12장 URDF

12장 URDF

12-1. URDF 소개

12-1-1 URDF란?

- **URDF(Unified Robot Description Format)** 는 로봇의 구조와 물리적 속성을 XML 형식으로 정의하는 파일 형식입니다.
- ROS 2 Humble에서 로봇 모델링의 핵심으로 사용되며, 로봇의 링크와 조인트를 기술하여 시각화(RViz2) 및 시뮬레이션(Gazebo)에 활용됩니다.
- 주요 용도:

    - 로봇의 기하학적 구조 정의.
    - RViz2에서 로봇 시각화.
    - Gazebo에서 물리 기반 시뮬레이션.
- 예: 차량, 로봇 팔, 드론 등의 3D 모델을 URDF로 표현.

12-1-2 URDF의 주요 구성 요소

URDF는 로봇의 구조를 `<robot>`, `<link>`, `<joint>` 태그로 정의합니다. 주요 요소는 다음과 같습니다:

- **`<link>`**: 로봇의 물리적 부분(예: 바퀴, 몸체)을 나타냅니다.
    - `<visual>`: 시각적 모양(색상, 형태).
    - `<collision>`: 충돌 검사를 위한 지오메트리.
    - `<inertial>`: 물리적 속성(질량, 관성).
    - 예:

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder radius="0.2" length="0.5"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="0.2" length="0.5"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <inertia ixx="0.1" ixy="0.0" ixz="0.0" iyy="0.1" iyz="0.0" izz="0.
1"/>
  </inertial>
</link>
```

- **`<joint>`**: 링크 간의 연결(예: 고정, 회전, 직선 이동).

    - 속성: `type`(fixed, revolute, prismatic 등), `parent`, `child`, `origin`.
    - 예:

```
<joint name="base_to_arm" type="revolute">
  <parent link="base_link"/>
  <child link="arm_link"/>
  <origin xyz="0 0 0.5" rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
</joint>
```

- `<robot>`: URDF 파일의 루트 요소로, 모든 링크와 조인트를 포함.
    - 예: `<robot name="my_robot">`.

12-1-3 URDF의 XML 구조와 문법

- URDF는 XML 형식으로 작성되며, 계층적 구조를 따릅니다.
- 기본 구조:

```
<?xml version="1.0"?>
<robot name="my_robot">
  <!-- 링크와 조인트 정의 -->
</robot>
```

- 문법 규칙:
    - 모든 태그는 올바르게 닫혀야 함.
    - `<link>`와 `<joint>`는 `<robot>` 안에 정의.
    - `<origin>`는 위치(xyz)와 방향(rpy, 롤-피치-요)을 지정.
    - `<material>`은 `<visual>` 내부 또는 전역으로 정의.
- 참고: http://wiki.ros.org/urdf/XML

12-1-4 ROS 2 Humble에서의 URDF 역할

- **RViz2**: URDF를 `robot_state_publisher`와 함께 사용하여 로봇의 TF 변환을 시각화.
- **Gazebo**: URDF에 물리 속성과 플러그인을 추가하여 시뮬레이션.
- **MoveIt**: URDF를 기반으로 모션 플래닝 설정.

- URDF는 ROS 2의 다양한 도구와 통합되어 로봇 개발의 핵심 역할을 합니다.

12-2. ROS 2 Humble 워크스페이스 설정 및 패키지 생성

12-2-1 ROS 2 Humble 패키지 설치

ROS 2 Humble에서 URDF 작업을 위해 필요한 패키지를 설치합니다.

- **실습 목표**: 필수 패키지(`urdf`, `joint_state_publisher`, `robot_state_publisher`, `rviz2`) 설치.
- **단계**:
  1. 터미널에서 다음 명령어를 실행하여 패키지 설치:

```
sudo apt update
sudo apt install ros-humble-urdf ros-humble-joint-state-publisher r
os-humble-robot-state-publisher ros-humble-rviz2 ros-humble-urd
f-launch
```

12-2-3 새로운 ROS 2 패키지 생성

URDF 파일을 저장하고 실행할 새로운 패키지를 생성합니다. 참고 저장소의 `kuLimo` 패키지 구조를 기반으로 설정합니다.

- **실습 목표**: `ku_description` 패키지 생성 및 `setup.py`, `package.xml` 설정.
- **단계**:
  1. `src` 디렉토리로 이동:

```
cd ~/kuLimo/colcon_ws/src
```

  2. 새로운 패키지 생성:

