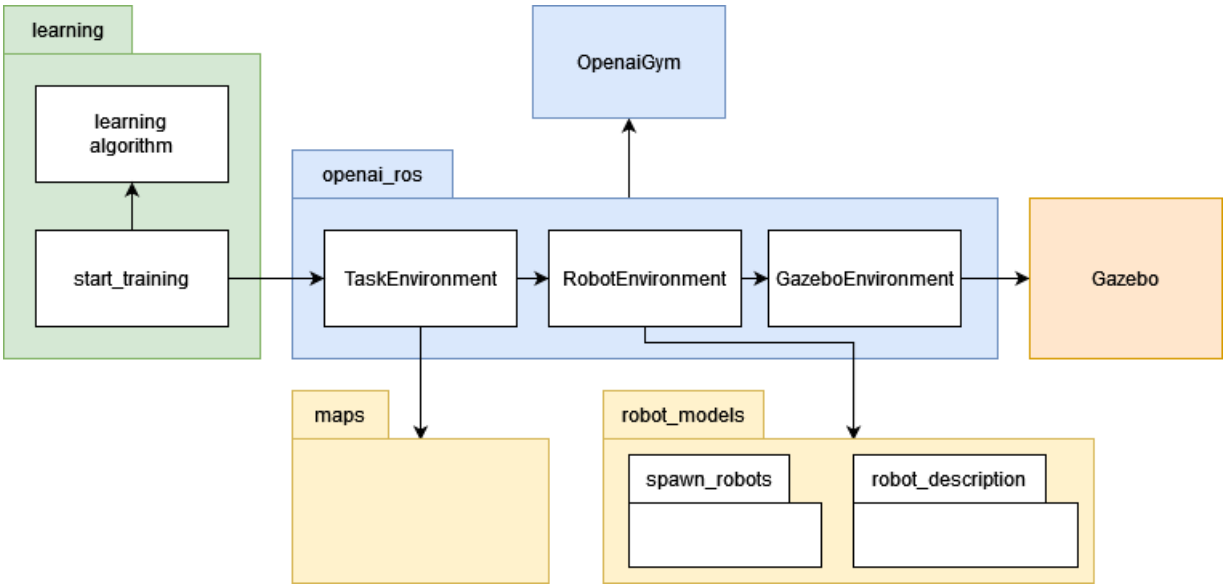
gazebo_openai_tool

Das gazebo_openai_tool ist eine Struktur von Paketen welche auf dem openai_ros Paket von TheConstruct [http://wiki.ros.org/openai_ros] aufbaut. Diese Struktur arbeitet mit dem Gazebo Simulator und OpenAI Gym. Es wurde mit ROS Noetic auf Ubuntu 20.04 entwickelt und getestet. Sie verbindet Simulationsumgebung, Roboter Middleware und Machine Learning bzw. Reinforcement Learning, um eine Umgebung zu schaffen welche einfach aufzusetzen und zu erweitern ist.
Das Ziel dieses Werkzeugs ist es Menschen welche neu in den Gebieten Robotik und Machine Learning sind, eine einfach zu verwendende Anwendung zu geben um ihre ML-Algorithmen zu testen.

Die Paketstruktur wird in vier Module unterteilt: openai_ros, learning, maps und robot_models



Inhalt

openai_ros: 2
learning: 3
maps: 3
robot_models: 3
Verwendung: 4
Neues AI Reinforcement Learning Skript: 5
Neuen Task erstellen: 6
Beschreibung der Welt: 6
TaskEnvironment: 7
Neuen Roboter einfügen: 8
Beschreibung des Roboters: 8
RobotEnvironment: 11

openai_ros:

openai_ros ist der Kern von gazebo_openai_tool. Es wird von TheConstruct entwickelt und zur Verfügung gestellt. Es ist eine Struktur, welche es einfach macht, OpenAI Gym mit ROS und Gazebo zu verbinden. Es stellt Simulationsumgebungen für viele in ROS häufig verwendete Roboter zur Verfügung und kann einfach erweitert werden. Dadurch müssen die Nutzer sich nur auf den AI Teil konzentrieren und müssen sich nicht um die Verbindung zum simulierten Roboter kümmern.
openai_ros wird wiederum in GazeboEnvironment, RobotEnvironment und TaskEnvironment unterteilt.

