest jednak to, że są napisane w formacie SDF - niekompatybilnym z wizualizatorem *RViz*, który jest częścią ROS. Aby zobaczyć naszego robota w tym programie, musimy użyć pliku w formacie URDF. Za użyciem tego drugiego formatu przemawia też fakt, że potrzebuje on jedynie dodania kilku linijek kodu, aby mógł być uruchomiony w Gazebo.



Rys. 3.2. Model robota Pioneer 3-DX w symulatorze Gazebo [źródło: własne]

Po przeszukaniu strony *wiki.ros.org* znaleziono takie pliki na przykład w paczce *p2os_urdf*, która jest częścią pakietu *P2OS*, jak i w kilku innych paczkach zawartych w rozmaitych projektach. Po przeanalizowaniu dostępnych możliwości zdecydowano się skorzystać ze zmodyfikowanego modelu (rys. 3.2) znajdującego się w pakiecie *ua_ros_p3dx*, której autor poprawił kompatybilność kodu URDF z symulatorem. Model jest zapisany w dwóch plikach: *pioneer3dx.xacro* oraz *pioneer3dx_wheel.xacro*, które skopiowano do katalogu /p3dx/urdf/.

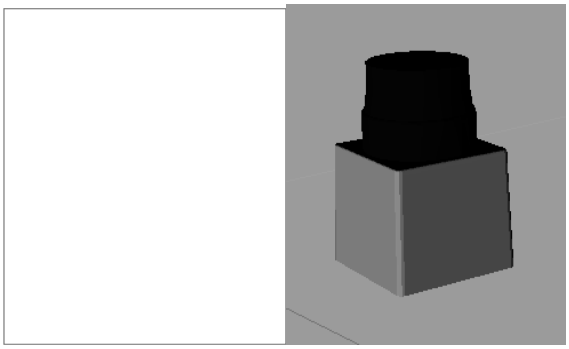
3.2. Akcesoria

Sama baza robota to za mało, by można było przeprowadzać bardziej skomplikowane symulacje. Dlatego zdecydowano się na dołączenie modeli przydatnych akcesoriów: czujnika laserowego wykorzystywanego do nawigacji oraz prostej kamery do celów wizyjnych.

3.2.1. Czujnik laserowy

SICK LMS-100 to popularny czujnik laserowy przeznaczony do użytku wewnątrz budynków. Jest relatywnie niewielki (105 mm x 102 mm x 152 mm) oraz lekki (1.1 kg). Kolejne zalety tego urządzenia to niski pobór mocy (max 20 W) i szerokie pole widzenia – do 270 stopni (w zależności od umiejscowienia czujnika na robocie). Na minus w porównaniu do większych modeli możemy zaliczyć niższą częstotliwość skanowania (25 Hz lub 50 Hz) i zwiększone błędy pomiaru.

Uproszczony model takiego czujnika (rys. 3.4) dostępny jest w bibliotece programu Gazebo. Został on pobrany i dołączony do podstawowego modelu robota w pliku *pioneer3dx.xacro.* Z uwagi na specyfikę środowiska pracy (niskie przedmioty mogące znajdować się na podłodze w mieszkaniu) zdecydowano się umiejscowić go na niewielkiej wysokości z przodu robota.

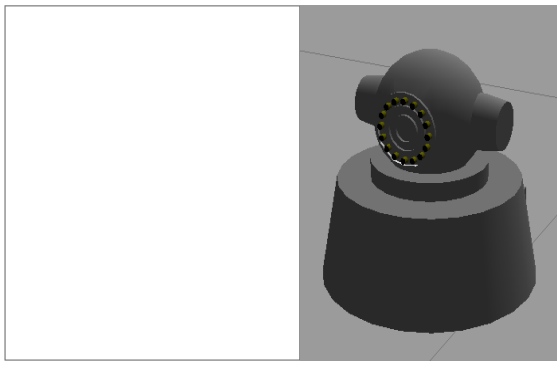


Rys. 3.4. Czujnik LMS-100 (po lewej) oraz jego uproszczony model w Gazebo (po prawej) [źródło: materiały producenta, własne]

3.2.2. Kamera

Obrazy rejestrowane przez kamerę umieszczoną na robocie mogą służyć do wielu celów: teleoperacji, widzenia maszynowego, analizy obrazu czy nawigacji. Równie duża jest ilość możliwego do zastosowania wraz z robotem Pioneer 3-DX oraz systemem ROS sprzętu: od prostych kamer internetowych oraz sterowanych programowo kamer uchylno-obrotowych (pan-tilt-zoom cameras; rys. 3.5), poprzez systemy 3D i kamery stereo, aż do zaawansowanych czujników typu *Kinect.* Każdy taki system wizyjny składa się z odpowiedniego oprogramowania służącego do kalibrowania sprzętu, kontroli nad nim oraz pozwalającego na obróbkę obrazu.

Niniejszy projekt nie skupia się na zaawansowanych systemach wizyjnych, dlatego na jego potrzeby zdecydowano się użyć prostej kamery bezprzewodowej, której parametry, takie jak rozdzielczość czy kąty widzenia, będą standardowymi wartościami dla sprzętu tej klasy. Przykładowy model 3D bezprzewodowej kamery dołączono do robota w pliku *pioneer3dx.xacro*.



Rys. 3.5. Przykładowa kamera uchylno-obrotowa umieszczona na rzeczywistym robocie (z lewej) i model prostej kamery w Gazebo (z prawej) [źródło: materiały producenta, własne]

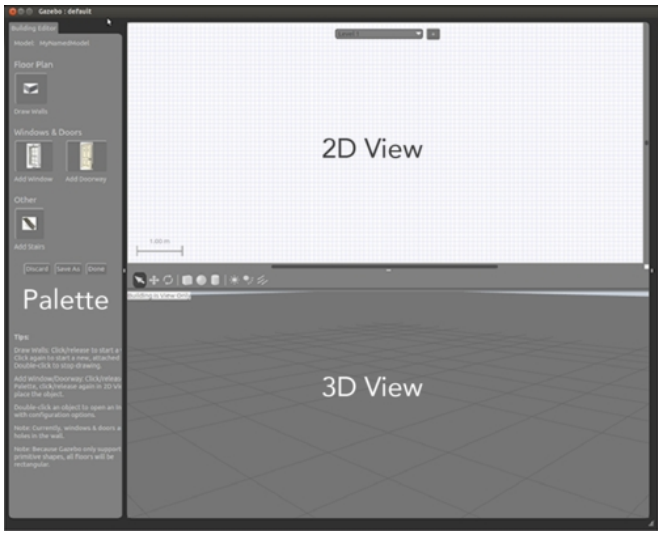
3.3. Środowisko pracy robota

Założeniem projektu jest symulacja robota mobilnego w środowisku domowym. Za takie można uznać dowolny budynek składający się z kilku pomieszczeń, wśród których można wyróżnić kuchnię, łazienkę, pokój, korytarz itp. Pomieszczenia te powinny być ze sobą połączone oraz wypełnione stosownymi obiektami, wśród których będzie pracował robot.

3.3.1. Edytor budowania

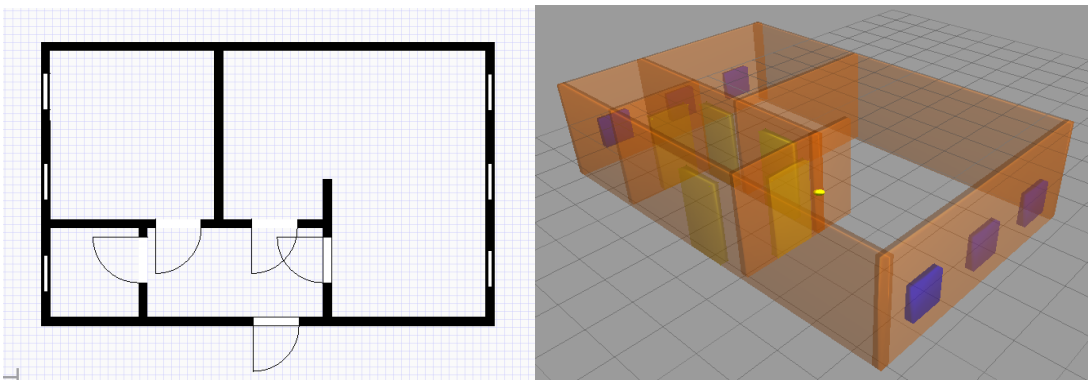
Gazebo ma wbudowany prosty edytor (rys. 3.6), który umożliwia tworzenie modeli budynków bez konieczności pracochłonnego pisania kodu. Korzystamy za to z myszki komputerowej i palety po lewej stronie okna programu. W widoku 2D u góry ekranu rysujemy ściany tworząc plan pomieszczeń, a następnie dodajemy drzwi, okna oraz schody, dzięki czemu nasz budynek może być wielokondygnacyjny. Efekty widzimy w czasie rzeczywistym w widoku 3D poniżej. Każdy dodany element możemy edytować, klikając na nim dwukrotnie lewym przyciskiem myszy. Ukończony budynek zapisujemy w formacie SDF, aby dało się go użyć jako modelu w Gazebo. Niestety, sam edytor ma sporo ograniczeń, przez które okna i drzwi to tylko proste otwory w ścianach, a tworzenie bardziej skomplikowanych budynków nie jest ani łatwe, ani intuicyjne. Dodatkowo nie ma możliwości wczytania pliku do edytora, więc jeśli zapomnimy dodać do naszego budynku np. drzwi, a wyjdziemy z programu, będziemy musieli zacząć od początku albo zagłębić się w kod zapisanego pliku SDF i tam dokonać zmian. Mimo wszystko, dzięki temu

edytorowi zaoszczędzono sporo pracy podczas tworzenia modelu budynku.



Rys. 3.6. Interfejs edytora budowania [2]

Na potrzeby projektu stworzono plan prostego budynku (rys. 3.7) składającego się z korytarza, łazienki, sypialni oraz przestrzeni mieszkalnej zawierającej aneks kuchenny, jadalnię i pokój dzienny. Całość zapisano w katalogu domowym do pliku *house.sdf*.



Rys. 3.7. Plany budynku w edytorze budowania: 2D (z lewej) oraz 3D (z prawej) [źródło: własne]

3.3.2. Modele statyczne i dynamiczne

Aby jak najbardziej przybliżyć symulację do rzeczywistości konieczne jest wypełnienie środowiska pracy robota względnie wieloma obiektami. Są to z głównie modele statyczne, z którymi robot nie będzie wchodził w interakcje, a jedynie uwzględniał ich położenie podczas poruszania się po mieszkaniu. Ponadto przyda się kilka modeli dynamicznych (pudełka, puszki, butelki), które będą mogły być popchnięte lub przewrócone przez poruszającego się robota dzięki posiadaniu dodatkowych właściwości: masy, inercji i współczynników tarcia.

W celu dodania przedmiotów do wcześniej utworzonego budynku, uruchomiono Gazebo wczytując jednocześnie plik *house.sdf* z katalogu domowego komendą:

```
gazebo house.sdf
```

Dodawanie obiektów następuje po przejściu do zakładki *[Insert]*, wybraniu modelu z bazy i umieszczeniu go za pomocą myszki w symulowanym świecie. Biblioteka jest dostępna online, jednak podczas pierwszego użycia modelu zostaje on automatycznie pobrany na dysk i zapisany w folderze /*.gazebo/models/* znajdującym się w katalogu domowym użytkownika. Po zapoznaniu się z bazą dostępnych modeli stwierdzono, że nie jest ona wystarczająca na potrzeby projektu. Liczba obiektów jest mała, a w dodatku większość z nich jest przeznaczona do symulacji środowisk zewnętrznych.