```
ros2 pkg create --build-type ament_python ku_description
```

     - `ament_python`: Python 기반 패키지.

12-2-4 `package.xml` 설정

`package.xml`은 패키지의 메타데이터와 의존성을 정의합니다. 참고 저장소의 `kuLimo` 패키지 `package.xml`을 기반으로 수정합니다.

- **실습 목표**: URDF 관련 의존성 추가.

- **단계**:
  1. `package.xml` 열기:

     ```
     nano ~/kuLimo/colcon_ws/src/ku_description/package.xml
     ```

  2. `<depend>` 태그 추가하여 URDF 관련 패키지 의존성 명시:

     ```
     <depend>urdf</depend>
     <depend>joint_state_publisher</depend>
     <depend>robot_state_publisher</depend>
     <depend>rviz2</depend>
     ```

  3. 예시 `package.xml` (참고 저장소 기반):

     ```
     <?xml version="1.0"?>
     <package format="3">
      <name>my_robot_description</name>
      <version>0.0.0</version>
      <description>URDF tutorial package for ROS 2 Humble</descripti
     on>
      <maintainer email="user@example.com">Your Name</maintainer
     >
      <license>Apache-2.0</license>
      <buildtool_depend>ament_python</buildtool_depend>
      <depend>rclpy</depend>
      <depend>std_msgs</depend>
      <depend>urdf</depend>
      <depend>joint_state_publisher</depend>
      <depend>robot_state_publisher</depend>
      <depend>rviz2</depend>
      <test_depend>ament_copyright</test_depend>
      <test_depend>ament_flake8</test_depend>
      <test_depend>ament_pep257</test_depend>
      <export>
        <build_type>ament_python</build_type>
      </export>
     </package>
     ```

12-2-5 `setup.py` 설정

`setup.py`는 Python 패키지의 설치 및 실행 파일을 정의합니다. 참고 저장소의 `setup.py`를 기반으로 설정합니다.

- **실습 목표**: 패키지 메타데이터 및 실행 파일 설정.
- **단계**:
  1. `setup.py` 열기: VsCode
  2. `package.xml`과 일치하도록 메타데이터 수정 및 실행 파일 추가:

```
import os
from glob import glob

from setuptools import find_packages, setup

package_name = "ku_description"

setup(
    name=package_name,
    version="0.0.0",
    packages=find_packages(exclude=["test"]),
    data_files=[
        ("share/ament_index/resource_index/packages", ["resource/"
+ package_name]),
        ("share/" + package_name, ["package.xml"]),
        (
            "share/" + package_name + "/launch",
            glob(os.path.join("launch", "*.launch.py")),
        ),
        (
            "share/" + package_name + "/urdf",
            glob(os.path.join("urdf", "*.*")),
        ),
    ],
    install_requires=["setuptools"],
    zip_safe=True,
    maintainer="aa",
    maintainer_email="freshmea@naver.com",
```

```
    description="TODO: Package description",
    license="TODO: License declaration",
    tests_require=["pytest"],
    entry_points={
        "console_scripts": [],
    },
)
```

3. 참고: `entry_points`는 나중에 Python 노드를 추가할 때 사용됩니다. 현재는 비어 있음.

12-2-6 패키지 빌드 및 테스트

```
# alias 된 명령어를 이용한다.
cb
sb
```

12-2-7 URDF 디렉토리 생성

URDF 파일을 저장할 디렉토리를 생성합니다. 참고 저장소의 `urdf` 디렉토리를 기반으로 합니다.

- **단계**:
  1. `urdf` 디렉토리 생성: VsCode 사용
  2. 빈 URDF 파일 생성(다음 수업에서 작성): `myfirst.urdf` 생성

12-2. 간단한 URDF 모델 작성

12-2-2 이론: URDF 모델 작성 기초

단일 링크 로봇 정의

- URDF 파일은 로봇의 구조를 XML로 정의하며, 단일 링크 로봇은 가장 기본적인 모델입니다.

- **링크(`<link>`)**: 로봇의 물리적 부분(예: 원통형 몸체)을 나타냅니다.
  - `<visual>`: RViz2에서 보이는 시각적 모양(예: 색상, 형태).
  - `<collision>`: 시뮬레이션에서 충돌 검사를 위한 지오메트리.
  - `<inertial>`: 물리적 속성(질량, 관성, Gazebo에서 필요). 이 세션에서는 생략 가능.
- 예시: 단순한 urdf

```
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link"> <!-- tf 이름 -->
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2" />
      </geometry>
    </visual>
  </link>
</robot>
```

- 예시: 원통형 링크

```
<link name="base_link">
 <visual>
  <geometry>
   <cylinder radius="0.2" length="0.5"/>
  </geometry>
  <material name="blue"/>
 </visual>
 <collision>
  <geometry>
   <cylinder radius="0.2" length="0.5"/>
  </geometry>
 </collision>
</link>
```

색상 설정 (`<material>`)

- `<material>` 태그는 시각적 요소에 색상 또는 텍스처를 지정합니다.

- 전역 또는 `<visual>` 내부에서 정의 가능.
- 예시: 파란색 재질 정의

```
<material name="blue">
  <color rgba="0 0 1 1"/>
</material>
```

- `rgba`: 빨강, 초록, 파랑, 투명도(0~1).

URDF 파일 구조

- 기본 구조:

```
<?xml version="1.0"?>
<robot name="my_robot">
  <!-- 링크 정의 -->
</robot>
```

- 단일 링크 로봇은 `<joint>` 없이도 시각화 가능.

RViz2에서 URDF 시각화

- `robot_state_publisher`: URDF를 읽어 TF 변환을 발행.
- `joint_state_publisher`: 조인트 상태를 시뮬레이션(단일 링크는 필요 없음).
- RViz2에서 `RobotModel` 디스플레이를 추가하여 URDF 시각화.
- 명령어:

```
ros2 run rviz2 rviz2
ros2 run robot_state_publisher robot_state_publisher --ros-args -p robo
t_description:="$(cat myfirst.urdf)"
```

12-2-3 실습: 간단한 URDF 모델 작성 및 시각화

실습 목표

- `ku_description` 패키지의 `urdf` 디렉토리에 `myfirst.urdf` 파일 작성.
- 원통형 단일 링크 로봇 정의(시각적 및 충돌 속성 포함).

- RViz2에서 모델 시각화.

단계

1. **URDF 파일 작성**
   - `ku_description/urdf` 디렉토리에 `myfirst.urdf` 파일을 작성합니다.
   - 명령어:

     ```
     nano ~/kuLimo/colcon_ws/src/ku_description/urdf/myfirst.urdf
     ```

   - 내용:

     ```
     <?xml version="1.0"?>
     <robot name="my_robot">
      <material name="blue">
       <color rgba="0 0 1 1"/>
      </material>
      <link name="base_link">
       <visual>
        <geometry>
         <cylinder radius="0.2" length="0.5"/>
        </geometry>
        <material name="blue"/>
       </visual>
       <collision>
        <geometry>
         <cylinder radius="0.2" length="0.5"/>
        </geometry>
       </collision>
      </link>
     </robot>
     ```

   - 설명:
     - `<robot name="my_robot">`: 로봇 이름 정의.
     - `<material name="blue">`: 전역 파란색 재질 정의.
     - `<link name="base_link">`: 원통형 링크(반지름 0.2m, 길이 0.5m).
     - `<visual>`: 시각적 모양(파란색 원통).

- `<collision>`: 충돌 지오메트리(동일한 원통).

2. **패키지 빌드**
   - 빌드:

```
# 패키지 하나만 빌드 하고 싶다면
cd ~/kuLimo/colcon_ws
colcon build --packages-select ku_description
# alias 사용
cb
```

3. **런치 파일 생성**
   - `ku_description/launch` 디렉토리에 런치 파일 작성.
   - 명령어: VsCode
   - 내용:

```
from launch import LaunchDescription
from launch_ros.actions import Node
import os
from ament_index_python.packages import get_package_share_dir
ectory

def generate_launch_description():
    package_name = 'ku_description'
    urdf_file = os.path.join(get_package_share_directory(package_na
me), 'urdf', 'myfirst.urdf')

    with open(urdf_file, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            parameters=[{'robot_description': robot_desc}]
```

```
        ),
        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            output='screen'
        )
    ])
```

- 설명:
  - `robot_state_publisher`: URDF 파일을 읽어 TF 변환 발행.
  - `rviz2`: RViz2 실행.
  - URDF 파일을 `ku_description/urdf/myfirst.urdf`에서 로드.
- urdf_launch 활용

```
# sudo apt install ros-humble-urdf-launch
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescr
from launch.substitutions import LaunchConfiguration, PathJoinSubstitutio
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    default_model_path = PathJoinSubstitution(["urdf", "myfirst.urdf"])
    model = DeclareLaunchArgument(
        name="model", default_value=default_model_path, description="myfi
    )
    return LaunchDescription(
        [
            model,
            IncludeLaunchDescription(
                PathJoinSubstitution(
                    [FindPackageShare("urdf_launch"), "launch", "display.launch.p
                ),
                launch_arguments={
                    "urdf_package": "ku_description",
```

```
                "urdf_package_path": LaunchConfiguration("model"),
            }.items(),
        ),
    ]
)
```

4. **패키지 빌드 및 실행**
   - 런치 파일 실행:

     ```
     ros2 launch ku_description display.launch.py
     ```

   - RViz2에서:
     - `Add` → `By topic` → `RobotModel` 디스플레이 추가.
     - `Fixed Frame`을 `base_link`로 설정.
     - 파란색 원통이 나타나는지 확인.
5. **문제 해결**
   - URDF 파일 오류(예: 태그 누락, 잘못된 속성) 시, 터미널 에러 메시지 확인.
   - RViz2에서 모델이 안 보일 경우:
     - `robot_description` 토픽 확인: `ros2 topic list | grep robot_description`
     - URDF 파일 경로 확인: `ls ~/kuLimo/colcon_ws/install/ku_description/share/ku_description/urdf`

12-3. 조인트 및 다중 링크 추가

12-3-2 이론: 조인트 및 다중 링크 로봇

조인트 (`<joint>`) 정의

- `<joint>` 태그는 두 링크 간의 연결을 정의하며, 로봇의 동적 또는 고정된 움직임을 표현합니다.
- 주요 속성:
  - `name`: 조인트 이름.

    - `type`: 조인트 유형.
        - `fixed`: 고정된 연결, 움직임 없음.
        - `continuous`: 무제한 회전 조인트(예: 바퀴).
        - `revolute`: 제한된 회전 조인트.
        - `prismatic`: 직선 이동 조인트.
    - `parent`: 부모 링크.
    - `child`: 자식 링크.
    - `origin`: 조인트의 위치(xyz)와 방향(rpy, 롤-피치-요).
    - `axis`: 회전 또는 이동 축(continuous/revolute/prismatic에 필요).
- 예시: 고정 조인트

```
<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
</joint>
```

- 예시: 연속 조인트