GazeboEnvironment wird verwendet, um die simulierte Umgebung mit dem Gazebo Simulator zu verbinden.

Hier werden einige der benötigten Funktionen von OpenaAI Gym für Reinforcement Learning implementiert.
Zum Beispiel um die Simulation zurückzusetzen.
Diese Klasse publiziert auch das Ergebnis der letzten Episoden auf dem ROS Topic /openai/reward

Wenn man den Gazebo Simulator verwenden möchte, muss man nichts verändern. Sollte man aber einen anderen Simulator verwenden wollen, muss man hierfür eine neue Python Datei erstellen.
Der Code für das GazeboEnvironment liegt in robot_gazebo_env.py

RobotEnvironment beinhaltet alle ROS Funktionalitäten des Roboters, der verwendet werden soll. Hier wird für jeden Roboter ein eigenes RobotEnvironment benötigt.

openai_ros beinhaltet standardmäßig die RobotEnvironments für die Roboter:
- Cartpole
- Cube robot
- Hopper robot
- ROSbot by Husarion
- Wam by Barret
- Parrot drone
- Sawyer by Rethink robotics
- Shadow Robot Grasping Sandbox
- Summit XL by Robotnik
- Turtlebot2
- Turtlebot3 by Robotis
- WAMV water vehicle of the RobotX Challenge

Um die selbst erstellten Tasks erfüllen zu können wurde ein eigenes Turtlebot3 RobotEnvironment erstellt und das Modell des Roboters um einen Kontaktsensor erweitert.

Das RobotEnvironmetn erbt von dem GazeboEnvironment.

TaskEnvironment beinhaltet die Informationen in welchem Kontext sich der Roboter in der simulierten Welt befinden und was seine Aufgaben und seine Ziele sind.

Es ist wahrscheinlich, dass man ein eigenes TaskEnvironment für unterschiedliche Roboter erstellen muss, auch wenn diese dieselben Aufgaben in derselben Welt erfüllen sollen. Das kommt daher, dass die Roboter die Welt unterschiedlich wahrnehmen und damit interagieren.

Das TaskEnvironment erbt von dem RobotEnvironment.

Für genauere Informationen wird empfohlen sich das OpenAI ROS Wiki [http://wiki.ros.org/openai_ros] und die API-Dokumentation [https://theconstructcore.bitbucket.io/openai_ros/index.html] durchzulesen.

learning:

Dies ist das einzige ROS Paket mit dem End User interagieren müssen, um ihre ML-Skripten zu testen. Vorausgesetzt, dass das Environment in wurde, openai_ros richtig definiert wurde.

Hier werden befinden sich die Dateien, welche zum Start und zur Ausführung des Trainings verwendet werden. Dazu gehören die Datei, welche den RL-Algorithmus beinhaltet und die Datei, welche die Simulationsumgebung initialisiert und den Algorithmus ausführt. Außerdem eine Konfigurationsdatei, welche die benötigten Parameter für die Simulation und den RL-Algorithmus angibt und eine Launch Datei, welche ausgeführt wird, um die Simulation bzw. das Training zu starten.

Die Dateien für das Training sind komplett unabhängig von der Simulationsumgebung. Dadurch kann das Trainingsskript ausgetauscht werden, ohne die Simulationsumgebungen bearbeiten zu müssen.

maps:

In diesem ROS Paket befinden sich alle Dateien, welche für das TaskEnvironment benötigt werden. Dazu gehören verschiedene Modelle wie der Kurs, Hindernisse, Strasenschilder, Ampeln und vieles andere. Eine Gazebo Welt Datei, welche die simulierte Welt beschreibt und alle Modelle an die gewünschte Position setzt. Und ein Launch File, welches vom TaskEnvironment verwendet wird, um die simulierte Welt zu laden.

robot_models:

In diesem Ordner befinden sich alle ROS Pakete, welche mit den Robotern in Verbindung stehen.

Die Dateien der Robotermodelle mit ihren Sensoren befinden sich in den „_description“ ROS Paketen. In dem spawn_robots ROS Paket befinden sich die Launch Files.