Rozwiązaniem tego problemu okazał się fakt, że Gazebo obsługuje trójwymiarowe modele w formacie COLLADA (*.dae*). Ogromną i darmową bazą takich plików jest serwis *Google SketchUp* [5]. Za jego pomocą wyszukano kilka modeli niezbędnych do umeblowania budynku. Podczas ich wybierania kierowano się subiektywnymi względami wizualnymi i liczbą wielokątów, z których się składają. Mniejsza liczba wielokątów obniża jakość obiektu 3D, w zamian za to (co uznano za priorytet) przyspiesza jego generowanie w symulatorze i zmniejsza zużycie zasobów. Poniżej przedstawiono przykładowy proces dodawania do biblioteki Gazebo statycznego modelu kanapy:

1. Pobrano i rozpakowano plik *sofa.kmz*.

2. W folderze /p3dx/gazebo_models/ utworzono folder /*sofa/*, a w nim folder /*meshes/*, do którego skopiowano całą zawartość uprzednio rozpakowanego folderu *sofa.kmz_FILES*.

3. W folderze */sofa/* utworzono plik *model.config* o zawartości:

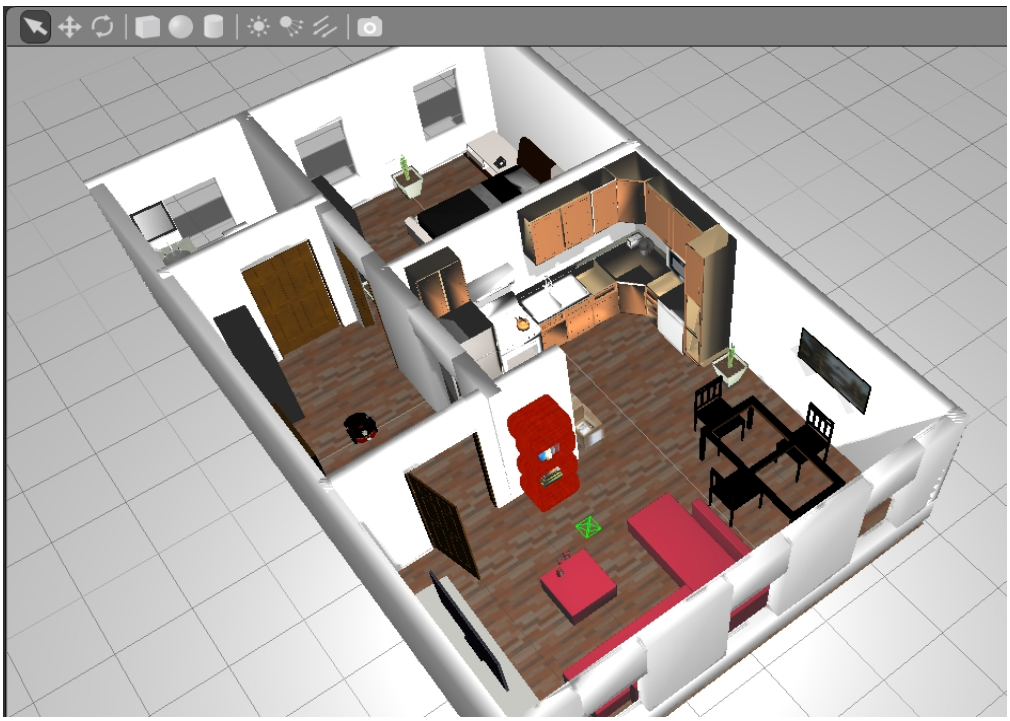
```
<?xml version=”1.0”?>
<model>
<name>Sofa</name>
<sdf version=”1.4”>model.sdf</sdf>
</model>
```

4. W tym samym folderze utworzono plik *model.sdf* o zawartości:

```
<?xml version=”1.0”?>
<sdf version=”1.4”>
  <model name=”Sofa”>
    <static>true</static>   <!-- Obiekt statyczny -->
```

```
      <link name="link">
        <collision name="collision">   <!-- Model kolizyjny dla obiektów statycznych
                                            może być taki sam jak wizualny -->
          <geometry>
            <mesh>
              <uri>model://sofa/meshes/untitled.dae</uri>   <!-- Scieżka dostępu do
                                                  pliku COLLADA z modelem 3D -->
              <scale>0.8 0.8 0.8</scale>   <!-- Skalowanie obiektu jeśli konieczne -->
            </mesh>
          </geometry>
        </collision>
        <visual name="visual">   <!-- Model wizualny -->
          <geometry>
            <mesh>
              <uri>model://sofa/meshes/untitled.dae</uri>
              <scale>0.8 0.8 0.8</scale>
            </mesh>
          </geometry>
        </visual>
      </link>
    </model>
  </sdf>
```

Kiedy wszystkie potrzebne modele 3D znalazły się w bazie, ponownie uruchomiono model budynku w Gazebo i wypełniono go obiektami. Tak przygotowany świat (rys. 3.8) zapisano do pliku *p3dx.world* i umieszczono w katalogu /p3dx/worlds/.



Rys. 3.8. Mieszkanie – środowisko pracy robota [źródło: własne]

4. Implementacja pakietów

W tym rozdziale zebrano, opisano i wstępnie skonfigurowano wszystkie pakiety ROS wykorzystywane w projekcie. Z uwagi na to, że ROS jest środowiskiem wielu programistów, często znaleźć można pakiety realizujące podobne lub takie same funkcje. Dlatego skupiono się głównie na tych zawartych w pełnej instalacji ROS Hydro. Dzięki temu nie ma potrzeby dodatkowego ich pobierania i dołączania do projektu. Ponadto zawsze dysponują one pełną dokumentacją, a czasami też przystępnymi samouczkami pomagającymi w zrozumieniu zasad ich działania. Konfigurację pakietów przeprowadzono najczęściej na podstawie dokumentacji, z myślą o dostosowaniu parametrów w trakcie testów symulacji.

4.1. Podstawowy sterownik robota

Robot mobilny Pioneer 3-DX, tak jak pozostałe roboty firmy *Adept MobileRobots*, używa protokołu komunikacyjnego ARIA. W celu uzyskania kompatybilności z ROS możemy skorzystać z dwóch pakietów: ROSARIA lub P2OS. Podczas korzystania z pakietu ROSARIA cała konfiguracja robota i komunikacja z nim jest wykonywana poprzez protokół ARIA, którego aktualna wersja jest pobierana automatycznie podczas budowania pakietu. Pakiet P2OS zawiera z kolei własną implementację protokołu i komunikuje się bezpośrednio z robotem. Oba pakiety dostarczają interfejsy do sterowania robotem i szacowania jego pozycji, a także używają odpowiednich węzłów ROS do obsługi czujników dołączonych do bazy robota.

W przypadku symulacji w programie Gazebo rolę podstawowego sterownika przejmuje właśnie symulator, czyli pakiet *gazebo_ros*. Interfejsy są dostarczane poprzez specjalne wtyczki – pluginy. Są to fragmenty kodu C++, który jest kompilowany i dołączany do symulacji.

W celu konfiguracji modelu URDF na potrzeby symulatora Gazebo (rozdział 2.3.1) utworzono plik *pioneer3dx.gazebo* i zapisano go w folderze /p3dx/urdf/. W pliku tym znajdują się parametry i właściwości robota, których nie ma w pliku URDF, a są niezbędne do symulacji w Gazebo. Aby kolory elementów robota były poprawnie wyświetlane, należy je tutaj zadeklarować. Dla przykładu: baza robota jest czerwona dzięki poniższemu fragmentowi kodu:

```
<gazebo reference="base_link">
  <material>Gazebo/Red</material>
</gazebo>
```

Koła robota wymagają podania dodatkowych współczynników. Przykładowo, za pomocą poniższego kodu ustawiono kolor, współczynniki tarcia oraz sztywność koła środkowego

(najmniejszego):

```
<gazebo reference="center_wheel">
  <material>Gazebo/Black</material>
  <mu1>10.0</mu1>
  <mu2>10.0</mu2>
  <kp>1000000.0</kp>
  <kd>1.0</kd>
</gazebo>
```

Dzięki odpowiedniemu skonfigurowaniu wszystkich kół, robot mobilny podczas symulacji będzie przyspieszał bez poślizgu, a po zaniku sygnału sterującego zatrzyma się dzięki tarciu.

W pliku *pioneer3dx.gazebo* znajdują się także implementacje wtyczek. Spośród wszystkich dostępnych (spis dostępny na rys. 2.3) wybrano i opisano te użyte w projekcie. Nazwy wtyczek pochodzą od klas, przykładowo *DiffDrive* pochodzi od klasy *GazeboRosDiffDrive* i znajduje się w /*gazebo_plugins/src/gazebo_ros_diff_drive.cpp*.

4.1.1. Wtyczka *DiffDrive*

Wtyczka *DiffDrive* zapewnia podstawowy kontroler dla robotów z napędem różnicowym. Jest wtyczką typu „model”, dlatego do poprawnego działania wymaga dobrze skonfigurowanego modelu robota mobilnego.

```
<gazebo>
<plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>100</updateRate>
  <leftJoint>base_right_wheel_joint</leftJoint>
  <rightJoint>base_left_wheel_joint</rightJoint>
  <wheelSeparation>0.39</wheelSeparation>
  <wheelDiameter>0.15</wheelDiameter>
  <torque>5</torque>
  <commandTopic>cmd_vel</commandTopic>
  <odometryTopic>odom</odometryTopic>
  <odometryFrame>odom</odometryFrame>
  <robotBaseFrame>base_link</robotBaseFrame>
</plugin>
</gazebo>
```

Parametry charakterystyczne dla danego modelu robota to rozmiar kół i odstęp między nimi. Tarcie ustawiono na wartość domyślną. Aby wtyczka pracowała poprawnie, ważne jest odpowiednie przypisanie złączy kół oraz układów współrzędnych. Kontroler przetwarza dane z tematu *cmd_vel* na ruch kół robota.

4.1.2. Wtyczka *P3D*

Wtyczka *P3D* odpowiada za przekazywanie pozycji dowolnego symulowanego elementu poprzez publikowanie danych odometrycznych (wiadomości *odom*) w temacie ROS.

```
<gazebo>
<plugin name="p3d_base_controller" filename="libgazebo_ros_p3d.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>100.0</updateRate>
  <bodyName>base_link</bodyName>
  <topicName>base_pose_ground_truth</topicName>
  <gaussianNoise>0.01</gaussianNoise>
  <frameName>map</frameName>
  <!-- initialize odometry for fake localization -->
  <xyzOffsets>0 0 0</xyzOffsets>
  <rpyOffsets>0 0 0</rpyOffsets>
</plugin>
</gazebo>
```

Jest to wtyczka typu „model”. W tym przypadku temat *base_pose_ground_truth* publikuje dane odometryczne informujące o położeniu bazy robota.

4.1.3. Wtyczka *Laser*

Wtyczka *Laser* symuluje czujnik laserowy poprzez przesyłanie wiadomości *LaserScan* w temacie ROS.

```
<gazebo reference="lms100">
<sensor type="ray" name="head_hokuyo_sensor">
  <pose>0 0 0 0 0 0</pose>
  <visualize>true</visualize>
  <update_rate>50</update_rate>
  <ray>
    <scan>
      <horizontal>
        <samples>360</samples>
        <resolution>1</resolution>
        <min_angle>-2.3562</min_angle>
        <max_angle>2.3562</max_angle>
      </horizontal>
    </scan>
    <range>
      <min>0.5</min>
      <max>20.0</max>
      <resolution>0.01</resolution>
    </range>
    <noise>
      <type>gaussian</type>
```

```
        <!-- Noise parameters based on published spec for Hokuyo laser achieving
        "+-30mm" accuracy at range < 10m. A mean of 0.0m and stddev of 0.01m will
        put 99.7% of samples within 0.03m of the true reading. -->
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
  <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
    <topicName>laserscan</topicName>
    <frameName>lms100</frameName>
  </plugin>
  </sensor>
  </gazebo>
```

Jest to wtyczka typu „sensor” i musi być dołączona do odpowiedniego członu robota – w tym wypadku *lms100*. Parametr *visualize* ustawiony na *true* sprawia, że aktualny obszar skanowania jest widoczny w Gazebo w postaci półprzeźroczystego promienia lasera. Częstotliwość odświeżania to 50 Hz, maksymalna dla tego modelu czujnika. W części <scan> ustawiono pole widzenia: 270 stopni daje 135 stopni w obie strony, co po przeliczeniu na radiany wynosi 2,3562. W sekcji <range> zdefiniowano zasięg lasera – od 0,5 do 20 metrów w rozdzielczości 1 mm. Dalej znajduje się standardowe określenie szumu dla laserów o niedokładności pomiaru +-30 mm, czyli odpowiednie dla modelu LMS100. Wtyczka publikuje odczyty lasera w temacie *laserscan*. Ważnym parametrem jest też <frameName>, który określa układ współrzędnych lasera, co zostanie wykorzystane w nawigacji przy wyznaczaniu położenia lasera względem bazy robota.