```
<joint name="right_front_wheel_joint" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="right_leg"/>
  <child link="right_front_wheel"/>
  <origin xyz="0 0 -0.65"/>
</joint>
```

다중 링크 로봇의 부모-자식 관계

- URDF는 트리 구조로 링크를 연결합니다.
- 부모 링크(`parent`)와 자식 링크(`child`)는 `<joint>`로 연결.
- `<origin>`은 자식 링크의 좌표계를 부모 링크 기준으로 정의:
    - `xyz`: x, y, z 위치(미터).
    - `rpy`: 롤, 피치, 요 회전(라디안).

- `visual.urdf`의 트리 구조: `base_link` → `right_leg` → `right_front_wheel`, `base_link` → `left_leg` → `left_front_wheel`.

URDF 모델 확장

- 단일 링크(`origin.urdf`: 원통 + 다리)에서 다중 링크(`visual.urdf`: 원통, 좌우 다리, 바퀴)로 확장.
- `<material>`로 색상 추가(파랑, 흰색, 검정).
- RViz2에서 `robot_state_publisher`로 TF 변환, `joint_state_publisher_gui`로 연속 조인트(바퀴) 상태 테스트.

URDF 검증 도구

- **check_urdf**: URDF 파일의 구문과 구조를 검증.
  - 명령어: `check_urdf path/to/urdf_file`
- **rqt_tf_tree**: TF 트리 시각화로 부모-자식 관계 확인.
  - 명령어: `ros2 run rqt_tf_tree rqt_tf_tree`

URDF 시각화 워크플로우

- URDF 파일 작성 → `robot_state_publisher`로 TF 변환 발행 → `joint_state_publisher_gui`로 조인트 상태 발행 → RViz2에서 `RobotModel` 디스플레이로 시각화.

12-3-3 실습: 다중 링크 URDF 모델 작성 및 시각화 (1시간)

실습 목표

- `ku_description` 패키지의 `urdf` 디렉토리에 제공된 `visual.urdf` 사용.
- 다중 링크 로봇(원통 몸체, 좌우 다리, 바퀴) 시각화.
- `check_urdf`와 `rqt_tf_tree`로 URDF와 TF 트리 검증.

단계

1. **URDF 파일 확인**
   - 제공된 `visual.urdf` 파일은 `ku_description/urdf` 디렉토리에 있다고 가정.
   - `visual.urdf` 내용:

```
<?xml version="1.0"?>
<robot name="myfirst">
 <material name="white">
  <color rgba="1 1 1 1"/>
 </material>
 <material name="blue">
  <color rgba="0 0 0.8 1"/>
 </material>
 <material name="black">
  <color rgba="0 0 0 1"/>
 </material>
 <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="0.6" radius="0.2"/>
   </geometry>
   <material name="blue"/>
  </visual>
 </link>
 <link name="right_leg">
  <visual>
   <geometry>
    <box size="0.6 0.1 0.2"/>
   </geometry>
   <origin rpy="0 1.57 0" xyz="0 0 -0.3"/>
   <material name="white"/>
  </visual>
 </link>
 <link name="right_front_wheel">
  <visual>
   <geometry>
    <cylinder length="0.1" radius="0.035"/>
   </geometry>
   <origin rpy="1.57 0 0" xyz="0 0 0"/>
   <material name="black"/>
  </visual>
 </link>
 <joint name="right_front_wheel_joint" type="continuous">
```

```
  <axis xyz="0 1 0"/>
  <parent link="right_leg"/>
  <child link="right_front_wheel"/>
  <origin xyz="0 0 -0.65"/>
</joint>
<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
</joint>
<link name="left_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>
<link name="left_front_wheel">
  <visual>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <origin rpy="1.57 0 0" xyz="0 0 0"/>
    <material name="black"/>
  </visual>
</link>
<joint name="left_front_wheel_joint" type="continuous">
  <axis xyz="0 1 0"/>
  <parent link="left_leg"/>
  <child link="left_front_wheel"/>
  <origin xyz="0 0 -0.65"/>
</joint>
<joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="0 0.22 0.25"/>
```

```
  </joint>
</robot>
```

- 설명:
  - `base_link`: 파란색 원통(길이 0.6m, 반지름 0.2m).
  - `right_leg`, `left_leg`: 흰색 박스(0.6m x 0.1m x 0.2m).
  - `right_front_wheel`, `left_front_wheel`: 검은색 바퀴(길이 0.1m, 반지름 0.035m).
  - 고정 조인트(`base_to_right_leg`, `base_to_left_leg`)와 연속 조인트(`right_front_wheel_joint`, `left_front_wheel_joint`).
- 실행:

```
ros2 launch ku_description display.launch.py model:=urdf/visual.urdf
```

2. **URDF 구조 검증**
   - `check_urdf` 설치:

     ```
     sudo apt install liburdfdom-tools
     ```

   - `visual.urdf` 검증:

     ```
     check_urdf ~/kuLimo/colcon_ws/src/ku_description/urdf/visual.urdf
     ```

   - 예상 출력:

     ```
     robot name is: myfirst
     ---------- Successfully Parsed XML ---------------
     root Link: base_link has 2 child(ren)
         child(1): right_leg
             child(1): right_front_wheel
         child(2): left_leg
             child(1): left_front_wheel
     ```

   - 오류 발생 시(예: 태그 누락, 잘못된 링크 이름), 파일 수정 후 재검증.
3. **런치 파일 실행**
   - `visual.urdf` 시각화:

```
ros2 launch ku_description display.launch.py model:=urdf/visual.urdf
```

- RViz2에서:
  - `Add` → `By topic` → `RobotModel` 디스플레이 추가.
  - `Fixed Frame`을 `base_link`로 설정.
  - 파란색 원통, 흰색 좌우 다리, 검은색 바퀴 확인.
  - `JointStatePublisherGui`로 바퀴 회전 테스트.

4. **TF 트리 검증**
   - `rqt_tf_tree` 실행:

```
ros2 run rqt_tf_tree rqt_tf_tree
```

   - TF 트리 확인:
     - 예상 구조: `base_link` → `right_leg` → `right_front_wheel`, `base_link` → `left_leg` → `left_front_wheel`.
     - 연결 누락 시, `<joint>`의 `parent`/`child` 확인.
5. **문제 해결**
   - URDF 파일 오류:
     - `check_urdf`로 구문 오류 확인.
     - 예: 잘못된 `<parent>` 또는 `<child>` 링크 이름.
   - RViz2에서 모델 안 보일 경우:
     - `robot_description` 토픽 확인:

```
ros2 topic echo /robot_description
```

     - URDF 파일 경로 확인:

```
ls ~/kuLimo/colcon_ws/install/ku_description/share/ku_description/urdf
```

   - TF 트리 오류:

- `rqt_tf_tree`에서 연결 확인.
- `joint_state_publisher_gui` 실행 확인:

```
ros2 node list | grep joint_state_publisher_gui
```

12-4. Xacro를 활용한 URDF 개선

12-4-2 이론: Xacro 기초 및 문법

Xacro란?

- **Xacro(XML Macros)**는 URDF를 간소화하고 재사용 가능하게 만드는 XML 기반 매크로 언어입니다.
- ROS 2 Humble에서 Xacro는 URDF의 복잡성을 줄이고 유지보수를 쉽게 합니다.
- 주요 기능:
  - **상수 정의**: 반복되는 값을 변수로 관리.
  - **매크로**: 반복적인 XML 코드를 함수처럼 재사용.
  - **블록 삽입**: 동적으로 XML 블록 삽입.
  - **파일 포함**: 다른 Xacro 파일을 포함하여 모듈화.
- 변환: Xacro 파일(`.xacro`)은 `ros2 run xacro xacro` 명령어로 URDF로 변환.

Xacro 문법 자세히

1. **상수 정의 (`<xacro:property>`)**:
   - 반복되는 값(예: 치수, 색상)을 변수로 정의.
   - 구문: `<xacro:property name="변수명" value="값"/>`
   - 사용: `${변수명}`으로 참조.
   - 예:

```
<xacro:property name="width" value="0.2"/>
<cylinder length="0.6" radius="${width}"/>
```

   - 제공된 코드에서:

```
<xacro:property name="width" value="0.2"/>
<xacro:property name="bodylen" value="0.6"/>
```

- `width`와 `bodylen`은 원통의 치수를 정의하며, 변경 시 한 곳만 수정.

2. **매크로 정의 (`<xacro:macro>`)**:
   - 반복적인 XML 코드를 함수처럼 정의.
   - 구문:

```
<xacro:macro name="매크로명" params="파라미터">
 <!-- XML 코드 -->
</xacro:macro>
<xacro:매크로명 파라미터="값"/>
```

   - 제공된 코드에서:

```
<xacro:macro name="default_inertial" params="mass *shape">
 <collision>
  <xacro:insert_block name="shape"/>
 </collision>
 <inertial>
  <mass value="${mass}"/>
  <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz
="1e-3"/>
 </inertial>
</xacro:macro>
```

     - `default_inertial` 매크로: `<collision>`과 `<inertial>`을 재사용.
     - `mass`: 링크의 질량.
     - `shape`: `<geometry>` 블록을 동적으로 삽입.

3. **블록 삽입 (`<xacro:insert_block>`)**:
   - 매크로 내에서 동적으로 XML 블록 삽입.
   - 구문: `<xacro:insert_block name="블록명"/>`
   - 제공된 코드에서:

```
<xacro:default_inertial mass="10">
  <geometry>
    <cylinder length="${bodylen}" radius="${width}"/>
  </geometry>
</xacro:default_inertial>
```

- `<geometry>` 블록이 `shape`로 매크로에 전달되어 `<collision>`에 삽입.

4. **네임스페이스 선언**:
   - Xacro 파일은 `<robot>` 태그에 `xmlns:xacro="http://www.ros.org/wiki/xacro"`를 포함.
   - 제공된 코드에서:

```
<robot xmlns:xacro="http://www.ros-org/wiki/xacro" name="myfirst">
```

5. **기타 문법**:
   - **조건문 (`<xacro:if>`)**: 조건에 따라 코드 포함/제외.
     - 예: `<xacro:if condition="${use_collision}"> ... </xacro:if>`
   - **인수 전달 (`<xacro:arg>`)**: 런치 파일에서 동적 파라미터 전달.
     - 예: `<xacro:arg name="robot_name" default="myfirst"/>`
   - **파일 포함 (`<xacro:include>`)**: 다른 Xacro 파일 포함.
     - 예: `<xacro:include filename="materials.xacro"/>`

Xacro의 장점

- **코드 중복 감소**: 매크로로 반복 코드 제거.
- **유지보수 용이**: 상수로 치수 관리, 한 곳 수정으로 전체 반영.
- **모듈화**: 복잡한 로봇을 여러 파일로 분리 가능.

Xacro에서 URDF 시각화

- Xacro 파일을 URDF로 변환:

```
ros2 run xacro xacro my_robot.xacro > my_robot.urdf
```

- `robot_state_publisher`와 `urdf_launch`로 RViz2에서 시각화.

12-4-3 실습: Xacro로 URDF 모델 작성 및 시각화

실습 목표

- `ku_description` 패키지의 `urdf` 디렉토리에 제공된 Xacro 파일 사용.
- Xacro로 모듈화된 로봇 모델(원통 몸체, 좌우 다리, 바퀴) 시각화.
- `check_urdf`와 `rqt_tf_tree`로 URDF와 TF 트리 검증.

단계

1. **Xacro 패키지 설치**
   - 설치 확인:

     ```
     sudo apt install ros-humble-xacro
     ```

2. **Xacro 파일 확인**
   - 제공된 Xacro 파일은 `ku_description/urdf/model.xacro`
   - 내용:

     ```
     <?xml version="1.0"?>
     <robot xmlns:xacro="http://www.ros-org/wiki/xacro" name="myfirs
     t">
      <xacro:property name="width" value="0.2"/>
      <xacro:property name="bodylen" value="0.6"/>
      <xacro:macro name="default_inertial" params="mass *shape">
       <collision>
        <xacro:insert_block name="shape"/>
       </collision>
       <inertial>
        <mass value="${mass}"/>
        <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz
     ="1e-3"/>
       </inertial>
      </xacro:macro>
      <material name="white">
       <color rgba="1 1 1 1"/>
      </material>
     ```