Verwendung:

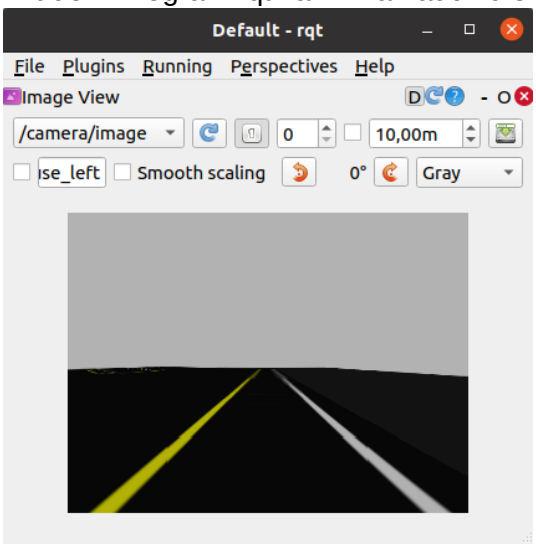
Um das Training zu starten, muss eines der “start_learning” Dateien mit der Launch Datei ausgeführt werden.Dafür muss man einfach den Befehl „roslaunch learning“ und den Namen der Launch Datei ausführen:

roslaunch learning start_training_ql_lr_course1_turtlebot3.launch

Je nachdem in welches Verzeichnis das Tool abgespeichert wurde muss man den Parameter „ros_ws_abspath“ in der Konfigurationsdatei in gazebo_openai_tool/learning/config auf den entspechenden Pfad abändern

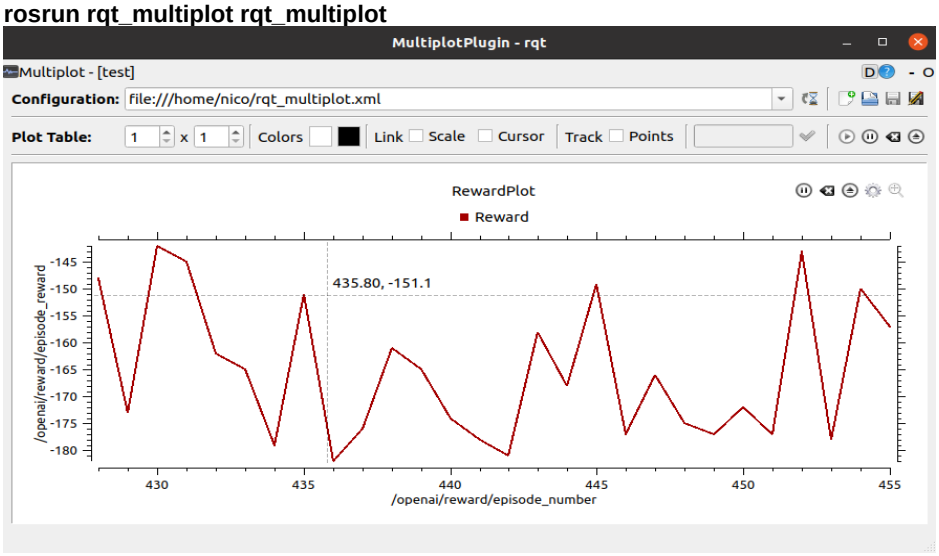
Nun sollte das Gazebo GUI erscheinen und man kann das Training beobachten. Vorausgesetzt in der Launch Datei der Welt unter „gazebo_openai_tool/maps/launch“ hat die Argumente „gui“ = „true“ und „headless“ = „false“ gesetzt.

Mit dem Program rqt kann man auch die Ausgabe der Kamera ausgeben.



Das Training kann während der Ausführung unter dem ROS Topic /openai/reward beobachtet werden. Dafür verwendet man einfach rqt_multiplot und verwendet für die x-Achse /openai/reward/episode_number und für die y-Achse /openai/reward/episode_reward

rosrun rqt_multiplot rqt_multiplot



Wenn das Training beendet ist, werden die Ergebnisse unter gazebo_openai_tool/learning/results ausgegeben.

Neues AI Reinforcement Learning Skript:

Um einen Roboter trainieren zu können müssen zwei Bedingungen erfüllt sein.