4.1.4. Wtyczka *Camera*

Wtyczka *Camera* dostarcza interfejs do symulacji kamery poprzez publikowanie wiadomości *CameraInfo* oraz *Image* w temacie ROS.

```
  <gazebo reference="camera_link">
    <sensor type="camera" name="camera1">
      <update_rate>30.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>800</width>
          <height>800</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
          <far>300</far>
        </clip>
```

```
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>camera</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera_link</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

Wtyczka ta jest typu „sensor” i została dołączona do członu *camera_link*. Parametry znajdujące się w obszarze <camera>, czyli pole widzenia kamery, rozmiar obrazu w pikselach i jego format, a także zasięg widzenia oraz szum, ustawiono na wartości standardowe polecane do celów symulacyjnych. Ważne parametry to nazwa kamery i nazwy publikowanych tematów.

4.2. Pakiety do wyznaczania transformacji układów współrzędnych

Każdy system w robotyce składa się z wielu układów współrzędnych (bazowego, podstawy robota oraz poszczególnych członów), których położenie względem siebie jest zmienne w czasie. W systemie ROS zagadnieniem tym zajmuje się pakiet *tf* będący częścią biblioteki *geometry*, który publikuje drzewo transformacji układów współrzędnych. Diagram ten określa przesunięcia w zakresie translacji i rotacji pomiędzy poszczególnymi układami.

Najprostszą metodą określenia transformacji układów jest utworzenie pliku z kodem, w którym opisujemy przesunięcia za pomocą wektorów i kwaternionów, a następnie skompilowanie go i uruchomienie jako węzeł. Jednak kiedy robot składa się z wielu członów, manualne określenie położenia wszystkich układów współrzędnych jest pracochłonne. W przypadku, kiedy posiadamy plik ze starannie odwzorowanym modelem robota, staje się to zupełnie zbędne. Wtedy bowiem z pomocą przychodzi pakiet *robot_state_publisher*, który przekazuje pozycję robota w przestrzeni

trójwymiarowej do biblioteki *tf*.

robot_state_publisher

Subskrybowane tematy:
joint_states *[sensor_msgs/JointState] – informacje o pozycjach złączy robota.*

Parametry:
robot_description *[urdf map] – opis robota w formacie URDF.*

Węzeł ten subskrybuje pozycje złączy robota, więc muszą być one nadawane jako wiadomość odpowiedniego typu. Zadanie to spełnia pakiet *joint_state_publisher* zawierający węzeł o tej samej nazwie. Czyta on parametr *robot_description*, znajduje wszystkie ruchome złącza i publikuje wiadomość z ich definicją.

joint_state_publisher

Publikowane tematy:
joint_states *[sensor_msgs/JointState] – informacje o pozycjach złączy robota.*

Parametry:
robot_description *[urdf map] – opis robota w formacie URDF.*

Oba węzły wymagają dostarczenia pliku URDF, czyli inaczej mówiąc modelu kinematycznego robota. Najlepszym sposobem na to jest załadowanie pliku na Serwer Parametrów (rozdział 2.3.3), dzięki czemu model będzie dostępny dla wszystkich węzłów, które go potrzebują.

Plik *tf.launch* uruchamiający oba węzły zapisano w folderze /p3dx/launch/tools/:

```
<launch>
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
</launch>
```

Podsumowując: posiadając model kinematyczny robota z pliku URDF oraz subskrybując pozycje wszystkich jego złączy nadawane przez węzeł *joint_state_publisher,* węzeł *robot_state_publisher* potrafi obliczać i nadawać do biblioteki *tf* pozycję każdego członu robota w przestrzeni trójwymiarowej.

4.3. Pakiet do manualnego sterowania robotem

Podstawową funkcjonalnością robota mobilnego powinna być możliwość manualnego sterowania jego ruchem. Polega to na wysyłaniu wiadomości do tematu *cmd_vel*, które następnie zostają zamienione na ruch kół przez sterownik robota. Najprostszą, ale na pewno nie najwygodniejszą opcją jest wysyłanie komend za pomocą poleceń wpisywanych w konsoli. Dużo lepiej nadają się do tego celu kontrolery: klawiatury, myszki, joysticki lub pady. Programy do ich obsługi znajdują się na przykład w pakiecie *turtleop_teleop*.

W wypadku przeprowadzania symulacji, kiedy znajdujemy się cały czas przed komputerem, sterowanie za pomocą klawiatury uznano za najlepsze i wystarczające. Dlatego zamiast pobierania całego pakietu *turtleop_teleop* zdecydowano się na wykorzystanie jedynie pliku *turtlebot_teleop_key* z katalogu /scripts/. Plik ten zawiera kod w języku Python odpowiadający za pobieranie danych z klawiatury i przekazywanie odpowiednich informacji do tematu *cmd_vel*. Po przeprowadzeniu kilku zmian dostosowujących plik do wymagań projektu zapisano go w katalogu /p3dx/scripts/ pod nazwą *p3dx_key_control*. Następnie do pliku *CmakeLists.txt* dodano następujący wpis:

```
install(PROGRAMS
  scripts/p3dx_key_control
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Dzięki temu plik zostanie skompilowany podczas budowania środowiska *Catkin*, jego uruchomienie będzie zaś możliwe za pomocą poniższego pliku *.launch*:

```
<launch>
  <node pkg="p3dx" type="p3dx_key_control" name="p3dx_key_control"  output="screen"/>
</launch>
```

Plik ten pod nazwą *wsad.launch* zapisano w folderze /p3dx/launch/tools/.

4.4. Pakiet do tworzenia map pomieszczeń

Korzystając z pakietu *gmapping* można stworzyć dwuwymiarową siatkę zajętości obszaru (na przykład plan pomieszczeń budynku), wykorzystując dane z czujnika laserowego i pozycje zebrane przez robota mobilnego. Pakiet ten zawiera węzeł ROS o nazwie *slam_gmapping*, który odpowiada za jednoczesną lokalizację i mapowanie (*Simultaneous Localization and Mapping*).

slam_gmapping

Subskrybowane tematy:

tf *[tf/tfMessage] – transformacje układów współrzędnych;*

scan *[sensor_msgs/Laserscan] – skany, z których powstaje mapa;*

Publikowane tematy:

map *[nav_msgs/OccupancyGrid] – siatka 2D, w której każda komórka zawiera prawdopodobieństwo zajętości;*

map_metadata *[nav_msgs/MapMetaData] - zawiera informacje na temat siatki 2D (np. rozmiar, rozdzielczość, pozycja).*

Węzeł wymaga dostarczenia transformacji: czujnik laserowy -> baza robota. Jest to wartość stała, podawana przez węzeł *robot_state_publisher.* Druga wymagana transformacja to: baza robota -> układ bazowy (odometria), którą zazwyczaj zapewnia sterownik robota (w tym przypadku pluginy Gazebo). Dostarczana przez węzeł jest za to transformacja: mapą -> układ bazowy. Dzięki tym transformacjom znana jest aktualna szacunkowa pozycja robota.

Opis wszystkich parametrów węzła oraz ich proponowane wartości można znaleźć na stronie *wiki.ros.org/gmapping*. Takie domyślne wartości zapisano do pliku *gmapping.yaml*, który następnie umieszczono w folderze /p3dx/config/. W folderze /p3dx/launch/nav/ utworzono natomiast plik *gmapping.launch* o następującej zawartości:

```
<launch>
  <node pkg="gmapping" type="slam_gmapping" name="gmapping" args="scan:=laserscan" output="screen">
    <rosparam file="$(find p3dx)/config/gmapping.yaml" command="load" />
  </node>
</launch>
```

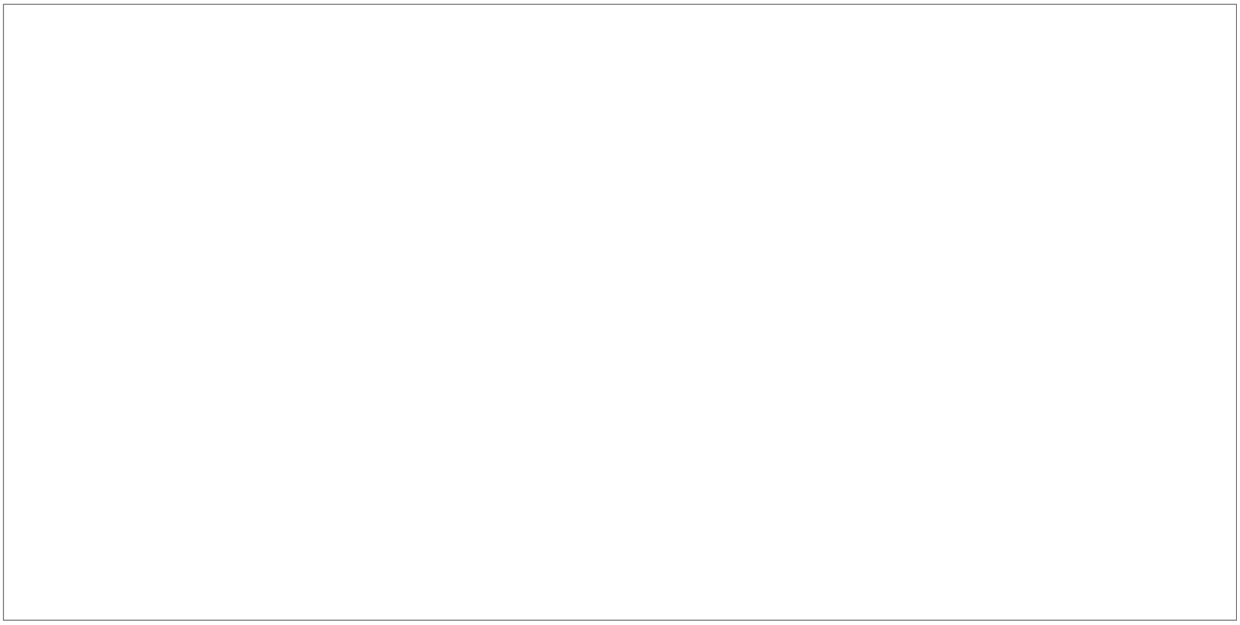
Plik ten uruchamia węzeł *gmapping* z parametrami zapisanymi w pliku *gmapping.yaml*.

4.5. Pakiety nawigacyjne

Meta-pakiet ROS Navigation zawiera wszystko, co niezbędne w nawigacji 2D. Ogólna zasada działania polega na pobieraniu danych z czujnika laserowego oraz wiadomości odometrycznych i wysyłaniu poleceń zmiany prędkości do bazy robota. Jest skonstruowany tak, aby mógł być wykorzystywany w jak największej liczbie robotów, jednak istnieje kilka założeń, które muszą zostać spełnione. Po pierwsze, robot musi być kompatybilny z ROS oraz posiadać pełne drzewo transformacji *(tf)*. Powinien ponadto posiadać napęd różnicowy lub być holonomiczny, ponieważ

polecenia zmiany prędkości są wysyłane w formie prędkości x, y, theta. Najlepiej gdyby podstawa robota była bliska kwadratowej lub okrągłej, bo chociaż pakiety zadziałają niezależnie od kształtu robota, to mogą wystąpić problemy podczas pokonywania wąskich przejść, jak na przykład drzwi. Do tworzenia mapy i lokalizacji wymagany jest czujnik laserowy zamontowany na robocie, który publikuje dane zgodnie z odpowiednimi typami wiadomości ROS.