```
<material name="blue">
  <color rgba="0 0 0.8 1"/>
</material>
<material name="black">
  <color rgba="0 0 0 1"/>
</material>
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="${bodylen}" radius="${width}"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <xacro:default_inertial mass="10">
    <geometry>
      <cylinder length="${bodylen}" radius="${width}"/>
    </geometry>
  </xacro:default_inertial>
</link>
<link name="right_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
  <xacro:default_inertial mass="1">
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
  </xacro:default_inertial>
</link>
<link name="right_front_wheel">
  <visual>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
```

```
    <origin rpy="1.57 0 0" xyz="0 0 0"/>
    <material name="black"/>
   </visual>
   <xacro:default_inertial mass="1">
    <geometry>
     <cylinder length="0.1" radius="0.035"/>
    </geometry>
   </xacro:default_inertial>
  </link>
  <joint name="right_front_wheel_joint" type="continuous">
   <axis xyz="0 1 0"/>
   <parent link="right_leg"/>
   <child link="right_front_wheel"/>
   <origin xyz="0 0 -0.65"/>
  </joint>
  <joint name="base_to_right_leg" type="fixed">
   <parent link="base_link"/>
   <child link="right_leg"/>
   <origin xyz="0 -0.22 0.25"/>
  </joint>
  <link name="left_leg">
   <visual>
    <geometry>
     <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57 0" xyz="0 0 -0.3"/>
    <material name="white"/>
   </visual>
   <xacro:default_inertial mass="10">
    <geometry>
     <box size="0.6 0.1 0.2"/>
    </geometry>
   </xacro:default_inertial>
  </link>
  <link name="left_front_wheel">
   <visual>
    <geometry>
     <cylinder length="0.1" radius="0.035"/>
```

```
    </geometry>
    <origin rpy="1.57 0 0" xyz="0 0 0"/>
    <material name="black"/>
   </visual>
   <xacro:default_inertial mass="1">
    <geometry>
     <cylinder length="0.1" radius="0.035"/>
    </geometry>
   </xacro:default_inertial>
  </link>
  <joint name="left_front_wheel_joint" type="continuous">
   <axis xyz="0 1 0"/>
   <parent link="left_leg"/>
   <child link="left_front_wheel"/>
   <origin xyz="0 0 -0.65"/>
  </joint>
  <joint name="base_to_left_leg" type="fixed">
   <parent link="base_link"/>
   <child link="left_leg"/>
   <origin xyz="0 0.22 0.25"/>
  </joint>
</robot>
```

- 설명:
  - `<xacro:property>`: `width`(0.2m), `bodylen`(0.6m)으로 원통 치수 정의.
  - `<xacro:macro name="default_inertial">`: `<collision>`과 `<inertial>`을 재사용, 질량과 지오메트리 블록 삽입.
  - 링크: `base_link`(파란색 원통), `right_leg` / `left_leg`(흰색 박스), `right_front_wheel` / `left_front_wheel`(검은색 바퀴).
  - 조인트: 고정(`base_to_right_leg`, `base_to_left_leg`), 연속(`right_front_wheel_joint`, `left_front_wheel_joint`).

3. **Xacro 파일 검증**
   - Xacro를 URDF로 변환 후 검증:

```
ros2 run xacro xacro ~/kuLimo/colcon_ws/src/ku_description/urdf/model.xacro > /tmp/model.urdf
```

```
check_urdf /tmp/model.urdf
```

- 예상 출력:

```
robot name is: model
---------- Successfully Parsed XML ---------------
root Link: base_link has 2 child(ren)
    child(1): right_leg
        child(1): right_front_wheel
    child(2): left_leg
        child(1): left_front_wheel
```

4. **패키지 빌드 및 실행**
   - 런치 파일 실행:

```
ros2 launch ku_description display.launch.py model:=urdf/model.xacro
```

   - RViz2에서:
     - `Add` → `By topic` → `RobotModel` 디스플레이 추가.
     - `Fixed Frame`을 `base_link`로 설정.
     - 파란색 원통, 흰색 좌우 다리, 검은색 바퀴 확인.
     - `JointStatePublisherGui`로 바퀴 회전 테스트.
5. **TF 트리 검증**
   - `rqt_tf_tree` 실행:

```
ros2 run rqt_tf_tree rqt_tf_tree
```

   - TF 트리 확인:
     - 예상 구조: `base_link` → `right_leg` → `right_front_wheel`, `base_link` → `left_leg` → `left_front_wheel`.
     - 연결 누락 시, `<joint>`의 `parent`/`child` 확인.
6. **문제 해결**
   - Xacro 파일 오류(예: 매크로 파라미터 누락, 상수 오타):

    - 변환 테스트:

      ```
      ros2 run xacro xacro ~/kuLimo/colcon_ws/src/ku_description/urdf/myfirst.xacro
      ```

    - 에러 메시지 확인.
- RViz2에서 모델 안 보일 경우:
    - `robot_description` 토픽 확인:

      ```
      ros2 topic echo /robot_description
      ```

    - 파일 경로 확인:

      ```
      ls ~/kuLimo/colcon_ws/install/ku_description/share/ku_description/urdf
      ```

- TF 트리 오류:
    - `rqt_tf_tree`에서 연결 확인.
    - `joint_state_publisher_gui` 실행 확인:

      ```
      ros2 node list | grep joint_state_publisher_gui
      ```

12-5. Xacro 고도화 및 매크로 활용

12-5-2 이론: Xacro 고도화

Xacro 고도화 전략

- **매크로 활용**: 반복적인 `<link>`와 `<joint>`를 매크로로 정의하여 코드 중복 제거.
- **상수 확장**: 치수, 오프셋, 질량 등을 상수로 관리하여 수정 용이성 증대.
- **조건문 사용**: `<xacro:if>`로 특정 조건에 따라 요소 포함/제외.
- **모듈화**: 복잡한 로봇 구조를 매크로와 상수로 간소화.

Xacro 문법 복습 및 확장

1. **상수 정의 (`<xacro:property>`)**:
   - 반복되는 값(치수, 오프셋, 질량)을 변수로 정의.
   - 예: `<xacro:property name="leg_width" value="0.1"/>`
   - 제공된 코드에서: `width`(0.2m), `bodylen`(0.6m).
2. **매크로 정의 (`<xacro:macro>`)**:
   - 반복적인 XML 코드를 함수처럼 정의.
   - 제공된 코드에서: `default_inertial` 매크로가 `<collision>`과 `<inertial>`을 처리.
   - 확장: `leg_macro`와 `wheel_macro`를 추가하여 좌우 대칭 링크/조인트 정의.
3. **블록 삽입 (`<xacro:insert_block>`)**:
   - 동적으로 XML 블록 삽입.
   - 제공된 코드에서: `<xacro:insert_block name="shape"/>`로 지오메트리 삽입.
4. **조건문 (`<xacro:if>`)**:
   - 조건에 따라 XML 코드 포함/제외.
   - 구문:

```
<xacro:if condition="조건">
 <!-- 포함할 코드 -->
</xacro:if>
```

   - 예: `<inertial>` 태그를 시뮬레이션(Gazebo)에서만 포함.

```
<xacro:if condition="${use_inertial}">
 <inertial>
  <mass value="10"/>
  <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz
="1e-3"/>
 </inertial>
</xacro:if>
```

5. **네임스페이스**:
   - `<robot xmlns:xacro="http://www.ros.org/wiki/xacro">`로 Xacro 활성화.

Xacro의 고도화 이점

- **재사용성**: 매크로로 좌우 대칭 구조(예: 다리, 바퀴) 간소화.
- **유연성**: 상수와 조건문으로 동적 설정 가능.
- **유지보수**: 단일 상수 수정으로 전체 모델 변경.

Xacro 시각화 워크플로우

- Xacro → URDF 변환: `ros2 run xacro xacro advanced_myfirst.xacro > output.urdf`
- `robot_state_publisher`와 `joint_state_publisher_gui`로 TF 변환 및 조인트 상태 발행.
- RViz2에서 `RobotModel` 디스플레이로 시각화.

12-5-3 실습: 고도화된 Xacro 모델 작성 및 시각화 (1시간 15분)

실습 목표

- `ku_description` 패키지의 `urdf` 디렉토리에 `advanced_myfirst.xacro` 작성.
- `leg_macro`와 `wheel_macro`를 추가하여 좌우 대칭 구조 모듈화.
- `<xacro:if>`로 `<inertial>` 태그 선택적 포함.
- RViz2에서 시각화 및 `check_urdf`, `rqt_tf_tree`로 검증.

단계

1. **Xacro 파일 작성**
   - `ku_description/urdf` 디렉토리에 `advanced_myfirst.xacro` 작성.
   - 명령어:

     ```
     nano ~/kuLimo/colcon_ws/src/ku_description/urdf/advanced_myfirst.xacro
     ```

   - 내용:

     ```
     <?xml version="1.0"?>
     <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="myfirst">
       <!-- 상수 정의 -->
       <xacro:property name="body_length" value="0.6"/>
     ```

```
  <xacro:property name="body_radius" value="0.2"/>
  <xacro:property name="leg_length" value="0.6"/>
  <xacro:property name="leg_width" value="0.1"/>
  <xacro:property name="leg_height" value="0.2"/>
  <xacro:property name="wheel_length" value="0.1"/>
  <xacro:property name="wheel_radius" value="0.035"/>
  <xacro:property name="leg_offset_y" value="0.22"/>
  <xacro:property name="leg_offset_z" value="0.25"/>
  <xacro:property name="wheel_offset_z" value="-0.65"/>
  <xacro:property name="use_inertial" value="true"/>

  <!-- 재질 정의 -->
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>
  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>
  <material name="black">
    <color rgba="0 0 0 1"/>
  </material>

  <!-- 매크로 정의 -->
  <xacro:macro name="default_inertial" params="mass *shape">
    <collision>
      <xacro:insert_block name="shape"/>
    </collision>
    <xacro:if condition="${use_inertial}">
      <inertial>
        <mass value="${mass}"/>
        <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" iz
z="1e-3"/>
      </inertial>
    </xacro:if>
  </xacro:macro>