Das Environment in openai_ros muss richtig eingerichtet und konfiguriert sein.

Im learning Ordner müssen die zu verwendenden Skripten mit einem Launch File zum Ausführen und einer Yaml Datei mit den nötigen Parametern vorliegen.

Der Teil, in dem der Roboter lernt, wird dann in zwei Teilen im learning Ordner abgespeichert.

In der “learning algorithm” Python Datei wird der Code für den Reinforcement Learning Algorithmus, wie zum Beispiel Qlearning oder Sarsa, implementiert.

In der “start_learning” Python Datei wird dann der ROS Node, welcher den RL-Algorithmus ausführt, initialisiert und das OpenaAI ROS Environment gestartet.

Dies wird mit der Methode StartOpenAI_ROS_Environment(task_and_robot_environment_name) von openai_ros_common.py gemacht.

Um dies durchführen zu können muss die Registrierung des Environments in der Datei task_env_list.py eingetragen sein. Dazu muss nur das TaskEnvironment mit der OpenAI Gym Methode register(id, entry_point, max_episode_steps) registriert werden und die Datei importiert werden.
Dazu sieht man sich einfach die task_env_list.py Datei an und nimmt eine der bestehenden Einträge als Vorlage.

Wenn alle Variablen und das Logging System eingerichtet wurden wird die erste Episode mit dem Trainings Loop gestartet. In dieser Schleife wird der RL-Algorithmus ausgeführt und trainiert.
Der Trainings Loop läuft so lange bis die Episode als beendet gilt oder die maximale Anzahl von Schritten erreicht wurde.

Ein Schleifendurchgang läuft im Normalfall in dem folgenden Schema ab:
Zuerst wird eine Aktion im Bezug auf den aktuellen Zustand gewählt. Dann wird die gewählte Aktion im Environment ausgeführt und wir erhalten Feedback über die Auswirkungen dieser Aktion.
Der Lernalgorithmus lernt dann von den Änderungen und der gewählten Aktion. Ist der Durchlauf nicht abgeschlossen kommt der nächste Schleifendurchgang für einen Schritt und es wiederholt sich.
Falls der Traininigsdruchlauf als beendet gilt, wird die Umgebung zum Startzustand zurückgesetzt und das Training geht weiter.

Dies wiederholt sich bis die gewählte Anzahl der Episoden (nepisodes) erreicht wurde.

Um den Ablauf genauer zu verstehen und gezielt Aktionen auswählen zu können sollte man verstehen wie OpenAI Gym funktioniert und wie das Environment aufgebaut ist. Der Aufbau des Environments wird in openai_ros definiert.

Neuen Task erstellen:

Um einen neuen Task im gazebo_openai_tool verwenden zu können müssen drei Bedingungen erfüllt sein.

Der zu verwendende Roboter muss eingefügt sein und alle für den Task benötigten Funktionalitäten erfüllen. Siehe “Neuen Roboter erstellen”.

Im maps Ordner muss die Beschreibung der zu verwendende Welt vorliegen.
Dazu gehören auch alle in der Welt vorkommenden Objekte wie zum Beispiel Kurse, Wände oder Ampel und deren zugehörigen Meshes. Außerdem wird ein World und ein Launch File benötigt, um die Welt in Gazebo zu laden.