Nawigacja do działania wymaga robota mobilnego i pakietu skonfigurowanego według schematu na rys. 4.1. Białe elementy są tutaj niezbędne, a szare opcjonalne, jednak wszystkie zostały zaimplementowane w meta-pakiecie nawigacyjnym. Inaczej jest z elementami widocznymi na niebiesko, które są specyficzne dla każdego typu robota i muszą być utworzone oraz poprawnie skonfigurowane.



Rys. 4.1. Schemat konfiguracji meta-pakietu nawigacyjnego [1]

4.5.1. Węzeł *map_server*

Pakiet *map_server* zawiera węzeł ROS o takiej samej nazwie, który czyta mapy z dysku i oferuje je za pośrednictwem tematów i serwisów ROS. Mapy wykorzystywane przez węzeł powinny być zapisane za pomocą dwóch plików: tekstowego i graficznego. Plik *.yaml* koduje opcje mapy i nazwę pliku graficznego, a ten opisuje stan zajętości komórek za pomocą kolorów: białe są wolne, czarne zajęte, a szare nieznane. Obecna wersja węzła konwertuje kolory na wartości: wolne (0), zajęte (100) lub nieznane (-1). Mapa może być pobierana zarówno poprzez subskrybcję tematów jak i za pośrednictwem serwisu *static_map*.

map_server

Publikowane tematy:

map *[nav_msgs/OccupancyGrid] – siatka 2D, w której każda komórka zawiera prawdopodobieństwo zajętości;*

map_metadata *[nav_msgs/MapMetaData] - zawiera informacje na temat siatki 2D (np. rozmiar, rozdzielczość, pozycja).*

Serwisy:

static_map *[nav_msgs/GetMap]*

Drugim ważnym elementem pakietu *map_server* jest narzędzie *map_saver* pozwalające na zapisywanie map tworzonych za pomocą procesu *gmapping*. Wykorzystanie tego narzędzia wygląda następująco:

```
rosrun map_server map_saver -f moja_mapa
```

Powyższa komenda tworzy pliki *moja_mapa.yaml* i *moja_mapa.pgm* w katalogu domowym użytkownika.

W katalogu /p3dx/launch/nav/ utworzono plik *map_server.launch* o następującej zawartości:

```
<launch>
  <node name="map_server" pkg="map_server" type="map_server" args="$(find p3dx)/config/p3dx_map.yaml"/>
</launch>
```

Będzie on uruchamiał węzeł *map_server* wczytujący mapę z pliku *p3dx_map.yaml* znajdującego się w folderze /p3dx/config/.

4.5.2. Węzeł *move_base*

Głównym elementem meta-pakietu nawigacyjnego jest znajdujący się w nim węzeł *move_base*.

move_base

Subskrybowane tematy:

move_base_simple/goal *[geometry_msgs/PoseStamped] – punkt docelowy nawigacji;*

odom *[nav_msgs/Odometry] – dane odometryczne służące do przekazywania aktualnej prędkości robota do planowania lokalnego.*

Publikowane tematy:

cmd_vel *[geometry_msgs/Twist] – strumień komend prędkości wysyłany do bazy robota.*

Tematy publikowane przez pakiety składowe, służące do celów wizualizacyjnych:
plan *[nav_msgs/Path] – ostatnia trajektoria obliczona przez planer globalny;*
global_plan *[nav_msgs/Path] – część trasy globalnej, którą stara się podążać planer lokalny;*
local_plan *[nav_msgs/Path] – miejscowa trajektoria;*
cost_cloud *[sensor_msgs/PointCloud2] – siatka służąca do planowania kosztów;*

Węzeł ten zawiera składowe, które pochodzą z innych pakietów nawigacyjnych. Jednym z takich pakietów jest *nav_core*, który zawiera kluczowe interfejsy dla planowania globalnego i lokalnego oraz funkcji *recovery*. To, jakie pakiety są wykorzystywane, zależy w dużej mierze od konfiguracji parametrów węzła *move_base*. Ta znajduje się w pliku *move_base.yaml* w folderze /p3dx/config/. Poniżej zamieszczono i opisano wszystkie elementy tego pliku wraz z konfiguracją pakietów składowych.

PLANER GLOBALNY

```
base_global_planner: navfn/NavfnROS
NavfnROS:
  allow_unknown:        false
  planner_window_x:      0.0
  planner_window_y:      0.0
  default_tolerance:    0.0
  visualize_potential:  false
```

W pierwszej linijce wybrany zostaje pakiet *Navfn* obsługujący globalne planowanie trajektorii ruchu robota. Działa on na zasadzie obliczania (za pomocą algorytmu Dijkstry) najmniejszego kosztu pokonania trasy od punktu startowego do końcowego. Dalej mamy parametry tego pakietu: nie pozwalamy na tworzenie trajektorii przebiegających przez obszary nieznane na mapie, nie korzystamy z opcjonalnego okna ograniczającego mapę kosztów, ustawiamy zerową tolerancję na punkt docelowy i wyłączamy chmurę punktów pokazującą obliczenia potencjału dokonywane przez *Navfn*.

PLANER LOKALNY

Za planowanie lokalne odpowiedzialny jest pakiet *base_local_planner*, który implementuje metodę Dynamicznych Okien (algorytm unikania kolizji uwzględniający ograniczenia kinematyczne robota). Korzystając z mapy kosztów oraz planu globalnego tworzy on lokalną trajektorię ruchu i wysyła polecenia zmiany prędkości do bazy robota. Podczas ruchu planer lokalny oblicza wokół robota funkcję kosztu reprezentowaną przez siatkę, która określa koszt przechodzenia przez komórki tej siatki. Siatka ta posiada rozmiar lokalnej mapy kosztów.

Planowanie lokalne jest konfigurowane dzięki kilku kategoriom parametrów.

```
base_local_planner: base_local_planner/TrajectoryPlannerROS
TrajectoryPlannerROS:
# Robot Configuration Parameters
  acc_lim_x:              2.5
  acc_lim_y:              0.0
  acc_lim_theta:          3.2
  max_vel_x:              1.2
  min_vel_x:              0.1
  max_vel_y:              0.0
  min_vel_y:              0.0
  max_vel_theta:          5.236
  min_vel_theta:          -5.236
  min_in_place_vel_theta: 0.4
  escape_vel:             -0.1
  holonomic_robot:        false
```

Pierwsza część parametrów określa konfigurację robota: zakres prędkości bazy oraz holonomiczność. Pioneer 3-DX nie jest robotem holonomicznym, dlatego prędkości i przyspieszenia w osi 'Y' są równe '0'. Maksymalna prędkość w osi 'X' to 1.2 m/s, a w osi 'theta' (prędkość obrotu) – 300 stopni/s. Z braku informacji o przyspieszeniach ustawiono je na wartości domyślne.

```
# Goal Tolerance Parameters
  yaw_goal_tolerance:     0.05
  xy_goal_tolerance:      0.10
  latch_xy_goal_tolerance: false
```

Tutaj określamy tolerancję planera na dokładność osiągnięcia punktu docelowego, odpowiednio: w radianach na rotację, w metrach na translację.

```
# Forward Simulation Parameters
  sim_time:               1.0
  sim_granularity:        0.025
  angular_sim_granularity: 0.025
  vx_samples:             3
  vy_samples:             0
  vtheta_samples:         20
  controller_frequency:   20   → 10 (rozdział 5.3.1.)
```

Kontroler nieprzerwanie planuje i symuluje kolejne ruchy bazy robota – domyślnie jedną sekundę do przodu (*sim_time*). Rozmiar kroku pomiędzy punktami trajektorii to *sim_granularity* – 25 mm, podobnie definiujemy go w radianach dla rotacji. Kolejne trzy parametry określają liczby próbek używane podczas odkrywania przestrzeni prędkości x, y, theta. Częstotliwość pracy kontrolera (planera lokalnego) to domyślnie 20 Hz.

```
# Trajectory Scoring Parameters
  meter_scoring:              true
  pdist_scale:                0.6 → 5.0
  gdist_scale:                0.8 → 0.5 (rozdział 5.3.1.)
  occdist_scale:              0.01
  heading_lookahead:          0.325
  heading_scoring:            false
  heading_scoring_timestep: 0.8
  dwa:                        true
  publish_cost_grid_pc:       false
  global_frame_id:            odom
```

Parametry punktowania trajektorii definiują sposób wyboru ścieżki ruchu. Pierwszy z nich powoduje korzystanie z jednostki metrów przy określaniu trzech kolejnych wartości: *pdist_scale* – jak blisko ścieżki globalnej powinien się trzymać planer lokalny, *gdist_scale* – jak bardzo ma starać się osiągnąć cel lokalny, *occdist_scale* – jak bardzo ma próbować unikać przeszkód. Parametr *heading_scoring* o wartości *false* powoduje bazowanie na odległości robota od ścieżki nie skupiając się na jego ustawieniu względem niej. Zmienna *dwa* powoduje korzystanie z metody Dynamicznych Okien (*Dynamic Window Approach*). Ostatnia linijka określa układ współrzędnych siatki kosztów.

```
# Oscillation Prevention Parameters
  oscillation_reset_dist:     0.05
# Global Plan Parameters
  prune_plan:                 true
```

Oscylacje następują, gdy w jednym kierunku ruchu (x, y, theta) są wybierane kolejno ujemne i dodatnie wartości prędkości. Aby im zapobiec, podczas ruchu robota ustawiana jest flaga, która blokuje możliwość ruchu w przeciwnym kierunku, aż do momentu gdy robot przebędzie drogę równą 5 cm i flaga zostanie zresetowana. Parametr *prune_plan* o wartości *true* powoduje obcinanie planu globalnego podczas ruchu robota; wcześniejsze punkty trajektorii są usuwane, kiedy robot znajdzie się w odległości jednego metra od nich.

MECHANIZM *RECOVERY*

```
  recovery_behavior_enabled:    true
  recovery_behaviors: [{name: conservative_reset, type:
clear_costmap_recovery/ClearCostmapRecovery}, {name: rotate_recovery, type:
rotate_recovery/RotateRecovery}, {name: aggressive_reset, type:
clear_costmap_recovery/ClearCostmapRecovery}]
  clearing_rotation_allowed:    true
  conservative_reset_dist:      3.0
```

Podczas nawigacji może zdarzyć się sytuacja, w której węzeł *move_base* uzna pozycję robota

za zablokowaną i nie będzie w stanie wyznaczyć dalszej trajektorii ruchu. Przydatne mogą okazać się wtedy domyślne mechanizmy przywracania sprawności nawigacji. Po pierwsze, przeszkody znajdujące się poza obszarem określonym w *conservative_reset_dist* zostają usunięte z mapy kosztów. Jeśli to nie pomaga, robot próbuje obrócić się w miejscu i odświeżyć informacje o przestrzeni. Kolejną czynnością jest wyczyszczenie całej mapy kosztów poza obszarem prostokąta, w którym możliwy jest obrót bazy robota w miejscu. Następnie robot wykonuje taki obrót i jeśli cel nadal jest nieosiągalny, proces nawigacji zostaje przerwany.

POZOSTAŁE PARAMETRY

```
shutdown_costmaps:      false
controller_patience:    15.0
planner_frequency:      0.0
planner_patience:       5.0
oscillation_timeout:    0.0
oscillation_distance:   0.5
```

Pierwszy parametr zapobiega usuwaniu map kosztów podczas nieaktywności węzła *move_base*. Zmienna *planner_frequency* określa częstotliwość pracy planera globalnego – zero oznacza, że jest uruchamiany tylko w razie potrzeby (np. po wskazaniu nowego celu).

MAPY KOSZTÓW

Do przechowywania informacji na temat przeszkód występujących w środowisku wykorzystywane są mapy kosztów. Jedna z nich jest globalna, to znaczy służy do długoterminowego planowania ruchu w całym środowisku, druga jest lokalna i służy unikaniu kolizji z przeszkodami znajdującymi się w pobliżu. Pakiet *costmap_2d* zapewnia realizację obu tych map. Niektóre parametry są wspólne dla mapy globalnej i lokalnej, a niektóre specyficzne dla danego jej typu, dlatego zalecane jest utworzenie trzech następujących plików konfiguracyjnych:

a) *costmap_common.yaml* (parametry wspólne)

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [ [0.07, 0.19], [0.17, 0.08], [0.17, -0.08], [0.07, -0.19], [-0.07, -0.19], [-0.09, -0.16], [-0.19, -0.16], [-0.26, -0.05], [-0.26, 0.05], [-0.19, 0.16], [-0.09, 0.16], [-0.07, 0.19] ]
inflation_radius: 0.55
observation_sources: scan
scan: {sensor_frame: lms100, data_type: LaserScan, topic: /laserscan, marking: true, clearing: true}
```

Parametry znajdujące się na początku definiują zakres odległości, w jakich nastąpi zapisywanie pozycji przeszkód do mapy kosztów. Pierwszy oznacza, że robot będzie aktualizował swoje mapy informacjami o przeszkodach znajdujących się w odległości do 2,5 metra od jego bazy. Drugi sprawia, że robot będzie próbował oczyścić przestrzeń mapy przed sobą do 3 metrów poza odczyt lasera. Dalej znajduje się określenie śladu (*footprint*) robota, przy założeniu, że jego centrum leży w punkcie [0, 0], a kolejne punkty obwodu podawane są zgodnie z kierunkiem ruchu wskazówek zegara. Parametr *inflation_radius* powinien być ustawiony na maksymalną odległość od przeszkody, przy której powinny być ponoszone „koszty”, co w tym wypadku oznacza, że robot będzie traktować wszystkie ścieżki znajdujące się minimum 0,55 metra od przeszkody jako mające równy koszt. W kolejnej linijce zdefiniowana jest lista czujników przekazujących informacje do mapy, a następnie określone są ich parametry. Tutaj jest to jeden czujnik *scan* znajdujący się w członie *lms100*, który publikuje dane typu *LaserScan* w temacie */laserscan* oraz jest używany zarówno do dodawania jak i usuwania informacji o przeszkodach.

b) *global_costmap.yaml*

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
```

Parametr *global_frame* definiuje układ współrzędnych dla mapy, a *robot_base_frame* układ odwołujący się do bazy robota. Kolejna linijka ustala częstotliwość aktualizacji mapy kosztów na 5 Hz. Ostatni parametr oznacza, że wczytana zostanie mapa utworzona wcześniej za pomocą pakietu *gmapping*. W przypadku jej braku konieczne jest ustawienie tego parametru na wartość *false*.

c) *local_costmap.yaml*

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 3.0
  height: 3.0
  resolution: 0.05
```

W przypadku mapy lokalnej układem współrzędnych jest odometria robota. Dwa kolejne

parametry są identyczne jak dla mapy globalnej. Częstotliwość publikowania informacji wizualnej jest określona w *publish_frequency* na 2 Hz. Mapa lokalna jest mapą tworzoną dynamicznie, ma domyślnie rozmiary 3 x 3 metry i rozdzielczość 0,05, a dzięki wartości *true* parametru *rolling_window* w jej centrum zawsze znajdował się będzie robot mobilny.

URUCHAMIANIE WĘZŁA *move_base* korzystającego z wcześniej utworzonych plików konfiguracyjnych odbywa się za pomocą pliku *move_base.launch* o następującej zawartości:

```
<launch>
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find p3dx)/config/move_base.yaml" command="load"/>
    <rosparam file="$(find p3dx)/config/costmap_common.yaml" command="load"
ns="global_costmap" />
    <rosparam file="$(find p3dx)/config/costmap_common.yaml" command="load"
ns="local_costmap" />
    <rosparam file="$(find p3dx)/config/global_costmap.yaml" command="load"  />
    <rosparam file="$(find p3dx)/config/local_costmap.yaml" command="load"/>
  </node>
</launch>
```

4.5.3. Węzeł *AMCL*

AMCL to system lokalizowania probabilistycznego przeznaczony dla robota poruszającego się w dwóch wymiarach. Implementuje adaptacyjne lokalizowanie metodą Monte Carlo; to znaczy korzysta z filtru cząsteczkowego w celu estymowania pozycji robota na znanej mapie. Węzeł działa tylko w połączeniu z czujnikiem laserowym i mapami utworzonymi za jego pomocą.

AMCL

Subskrybowane tematy:

- ***scan*** *[sensor_msgs/LaserScan] – dane pochodzące z czujnika laserowego;*
- ***tf*** *[tf/tfMessage] – transformacje układów współrzędnych;*
- ***initialpose*** *[geometry_msgs/PoseWithCovarianceStamped] – pozycja początkowa robota;*
- ***map*** *[nav_msgs/OccupancyGrid] – mapa otoczenia (opcjonalnie).*

Publikowane tematy:

- ***amcl_pose*** *[geometry_msgs/PoseWithCovarianceStamped] – estymowana pozycja robota;*
- ***particlecloud*** *[geometry_msgs/PoseArray] – zestaw estymowanych pozycji;*
- ***tf*** *[tf/tfMessage] -transformacja z układu odometrycznego do układu współrzędnych mapy.*

Konfiguracja węzła znajduje się w pliku *amcl.yaml* w folderze /p3dx/config/. Istnieją trzy

rodzaje parametrów wykorzystywanych do konfiguracji węzła AMCL:

a) Ogólne parametry filtra.

```
# Overall filter parameters
min_particles:          100
max_particles:          5000
kld_err:                0.01
kld_z:                  0.99
update_min_d:           0.2
update_min_a:           0.5235
resample_interval:      2
transform_tolerance:    0.1  –> 0.5 (konieczna zmiana wartości; rozdział 5.3.)
recovery_alpha_slow:    0.0
recovery_alpha_fast:    0.0
initial_pose_x:         0.0
initial_pose_y:         -3.0
initial_pose_a:         0.0
initial_cov_xx:         0.25
initial_cov_yy:         0.25
initial_cov_aa:         0.06853
gui_publish_rate:       -1.0
save_pose_rate:         0.5
use_map_topic:          false
first_map_only:         false
```

Tutaj można ustalić szereg wartości definiujących pracę filtra cząsteczek: minimalną i maksymalną dozwoloną liczbę cząsteczek, dopuszczalne błędy, minimalną translację (oraz rotację) powodującą odświeżanie filtra i tym podobne. Większość z tych parametrów ustawiono na wartości zalecane (dostępne na stronie *wiki.ros.org/amcl*). Doprecyzowano jedynie pozycję początkową robota poprzez zmianę parametru *initial_pose_y* na -3 metry (model robota podczas każdego uruchomienia symulacji znajduje się w pozycji [0,-3] w układzie współrzędnych mapy, wynika to z parametru serwisu *urdf_spawner*, który można edytować w pliku *gazebo.launch*).

b) Parametry czujnika laserowego.

```
# Laser Model Parameters
laser_min_range:            -1.0
laser_max_range:            -1.0
laser_max_beams:            30
laser_z_hit:                0.95
laser_z_rand:               0.05
laser_sigma_hit:            0.2
laser_lambda_short:         0.1
laser_likelihood_max_dist:  2.0
laser_model_type:           likelihood_field
```

Dostępne są dwa typy modelu lasera: *likelihood_field* (funkcja prawdopodobieństwa) oraz *beam* (promieniowy). Skorzystano tutaj z zalecanego typu i domyślnych wartości. Dwa pierwsze parametry ustawione na -1 sprawiają, że zasięg lasera nie będzie dodatkowo ograniczany. W przypadku zmian konfiguracji należy pamiętać, że wartości parametrów *laser_z_hit* oraz *laser_z_rand* muszą sumować się do jedynki.

c) Parametry modelu odometrii.

```
# Odometry model parameters
odom_model_type:   diff
odom_alpha1:       0.2
odom_alpha2:       0.2
odom_alpha3:       0.2
odom_alpha4:       0.2
odom_frame_id:     odom
base_frame_id:     base_link
global_frame_id:   map
```

Następnie definiujemy parametry modelu odometrycznego robota. Napędowi różnicowemu odpowiada model 'diff', który korzysta z czterech parametrów szumu (kolejno: dryf rotacyjny powodowany przez rotację; następnie przez translację; podobnie dryf translacyjny). Na końcu przyporządkowane zostały odpowiednie układy współrzędnych: układu odometrycznego, bazy robota oraz mapy budynku.

Uruchomienie węzła AMCL odbywa się za pomocą pliku *amcl.launch*:

```
<launch>
  <node pkg="amcl" type="amcl" name="amcl" output="screen" >
    <rosparam file="$(find p3dx)/config/amcl.yaml" command="load"  />
    <remap from="scan" to="laserscan" />
  </node>
</launch>
```

Węzeł korzysta z uprzednio utworzonego pliku konfiguracyjnego oraz danych pochodzących z tematu *laserscan*.

4.6. Programowanie akcji w C++

ROS dostarcza nam narzędzia, takie tak RViz czy RQT, dzięki którym możemy w prosty sposób kontrolować pewne funkcje robota. Ponieważ zestaw tych funkcji jest ograniczony i wystarcza jedynie do podstawowych zastosowań, bardzo ważna jest możliwość samodzielnego zaprogramowania przez użytkownika każdej akcji podejmowanej przez robota. Punkt docelowy

ruchu możemy wprawdzie wskazać za pomocą interfejsu graficznego wizualizatora, ale równie przydatne jest zrobienie tego poprzez uruchomianie pliku wykonywalnego.

4.6.1. Biblioteka *Actionlib*

W systemie ROS zadania, które polegają na wysyłaniu żądania do węzła i oczekiwaniu na odpowiedź, są zazwyczaj obsługiwane przez serwisy. W przypadku, kiedy taki proces trwa dłuższy czas, przydatną funkcją jest informacja na temat przebiegu procesu i możliwość jego przerwania przez użytkownika. Biblioteka *Actionlib* dostarcza wyspecjalizowane narzędzia do tworzenia serwerów wykonujących długotrwałe procesy, a także interfejsu klienta wysyłającego żądania do serwera. *ActionClient* oraz *ActionServer* komunikują się ze sobą poprzez *ROS Action Protocol* za pomocą kilku wiadomości składających się na specyfikację akcji:

a) GOAL – definiuje pojęcie celu wysyłanego przez klienta do serwera. Kiedy akcją jest ruch robota mobilnego, celem jest wiadomość *PoseStamped* zawierająca informację o miejscu, do którego powinien pojechać robot;

b) FEEDBACK – realizuje stałe powiadamianie klienta o bieżących postępach w wykonywaniu procesu. Może to być na przykład aktualna pozycja robota na trasie jego ruchu;

c) RESULT – rezultat akcji wysyłany do klienta jednorazowo po ukończeniu procesu. Jest bardzo ważny, kiedy akcja dotyczy uzyskiwania pewnego rodzaju informacji, np. wykonania skanu otoczenia przez laser. W wypadku nawigacji robota rezultatem jest jego pozycja końcowa, która zazwyczaj różni się nieznacznie od celu w zależności od sprawności pakietów nawigacyjnych.

Specyfikacja akcji jest zdefiniowana w pliku *.action*, który zawiera wiadomości oddzielone znakiem '---'. Plik na potrzeby bieżącego projektu o nazwie *goal.action* utworzono w folderze /p3dx/action/; wygląda on następująco:

```
# Define the goal
float32 PoseStamped
---
# Define the result
float32 Pose
---
# Define a feedback message
float32 Pose
```

W celu poprawnej kompilacji pakietu wykorzystującego bibliotekę *Actionlib* konieczne jest dodanie i uzupełnienie kilku wpisów na początku pliku *CmakeList.txt*, w których między innymi

wskazujemy nazwy zewnętrznych pakietów oraz ścieżkę do pliku ze specyfikacją akcji:

```
find_package(catkin REQUIRED COMPONENTS roscpp rospy urdf actionlib actionlib_msgs std_msgs)
add_action_files(DIRECTORY action FILES goal.action)
generate_messages(DEPENDENCIES actionlib_msgs std_msgs)
```

Konieczne jest również dodanie zależności do manifestu pakietu (plik *package.xml*):

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>actionlib</run_depend>
<run_depend>actionlib_msgs</run_depend>
<run_depend>std_msgs</run_depend>
```

4.6.2. Program realizujący ruch robota do docelowej pozycji

Akcja, która powinna być realizowana po uruchomieniu pliku wykonywalnego, to podanie pakietowi nawigacyjnemu punktu na mapie, do którego następnie uda się robot. Serwerem akcji będzie więc węzeł *move_base*, który w pakiecie nawigacyjnym odpowiada za ruch bazy robota. Konieczne jest jedynie utworzenie klienta, który wyśle współrzędne punktu docelowego do węzła *move_base*. W folderze /p3dx/scripts/ utworzono plik *goal_base.cpp*, który jako cel ruchu poda bazę (na przykład punkt ładowania) robota.