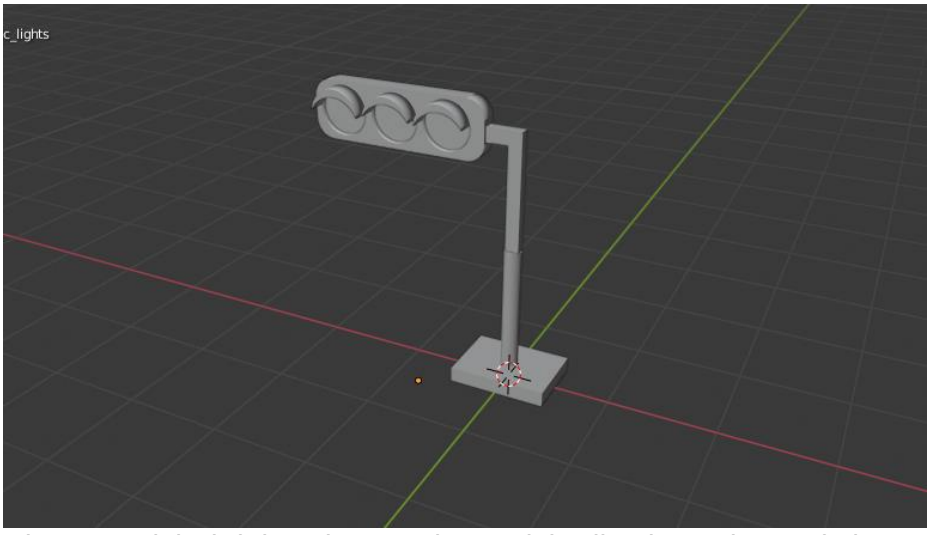
Im "openai_ros" Ordner muss das zugehörige TaskEnvironment enthalten sein. Dazu gehört auch eine Yaml Datei, welche die Parameter enthält, welche für den Task benötigt werden. Außerdem muss das TaskEnvironment noch registriert werden.

Beschreibung der Welt:

Der Aufbau kann sich von Welt zu Welt stark unterscheiden. Je Nachdem was für eine Art von Roboter welche Aufgabe erfüllen soll. Um eine Welt in Gazebo zu erstellen kann der Gazebo Editor verwendet werden. Hier können neue Modelle eingefügt und positioniert werden. Man kann Modelle aus verschiedenen Datenbanken einfügen.

Um eigene Modelle zu erstellen oder Modelle zu bearbeiten verwendet man den Model Editor von Gazebo. Hier können je nach Bedürfnis auch Komplexere Modelle wie Roboter erstellt werden. Da Gazebo aber weit verbreitet ist gibt es in den meisten Fällen bereits Modelle, welche die gewünschten Anforderungen erfüllen.

Beispiel Ampel erstellen:
Um die Ampel zu erstellen, wurde das CAD Model von
https://robotisim.com/2021/08/21/ros2-gazebo-dynamic-traffic-lights-for-real-time-simulation/
verwendet.



Dieses Model wird dann im Gazebo Model Editor importiert und als SDF Datei abgespeichert. Dadurch wird eine model.config und eine model.sdf Datei erstellt.

Um die Ampel funktionsfähig zu machen, wurden dann noch Modele für das rote, gelbe und grüne Licht erstellt. Die Ampel wird über die Python Datei control_traffic_light.py angesteuert in welcher das Ampelmodel geladen und die Lichter angesteuert werden. Die Ampel wird über das TaskEnvironment selbst gesteuert, da ein eigener Node nicht funktioniert hat.

TaskEnvironment:

Das TaskEnvironment wird im “openai_ros” Paket als Python Datei abgespeichert. Diese Datei erbt von dem RobotEnvironment welche die Verbindung zum Roboter herstellt.

Das TaskEnvironment beschreibt den Kontext mit welchem der Roboter in der Welt lernt. Das bedeutet, dass das TaskEnvironment je nach Roboter und Aufgabe anders ist. Das Trainieren des Roboters selbst wird hier nicht implementiert. Nur die Handlungen und Funktionen, welche in den einzelnen Trainingsschritten verwendet werden, werden implementiert.
Es müssen jedoch immer die Funktionen _set_action, _get_obs, _compute_reward, _init_env_variables, _is_done und _set_init_pose implementiert sein.

_set_action: die nächste Aktion des Roboters wird ausgeführt
_get_obs: die Beobachtungen (Daten der Sensoren) des Roboters nach der Aktion werden abgerufen
_compute_reward: die Belohnung für den Lernvorgang wird berechnet
_init_env_variables: die Variablen, welche nach jeder Episode zurückgesetzt werden müssen, werden geändert
_is_done: entscheiden, ob die Episode als beendet gilt
_set_init_pose: die Position des Roboters, welche nach jeder Episode zurückgesetzt werden muss, wird definiert

Um die Welt in die Simulationsumgebung zu laden, muss das Launch File ausgeführt werden. Dann wird die Yaml Datei mit den benötigten Parametern geladen und die Variablen werden damit befüllt.