Plik składa się z następujących części:

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
```

Na początku dołączono niezbędne biblioteki. *MoveBaseAction.h* zawiera specyfikację akcji umożliwiającej akceptowanie celu pochodzącego od klienta przez węzeł *move_base*. Plik *simple_action_client.h* to klasa prostego klienta pozwalającego na istnienie jednego aktywnego celu, tzn. wcześniejszy cel jest nadpisywany przez późniejszy.

```
typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
```

Tutaj utworzono przydatny *typedef* (słowo kluczowe w języku C++, które przypisuje nazwy alternatywne do istniejących typów), który pozwoli na komunikowanie się z akcjami odnoszącymi się do interfejsu *MoveBaseAction.*

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "navigation_goals");
  MoveBaseClient ac("move_base", true);
```

Następnie znajduje się początek funkcji *main*. Ostatnia linia tworzy klienta, który będzie się komunikował z akcją *move_base*.

```
while(!ac.waitForServer(ros::Duration(5.0))){
  ROS_INFO("Waiting for the move_base action server"); }
```

Przed wysłaniem celu do serwera akcji dobrym zwyczajem jest sprawdzenie, czy jest on uruchomiony i gotowy na przyjęcie celu.

```
move_base_msgs::MoveBaseGoal goal;
goal.target_pose.header.frame_id = "map";
goal.target_pose.header.stamp = ros::Time::now();
goal.target_pose.pose.position.x = -0.4;
goal.target_pose.pose.position.y = -3.6;
goal.target_pose.pose.orientation.w = 1.0;
ROS_INFO("Sending goal: base");
ac.sendGoal(goal);
```

W pierwszej linii następuje deklaracja wiadomość *goal*. Następnie wprowadzono położenie punktu docelowego w układzie współrzędnych mapy. Funkcja *ac.sendGoal* wysyła go do węzła *move_base*.

```
ac.waitForResult();
if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
ROS_INFO("You have arrived to the charging base");
else{ ROS_INFO("The base failed for some reason"); }
return 0; }
```

Teraz pozostaje już tylko czekanie na zakończenie procesu i wyświetlenie odpowiedniej informacji w zależności od tego, czy baza robota osiągnęła punkt docelowy, czy podczas procesu wystąpiły jakieś problemy.

Aby skompilować plik *goal_base.cpp* podczas kompilowania pakietu komendą *catkin_make* należy na końcu pliku *CMakeList.txt* dodać poniższe linijki:

```
add_executable(GoToBase scripts/goal_base.cpp)
add_dependencies(GoToBase ${p3dx_EXPORTED_TARGETS})
target_link_libraries(GoToBase ${catkin_LIBRARIES})
```

Uruchomienie pliku wykonywalnego jest odtąd możliwe za pomocą komendy:

```
rosrun p3dx GoToBase
```

W analogiczny sposób utworzono kilka kolejnych plików realizujących ruch robota do różnych punktów w symulowanym budynku.

Pozostaje jeszcze kwestia anulowania celu w trakcie trwania procesu nawigacji. Jest to możliwe poprzez wysłanie do odpowiedniego tematu wiadomości zawierającej pusty cel. Można to wykonać komendą:

```
rostopic pub /move_base/cancel actionlib_msgs/GoalID – {}
```

Dla ułatwienia napisano bardzo prosty program, który również dołączono do pakietu:

```
#include <ros/ros.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "CancelGoal");
  ros::start();
  ROS_INFO_STREAM("The goal has been cancelled!");
  system("rostopic pub /move_base/cancel actionlib_msgs/GoalID -- {}");
  ros::shutdown();
return 0;}
```

Realizuje on za nas czynność wpisywania i wykonywania komendy w konsoli, informując dodatkowo o tym, że cel został anulowany.

4.7. Konfiguracja RViz i RQT

Przed rozpoczęciem badań nad symulacją postanowiono skonfigurować wykorzystywane narzędzia, tak aby praca była wygodniejsza. Wizualizator RViz korzysta z modelu robota w formacie URDF, który jest pobierany z serwera parametrów (rozdział 2.2.3). Należy jedynie utworzyć plik *.launch*, który uruchamia węzeł *rviz*:

```
<launch>
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find p3dx)config/mapping.rviz" />
</launch>
```

Plik *mapping.rviz* zawiera konfigurację wizualizatora na potrzeby tworzenia mapy. Konfigurację taką wykonano podczas pracy z programem, a następnie skorzystano z funkcji „zapisz jako...”. Inaczej niż podczas mapowania budynku powinien wyglądać wizualizator podczas testowania nawigacji, dlatego następnie utworzono drugą konfigurację, którą zapisano pod nazwą *nav.rviz*. Konieczne było również utworzenie drugiego pliku *.launch*.

Konfiguracja narzędzia RQT sprowadziła się do wybrania żądanych pluginów i dołączenia ich do okna programu. Rozmiar i pozycję każdego z nich można dowolnie edytować i dostosować do swoich potrzeb. Następnie widok okna programu zapisano do pliku *p3dx.perspective* w katalogu /p3dx/config/. Plik '*.launch'* uruchamiający tak skonfigurowany program RQT wygląda

następująco:

```
  <launch>
    <node name="p3dx_panel" pkg="rqt_gui" type="rqt_gui" args="--perspective-file $(find
p3dx)config/p3dx.perspective" />
  </launch>
```

5. Realizacja wybranych scenariuszy symulacji

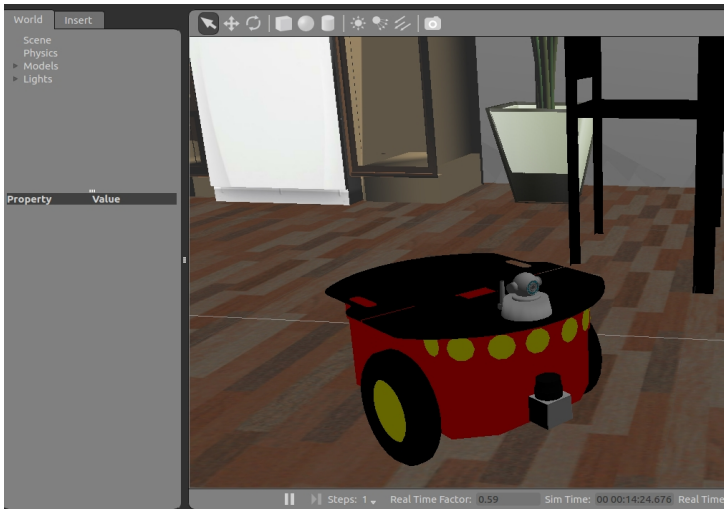
W tym rozdziale znajdują się opisy scenariuszy symulacji przeprowadzonych z wykorzystaniem wcześniej utworzonych modeli i zaimplementowanych pakietów.

5.1. Uruchomienie symulacji i manualne sterowanie robotem

Główny węzeł projektu, czyli symulator Gazebo ze wczytanym środowiskiem domowym oraz modelem robota Pioneer 3-DX (rys. 5.1), uruchamiamy za pomocą następującej komendy:

```
roslaunch p3dx gazebo_gui.launch
```

Plik *gazebo_gui.launch* uruchamia także węzły odpowiedzialne za transformacje układów współrzędnych. ROS Master startuje automatycznie wraz z Gazebo. Po krótkiej chwili pokazuje się interfejs graficzny symulatora. W środkowej części okna można obserwować symulowany świat, a korzystając z panelu po lewej stronie dodawać lub usuwać obiekty. Na dole ekranu znajduje się przycisk [pauza/start symulacji], a obok niego wskaźniki: upływ czasu w symulacji, w realnym świecie i wynikająca z nich relacja czasu w symulacji do czasu realnego. W idealnych warunkach współczynnik ten powinien być równy '1' lub większy, jednak wymaga to mocnego procesora. Na komputerze wyposażonym w Pentium Dual-Core 2x2.00 GHz wahał się on od około 0,65 do 0,15 (przy aktywnych wielu procesach jednocześnie), a wykorzystanie obu rdzeni procesora rzadko spadało poniżej 100%. Nie ma to jednak wpływu na przebieg symulacji, a jedynie na wygodę obserwacji i płynność działania procesów. W celu poprawy płynności zmniejszono częstotliwość odświeżania czujnika laserowego z 50 Hz na 25 Hz, a także kamery z 30 na 5 kl/s. Dodatkowo zmniejszono rozdzielczość obrazu z kamery do 240 x 240 pikseli. Nie wystąpiły problemy, zauważono za to niewielki wzrost wartości współczynnika.



Rys. 5.1. Model robota Pioneer 3-DX gotowy do pracy [źródło: własne]

Uruchomienie węzła pozwalającego na sterowanie bazą robota za pomocą klawiatury jest możliwe za pomocą komendy:

```
roslaunch p3dx wsad.launch
```

Spowoduje to wyświetlenie w oknie konsoli interfejsu objaśniającego użycie przycisków:

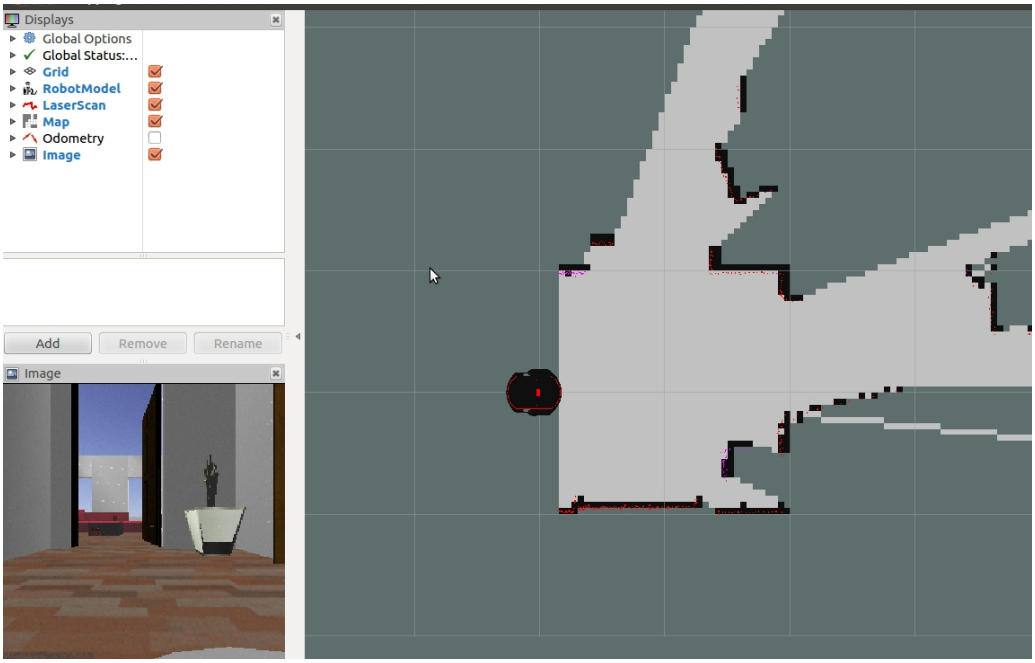
```
Pioneer 3DX Keyboard Control
---------------------------
Moving around:
      w
  a   s   d

q/z : increase/decrease max speeds by 10%
space key : force stop
anything else : stop smoothly
```

Robot poprawnie reaguje na komendy i nie zauważono żadnych błędów w trakcie manualnego sterowania. Oznacza to, że model robota jest prawidłowo skonfigurowany, także pod względem właściwości fizycznych.