Seit Version2 von openai_ros wurde auch die Funktion implementiert, Repositories welche nicht heruntergeladen wurden, selbständig herunterzuladen. Dabei wird in der ROSLaunch Klasse in der Datei openai_ros_common.py geprüft, ob sich das Repository im gegebenen Dateipfad befindet. Sonst wird die DonloadRepo(...) Methode ausgeführt. Wenn man diese Funktionalität weiterhin verwenden möchte, muss man diese Methode warten.
In der aktuellen Version des gazebo_openai_tool wird dies aber nicht benötigt, da alle benötigten Assets bereits vorhanden sind.

Um die selbstgebaute Umgebung verwenden zu können und damit zu trainieren, muss die Umgebung zuerst in OpenAI Gym registriert werden. Das kann in der TaskEnvironment Datei selbst gemacht werden oder man macht es gesammelt in der Datei task_env_list.py um eine bessere Übersicht zu bewahren. Wie in “Neues AI Reinforcement Learning Skript” beschreiben.

Neuen Roboter einfügen:

Um einen neuen Roboter im gazebo_openai_tool verwenden zu können müssen zwei Bedingungen erfüllt sein.

Im "robot_models" Ordner muss die Beschreibung des Roboters mit den zugehörigen Meshes, welche die einzelnen Komponenten wie Räder und dergleichen beschreiben, sowie ein Launch File, um den Roboter in Gazebo zu laden vorliegen.

Im "openai_ros" Ordner muss das zugehörige RobotEnvironment mit allen nötigen ROS Funktionalitäten implementiert sein.

Beschreibung des Roboters:

Die Beschreibung des Roboters wird in einer Xacro Datei gespeichert. Um den Roboter in Gazebo zu laden, werden zuerst drei Schritte ausgeführt.
Als erstes wird die Xacro Datei in eine URDF (Unified Robotic Description Format) Datei umgewandelt.
Als nächstes wird die URDF Datei in eine SDF (Simulation Description Format) Datei umgewandelt.
Zuletzt wird der in der SDF Datei beschriebene Roboter in Gazebo geladen.
Dieser schritte werden bei der Verwendung des Launch Files ausgeführt. Natürlich könnte man die Beschreibung des Roboters auch direkt als URDF oder SDF Datei abspeichern.

Xacro ist eine XML Macro Sprache und erlaubt es die Modularität und Wiederverwendbarkeit von Code beim Erstellen von URDF Modellen zu verbessern. Xacro ist also nur eine andere Art URDF Dateien zu definieren.

In ROS wird das URDF Dateiformat verwendet, um Roboter zu beschreiben. Jedoch haben sich in der Robotik viele neue Features entwickelt, welche von URDF nicht unterstützt werden. Deshalb wandelt Gazebo URDF Dateien in das SDF Dateiformat um, um Roboter in die Simulationsumgebung zu laden.

Im Launch File muss dann nur noch die Position, an welcher der Roboter erscheinen soll und die Xacro Datei angegeben werden. Durch Ausführen des Launch Files wird ein Node vom Typ "spawn_model" aus dem "gazebo_ros" Packet erstellt. Dieser Node führt dann auch die Umwandlung von Xacro zu SDF aus.

Um URDF in Gazebo zu verwenden müssen <gazebo> Elemente verwendet werden [http://gazebosim.org/tutorials/?tut=ros_urdf]. Um die Übersichtlichkeit zu verbessern, wird die Xacro Datei meist in zwei Dateien aufgeteilt. Einen Teil mit allen Gazebo Elementen und einen Teil ohne Gazebo Elemente. In diesem Projekt ist die Datei mit den Gazebo Elementen mit der Endung .gazebo.xacro und der Teil ohne Gazebo Elemente mit der Endung .urdf.xacro zu erkennen.

Bei der Konvertierung von Xacro zu SDF kann es zu Fehlern kommen, da einige Namen von selbst geändert werden. Ein Beispiel ist, dass der Name des Kollisionssensors bumper_sensor_collision bei der Konvertierung auf base_footprint_fixed_joint_lump__bumper_sensor_collision_collision_1 geändert wird.