5.2. Tworzenie mapy budynku

Plik *mapping.launch* uruchamia serwer programu Gazebo (środowisko pracy bez interfejsu graficznego), węzeł *gmapping* oraz odpowiednio skonfigurowany wizualizator RViz.



Rys. 5.2. Interfejs wizualizatora RViz skonfigurowany na potrzeby tworzenia mapy [źródło: własne]

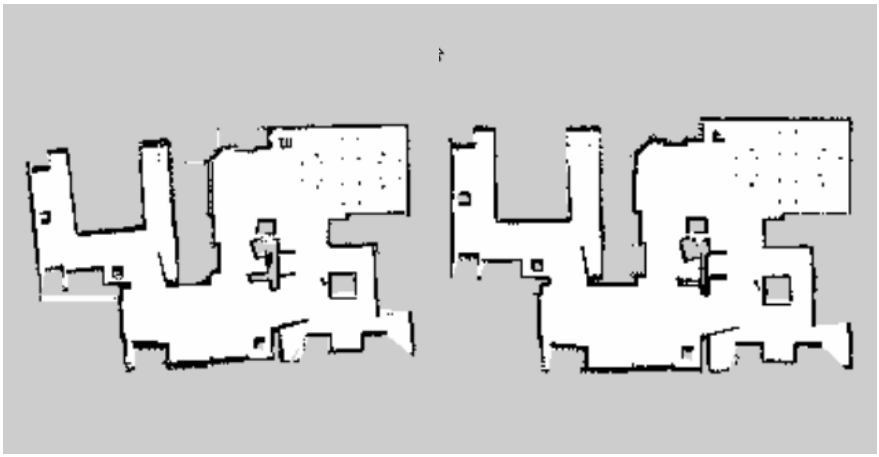
Plik ten uruchomiono komendą:

```
roslaunch p3dx mapping.launch
```

Po załadowaniu interfejsu wizualizatora (rys. 5.2) w nowym oknie konsoli uruchomiono plik *wsad.launch* i za pomocą przycisków klawiatury sterowano robotem po całym mieszkaniu. W międzyczasie obserwowano tworzącą się mapę i aktualne wskazania czujnika laserowego. Dodatkowo w lewym dolnym rogu ekranu znajduje się widok z kamery zamontowanej na robocie. Tworzenie mapy w powyższy sposób uznano za najwygodniejszy. W każdej chwili można także wyświetlić interfejs graficzny Gazebo uruchamiając klienta programu:

```
rosrun gazebo_ros gzclient
```

Podczas pierwszej próby tworzenia mapy zauważono problem wynikający z umieszczenia czujnika laserowego. Przy 270 stopniach pola widzenia lasera, część bazy robota była wykrywana jako przeszkoda, co powodowało powstawanie błędów na mapie pomieszczeń. Zdecydowano się ograniczyć programowo pole widzenia do 180 stopni, co skutecznie zniwelowało błędy. Kolejny problem wynikał z tego, że ściany będące równoległe do siebie, często na mapie wyglądały inaczej (rys. 5.3). Mogło to być spowodowane błędnym obliczaniem aktualnej pozycji robota przez pakiet *tf*. Po przeszukaniu wątków opisujących podobne problemy na portalu użytkowników ROS [7] zdecydowano się na dodanie do konfiguracji pakietu *gmapping* parametru *tf_delay* o wartości 0,1 sekundy. Podczas kolejnej próby mapa tworzyła się prawidłowo. Przy okazji zauważono, że plik z zapisaną mapą zajmuje 16 MB mimo niewielkich rozmiarów budynku. Spowodowane było to parametrami odpowiadającymi za początkową wielkość mapy. Zmniejszono je każdorazowo ze 100 m do 2 m, co spowodowało spadek rozmiaru pliku z mapą do 0,3 MB bez wystąpienia dodatkowych problemów.



Rys. 5.3. Mapa utworzona podczas pierwszej próby (z lewej) i mapa prawidłowa powstała po zmianach w konfiguracji pakietu gmapping (z prawej) [źródło: własne]

Kiedy mapa była poprawna zapisano ją komendą:

```
rosrun map_server map_saver -f p3dx_map
```

Następnie przeniesiono ją z katalogu domowego do folderu /p3dx/config/, skąd będzie ona pobierana przez pakiety nawigacyjne.

5.3. Nawigacja

Pierwszym zadaniem po utworzeniu mapy jest przetestowanie poprawności uruchamiania się pakietów nawigacyjnych. W tym celu w czasie symulacji w Gazebo uruchomiono plik *nav.launch*, który zawiera trzy węzły pakietu nawigacyjnego. Wszystkie procesy zadziałały, jednak w konsoli pojawił się błąd o treści:

```
[ERROR]: Extrapolation Error: Lookup would require extrapolation into the future. Requested time 1185.211000000 but the latest data is at time 1185.158000000, when looking up transform from frame [odom] to frame [map]
[WARN]: Could not transform the global plan to the frame of the controller
```

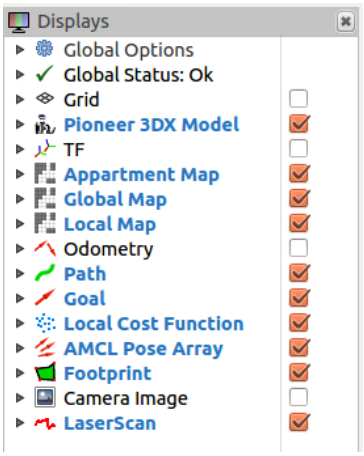
Okazał się to być problem pakietu AMCL związany z opóźnieniem publikowania transformacji układów współrzędnych, a spowodowany prawdopodobnie zbyt niską wydajnością komputera. Po zmianie wartości parametru *transform_tolerance* z 0,1 na 0,5 (w pliku *amcl.yaml*) problem został rozwiązany, a konsola nie wyświetlała żadnych błędów. Teraz można było przystąpić do realizowania scenariuszy autonomicznej nawigacji robota.

5.3.1. Autonomiczna nawigacja robota do miejsca wybranego na mapie

Pierwszy scenariusz nawigacji będzie polegał na wskazywaniu punktu docelowego za pomocą interfejsu graficznego wizualizatora RViz. Zestaw wszystkich niezbędnych pakietów uruchomiono za pomocą komendy:

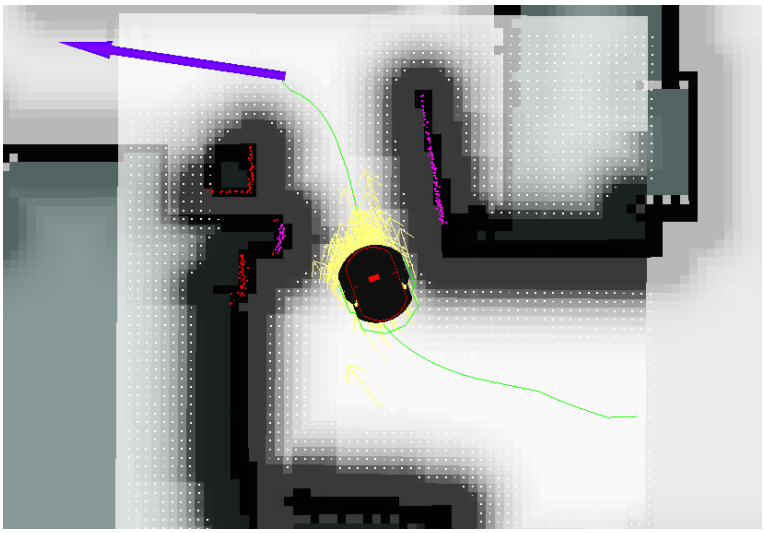
```
roslaunch p3dx nav1.launch
```

Powoduje to załadowanie wszystkich potrzebnych węzłów i wyświetlenie skonfigurowanego okna wizualizatora RViz. Po lewej stronie okna programu widać spis wszystkich możliwych do wizualizacji danych (rys. 5.4). Wyboru wyświetlanych elementów dokonujemy poprzez odznaczenie lub zaznaczenie okienka przy odpowiedniej nazwie. Wskazywanie punktu docelowego następuje poprzez wybranie z paska znajdującego się na górze okna narzędzia *2D Nav Goal* a następnie zaznaczenie punktu na mapie lewym przyciskiem myszki. Przytrzymując ten przycisk możemy dodatkowo ruchem myszki określić zwrot robota po osiągnięciu celu.



Rys. 5.4. Spis wszystkich elementów wizualizujących dane w programie Rviz [źródło: własne]

Robot natychmiastowo reaguje na wskazanie celu i zaczyna poruszać się w jego kierunku. W oknie wizualizatora możemy na bieżąco obserwować wybrane dane (rys. 5.5). Już podczas pierwszej próby robot osiągnął wyznaczony cel, jednak trwało to bardzo długo z powodu licznych, niepotrzebnych obrotów w miejscu. Ich źródłem okazała się być zbyt duża częstotliwość pracy planera lokalnego. Po zmniejszeniu jej wartości w pliku *move_base.yaml* z 20 Hz na 5 Hz robot osiągał cele z dużą prędkością, jednak wiązało się to czasami z zahaczaniem o przeszkody leżące w pobliżu trajektorii. Dlatego zdecydowano się na ustawienie wartości pośredniej wynoszącej 10 Hz.



Rys. 5.5. Robot w trakcie ruchu do punktu docelowego [źródło: własne]

Planowanie globalne wyznacza trajektorię cechującą się jak najmniejszym kosztem, co nie zawsze odpowiada najkrótszej możliwej drodze. Jednak z racji wąskich przejść znajdujących się w budynkach mieszkalnych znacznie ważniejsza od szybkości jest dokładność poruszania się bazy robota. Aby planowanie lokalne jak najlepiej podążało wyznaczoną trasą, zmieniono konfigurację

parametrów odpowiedzialnych za punktowanie trajektorii (rozdział 4.5.2).

Po wyżej opisanych zmianach robot szybko i bez problemów osiągał zakładane cele. W wypadku podania celu znajdującego się poza dostępnym dla bazy robota obszarem nazwa elementu *Path* zmieniła kolor na czerwony, sygnalizując brak możliwości wyznaczenia trajektorii.

W trakcie ruchu robota uruchomiono interfejs graficzny Gazebo, a następnie przeprowadzono dodatkowe testy sprawdzające zachowanie pakietu nawigacyjnego w zmieniających się (w typowy dla budynku mieszkalnego sposób) warunkach:

a) zatrzymano symulację, po czym „zamknięto” drzwi (zablokowano przejście) prowadzące do pomieszczenia, w którym mieści się punkt docelowy. Następnie wznowiono symulację. Robot przemieszczał się wcześniejszą trajektorią do momentu, kiedy zamknięte drzwi znalazły się w zasięgu lokalnej mapy kosztów. Wtedy planer globalny zaczął szukać nowej trajektorii, a jeśli znalezienie jej było niemożliwe, robot zatrzymywał się, informując o braku możliwości dotarcia do celu.

b) zatrzymano symulację, po czym na trasie robota umieszczono przeszkodę w postaci modelu puszki z napojem (symulując przeszkodę pojawiającą się dynamicznie) i wznowiono symulację. Po przeprowadzeniu prób z ustawianiem puszki w różnej odległości od bazy robota stwierdzono, że planer globalny uaktualnia trajektorię tylko wtedy, gdy przedmiot pojawi się poza zasięgiem lokalnej mapy kosztów. Jeśli przeszkoda pojawia się nagle w mniejszej odległości, zostaje wykryta przez czujnik, ale z powodu złej trajektorii robot kręci się w miejscu i wymaganie jest ponowne wprowadzenie celu. Rozwiązaniem prowadzącym do ulepszenia wykrywania dynamicznie pojawiających się przeszkód może być zmniejszenie wymiarów lokalnej mapy kosztów: przy wymiarach 1 x 1 metr omijane poprawnie są przeszkody pojawiające się w odległości około 30 cm od poruszającego się robota. Z drugiej strony zbyt mała mapa lokalna może powodować wydłużenie czasu osiągania wyznaczonego celu, dlatego najlepiej dopasować ten parametr do aktualnych potrzeb.

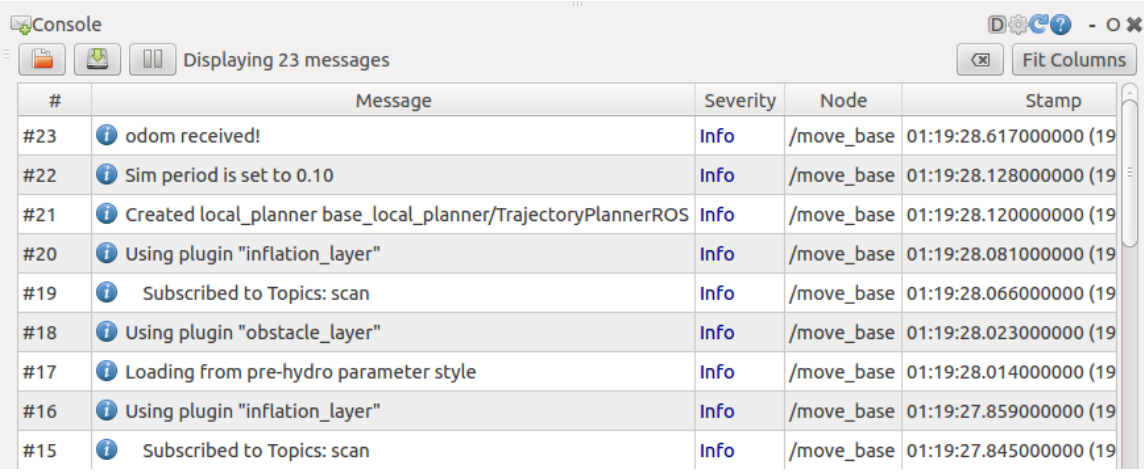
5.3.2. Autonomiczna nawigacja robota pomiędzy zaprogramowanymi miejscami

Drugi scenariusz nawigacji będzie przebiegał za pomocą interfejsu graficznego zaprojektowanego przy pomocy narzędzia RQT. Użycie komendy:

```
roslaunch p3dx nav2.launch
```

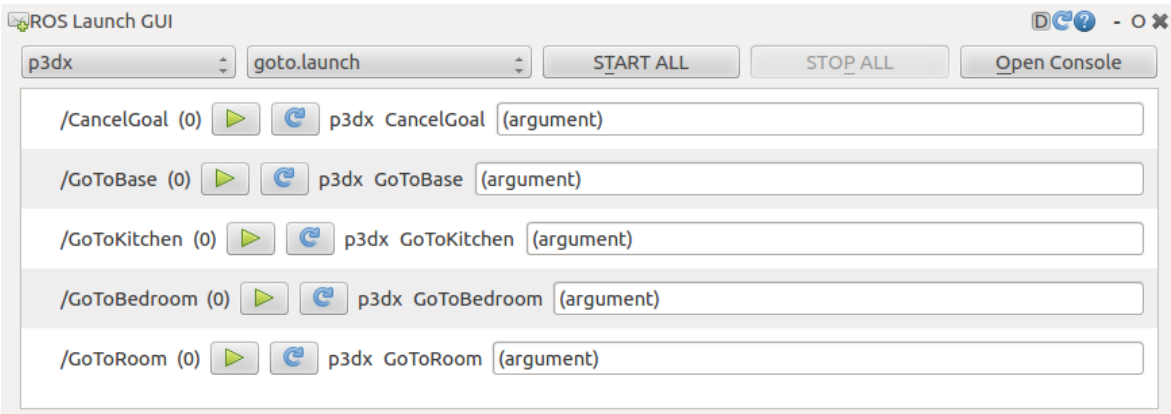
powoduje wczytanie wszystkich potrzebnych węzłów oraz wyświetlenie interfejsu graficznego.

W prawym dolnym rogu znajduje się konsola ROS (rys. 5.6), w której można śledzić przydatne informacje dostarczane przez uruchomione procesy lub te o ewentualnych błędach. Wszystkie procesy są uruchomione i gotowe do pracy po wyświetleniu wiadomości „odom received!”.



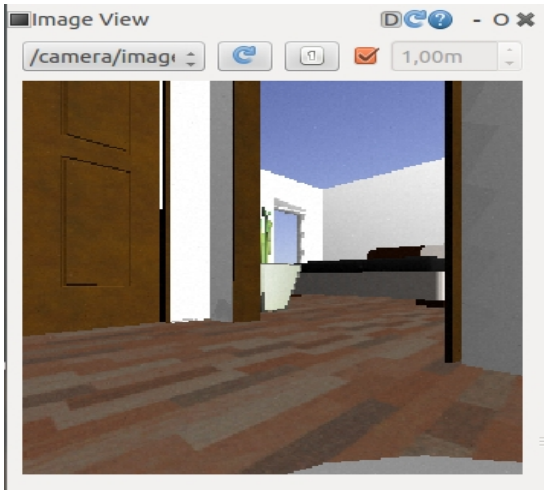
Rys. 5.6. Konsola ROS wyświetlająca bieżące informacje [źródło: własne]

Teraz w oknie *ROS Launch GUI* znajdującym się powyżej konsoli należy wybrać przycisk [gazebo_ros] i z rozwiniętej listy wybrać nazwę pakietu [p3dx]. Poniżej pojawi się pięć poleceń wraz z przyciskami (rys. 5.7), za pomocą których będziemy wydawać polecenia nawigacyjne. Wybranie przycisku z zieloną strzałką powoduje uruchomienie węzła realizującego proces zgodny z opisem znajdującym się obok.



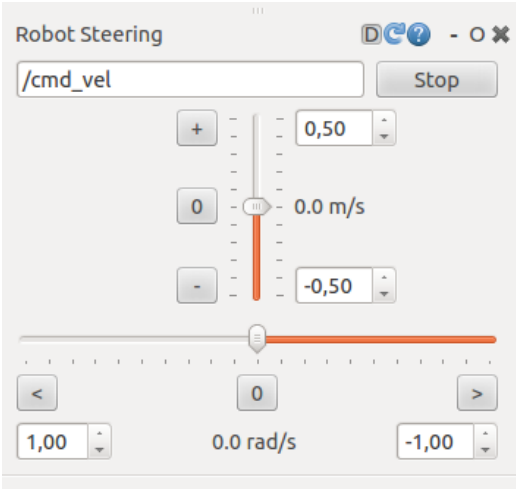
Rys. 5.7. Okno wydawania poleceń nawigacyjnych [źródło: własne]

W lewym górnym rogu znajduje się obraz z kamery służący do celów obserwacyjnych (rys. 5.8) W razie błędów w wyświetlaniu obrazu należy nacisnąć przycisk z niebieską strzałką służący do resetowania urządzenia a następnie w rozwijanej listy po lewej stronie wybrać subskrybowany temat [/camera/image_raw].



Rys. 5.8. Widok z symulowanej kamery [źródło: własne]

Ostatnim elementem interfejsu graficznego jest okno służące do manualnego sterowania ruchem robota (rys. 5.9). Polecenia zmiany prędkości wydajemy za pomocą suwaków, a robota zatrzymujemy przyciskiem [Stop]. Możliwe jest również używanie przycisków na klawiaturze, odpowiednio: w, s, a, d oraz spacji.



Rys. 5.9. Okno służące do manualnego sterowania robotem [źródło: własne]

Sterowanie robotem za pomocą opisanego interfejsu przebiega bez problemów. Po wciśnięciu zielonej strzałki uruchamiającej nawigację do wybranego celu, robot zaczyna się poruszać. W każdej chwili mamy możliwość zatrzymania robota poprzez wybranie przycisku realizującego funkcję *CancelGoal*. Każda czynność jest potwierdzana odpowiednią informacją w konsoli ROS, a lewym górnym rogu możemy na bieżąco obserwować widok z kamery umieszczonej na robocie.

6. Podsumowanie i wnioski

Wszystkie cele postawione w założeniach pracy zostały zrealizowane. Symulowany model robota poprawnie reaguje na manualne sterowanie. Zaimplementowana nawigacja pozwala mu także na autonomiczne przemieszczanie się do wybranego celu. Punkt docelowy może być wybrany przez użytkownika za pomocą interfejsu wizualizatora lub poprzez uruchomienie węzła realizującego program napisany w języku C++. Ruch robota po mieszkaniu składającym się głównie z wąskich przejść przebiega w zasadzie bez problemów. Robot dobrze radzi sobie także z omijaniem przeszkód pojawiających się dynamicznie.

Podczas prac nad projektem stwierdzono, że ROS zapewnia bardzo duże możliwości sterowania zachowaniem robotów. Dodatkowym atutem wykorzystanego oprogramowania jest możliwość relatywnie szybkiego utworzenia dopasowanego do wymagań użytkownika interfejsu graficznego za pomocą dołączonych narzędzi. Taki interfejs może być wygodnie obsługiwany za pomocą coraz popularniejszych (zwłaszcza w komputerach przenośnych) ekranów dotykowych.

Najbardziej czasochłonne było z pewnością poznanie zasad działania systemu ROS na podstawie dokumentacji w języku angielskim. Sporym problemem okazały się być także wysokie wymagania sprzętowe systemu, co na wykorzystywanym komputerze wymusiło około pięciokrotne spowolnienie upływu czasu w symulacji. Podjęte próby optymalizacji procesów poprzez zmniejszenie ilości przetwarzanych danych przyniosły niewielki efekt. Dlatego też w ramach dalszego rozwoju pracy należałoby uruchomić projekt na komputerze o dużej wydajności, który umożliwiłby znaczne przyspieszenie upływu czasu w symulacji. Z pewnością pomogłoby to lepiej ocenić sprawność systemu nawigacyjnego i, w razie potrzeby, dopasować jego parametry. Praca może stanowić także dobrą bazę symulacyjną i być podstawą do projektów implementujących nowe zachowania oraz akcesoria do robota Pioneer 3-DX lub zajmujących się badaniem wykorzystania wizji maszynowej i testowaniem algorytmów rozpoznawania przedmiotów w robotyce. Interesującym projektem mogłoby być na przykład rozszerzenie niniejszej pracy poprzez dodanie do robota prostego manipulatora i realizacja zadania odnalezienia (z wykorzystaniem analizy obrazów) i przyniesienia jakiegoś przedmiotu.

Bibliografia

[1] Dokumentacja systemu ROS [online – dostęp: 10.09.2014]:
http://wiki.ros.org

[2] Dokumentacja programu Gazebo [online – dostęp: 10.09.2014]:
http://gazebosim.org

[3] Dokumentacja modelu robota Pioneer 3-DX [online – dostęp: 10.09.2014]:
http://mobilerobots.com/ResearchRobots/PioneerP3DX.aspx

[4] Aaron Martinez, Enrique Fernandez *„Learning ROS for Robotics Programming”*, Packt Publishing 2013, ISBN 978-1-78216-144-8

[5] Internetowa baza darmowych modeli 3D [online – dostęp: 10.09.2014]:
https://3dwarehouse.sketchup.com

[6] Strona internetowa firmy Robotnik projektującej roboty użytkowe działające w oparciu o system ROS [online – dostęp: 10.09.2014]:
http://www.robotnik.eu

[7] Filmy znajdujące się na oficjalnym profilu ROS na portalu YouTube [online – dostęp: 10.09.2014]:
https://www.youtube.com/channel/UCW0TkDiuH6keHG-QvggweAw

[8] Portal użytkowników ROS [online – dostęp: 10.09.2014]:
http://answers.ros.org/questions