Xacro Datei:
turtlebot3_burger_for_lane_recognition.urdf.xacro

```
<!-- register contact -->
<joint name="bumper_joint" type="fixed">
  <parent link="base_link"/>
  <child link="bumper_sensor"/>
  <origin xyz="0.0 0.0 0.1" rpy="0 0 0"/>
  <axis xyz="0 0 0"/>
</joint>
<link name="bumper_sensor">
  <collision name="bumper_sensor_collision">
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.11 0.19 0.05"/>
    </geometry>
  </collision>
</link>
```

turtlebot3_burger_for_lane_recognition.gazebo.xacro

```
<gazebo reference="bumper_sensor">
  <!-- contact sensor -->
  <sensor type="contact" name="contact_sensor">
    <update_rate>1000.0</update_rate>
    <always_on>true</always_on>
    <contact>
      <collision>bumper_sensor_collision</collision>
      <topic>/bumper_contact</topic>
    </contact>
    <plugin name="gazebo_ros_bumper_controller" filename="libgazebo_ros_bumper.so">
            <alwaysOn>true</alwaysOn>
            <updateRate>1000.0</updateRate>
            <bumperTopicName>/robot_bumper</bumperTopicName>
            <frameName>world</frameName>
    </plugin>
  </sensor>
</gazebo>
```

URDF Datei: test_urdf.urdf

```
<gazebo reference="bumper_sensor">
  <!-- contact sensor -->
  <sensor name="contact_sensor" type="contact">
    <update_rate>1000.0</update_rate>
    <always_on>true</always_on>
    <contact>
      <collision>bumper_sensor_collision</collision>
      <topic>/bumper_contact</topic>
    </contact>
    <plugin filename="libgazebo_ros_bumper.so" name="gazebo_ros_bumper_controller">
      <alwaysOn>true</alwaysOn>
      <updateRate>1000.0</updateRate>
      <bumperTopicName>/robot_bumper</bumperTopicName>
      <frameName>world</frameName>
    </plugin>
  </sensor>
</gazebo>
```

SDF Datei: test_sdf.sdf

```
<collision name='base_footprint_fixed_joint_lump__bumper_sensor_collision_collision_1'>
  <pose>0 0 0.11 0 -0 0</pose>
  <geometry>
    <box>
      <size>0.11 0.19 0.05</size>
    </box>
  </geometry>
  <surface>
    <contact>
      <ode/>
    </contact>
    <friction>
      <ode/>
    </friction>
  </surface>
</collision>
```

Um solche Fehler zu finden kann man die Umwandlung von Xacro zu URDF und von URDF zu SDF manuell vornehmen und das Ergebnis überprüfen:

rosrun xacro xacro turtlebot3_burger_for_lane_recognition.urdf.xacro --inorder > test_urdf.urdf

gz sdf -p test_urdf.urdf > test_sdf.sdf

Um das Ergebnis ansehen zu können, lädt man den Roboter in Gazebo und öffnet ihn mit dem Gazebo Model Editor. Hier kann man mit der Xacro Datei erkennen welche Links und welche Komponenten wie dargestellt werden.

RobotEnvironment:

Das RobotEnvironment wird im “openai_ros” Paket als Python Datei abgespeichert. Diese Datei erbt von dem GazeboEnvironment welche die Verbindung zu Gazebo herstellt.

Es enthält alle Funktionen, welche zum Trainieren des Roboters benötigt werden. Diese Funktionen müssen alle ROS Funktionalitäten zur verfügung stellen.
Häufig verwendete Funktionen wären zum Beispiel die funktionalität der Sensoren des Roboters überprüfen, Daten der Sensoren abrufen, den Roboter bewegen und Vieles mehr.

Um den Roboter in die Simulationsumgebung zu laden, muss das Launch File ausgeführt werden. Dann wird die Simulation mit self.gazebo.unpauseSim() zum laufen gebracht.
Jetzt kann man die nötigen ROS Topics mit rospy.Publisher() und rospy.Subscriber() einrichten um Daten an den Roboter zu senden und vom Roboter zu empfangen.
Nachdem man geprüft hat ob alle Publisher funktionieren muss die Simulation wieder mit self.gazebo.pauseSim() gestoppt werden um den Initialisierungsvorgang nicht zu beeinträchtigen.