  <xacro:macro name="leg_macro" params="side">
    <link name="${side}_leg">
```

```
    <visual>
     <geometry>
      <box size="${leg_length} ${leg_width} ${leg_height}"/>
     </geometry>
     <origin rpy="0 1.57 0" xyz="0 0 -0.3"/>
     <material name="white"/>
    </visual>
    <xacro:default_inertial mass="1">
     <geometry>
      <box size="${leg_length} ${leg_width} ${leg_height}"/>
     </geometry>
    </xacro:default_inertial>
   </link>
   <joint name="base_to_${side}_leg" type="fixed">
    <parent link="base_link"/>
    <child link="${side}_leg"/>
    <origin xyz="0 ${side == 'right' ? -leg_offset_y : leg_offset_y}
${leg_offset_z}"/>
   </joint>
  </xacro:macro>

  <xacro:macro name="wheel_macro" params="side">
   <link name="${side}_front_wheel">
    <visual>
     <geometry>
      <cylinder length="${wheel_length}" radius="${wheel_radiu
s}"/>
     </geometry>
     <origin rpy="1.57 0 0" xyz="0 0 0"/>
     <material name="black"/>
    </visual>
    <xacro:default_inertial mass="1">
     <geometry>
      <cylinder length="${wheel_length}" radius="${wheel_radiu
s}"/>
     </geometry>
    </xacro:default_inertial>
   </link>
```

```
  <joint name="${side}_front_wheel_joint" type="continuous">
   <axis xyz="0 1 0"/>
   <parent link="${side}_leg"/>
   <child link="${side}_front_wheel"/>
   <origin xyz="0 0 ${wheel_offset_z}"/>
  </joint>
 </xacro:macro>

 <!-- 링크 및 조인트 정의 -->
 <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="${body_length}" radius="${body_radius}"/>
   </geometry>
   <material name="blue"/>
  </visual>
  <xacro:default_inertial mass="10">
   <geometry>
    <cylinder length="${body_length}" radius="${body_radius}"/>
   </geometry>
  </xacro:default_inertial>
 </link>

 <!-- 좌우 다리 및 바퀴 추가 -->
 <xacro:leg_macro side="right"/>
 <xacro:leg_macro side="left"/>
 <xacro:wheel_macro side="right"/>
 <xacro:wheel_macro side="left"/>
</robot>
```

- 설명:
  - **상수 확장**: `body_length`, `leg_width`, `wheel_radius` 등 모든 치수와 오프셋을 상수로 정의.
  - **매크로 추가**:
    - `leg_macro`: 좌우 다리(`right_leg`, `left_leg`)와 고정 조인트를 정의. `side` 파라미터로 대칭 처리.
    - `wheel_macro`: 바퀴(`right_front_wheel`, `left_front_wheel`)와 연속 조인트를 정의.

- **조건문**: `use_inertial` 상수로 `<inertial>` 태그를 선택적으로 포함(RViz2 시각화는 필요 없음, Gazebo 시뮬레이션에서 사용).
- **개선점**: 제공된 Xacro 파일 대비 더 많은 매크로와 상수로 코드 간소화, 좌우 대칭 구조를 단일 매크로 호출로 처리.

2. **패키지 설정 확인**
   - `package.xml`에 `xacro`와 `joint_state_publisher_gui` 의존성 확인.
   - `setup.py`는 `urdf` 디렉토리의 `.xacro` 파일을 포함하므로 수정 불필요.
3. **Xacro 파일 검증**
   - Xacro를 URDF로 변환 후 검증:

```
ros2 run xacro xacro ~/kuLimo/colcon_ws/src/ku_description/urdf/advanced_myfirst.xacro > /tmp/advanced_myfirst.urdf
check_urdf /tmp/advanced_myfirst.urdf
```

   - 예상 출력:

```
robot name is: myfirst
---------- Successfully Parsed XML ---------------
root Link: base_link has 2 child(ren)
    child(1): right_leg
        child(1): right_front_wheel
    child(2): left_leg
        child(1): left_front_wheel
```

4. **패키지 빌드 및 실행**
   - 런치 파일 실행:

```
ros2 launch ku_description display.launch.py model:=urdf/advanced_myfirst.xacro
```

   - RViz2에서:
     - `Add` → `By topic` → `RobotModel` 디스플레이 추가.
     - `Fixed Frame`을 `base_link`로 설정.
     - 파란색 원통, 흰색 좌우 다리, 검은색 바퀴 확인.

- `JointStatePublisherGui`로 바퀴 회전 테스트.

5. **TF 트리 검증**
   - `rqt_tf_tree` 실행:

     ```
     ros2 run rqt_tf_tree rqt_tf_tree
     ```

   - TF 트리 확인:
     - 예상 구조: `base_link` → `right_leg` → `right_front_wheel`, `base_link` → `left_leg` → `left_front_wheel`.
     - 연결 누락 시, `<joint>`의 `parent`/`child` 확인.
6. **문제 해결**
   - Xacro 파일 오류(예: 매크로 파라미터 누락, 상수 오타):
     - 변환 테스트:

       ```
       ros2 run xacro xacro ~/kuLimo/colcon_ws/src/ku_description/urdf/advanced_myfirst.xacro
       ```

     - 에러 메시지 확인.
   - RViz2에서 모델 안 보일 경우:
     - `robot_description` 토픽 확인:

       ```
       ros2 topic echo /robot_description
       ```

     - 파일 경로 확인:

       ```
       ls ~/kuLimo/colcon_ws/install/ku_description/share/ku_description/urdf
       ```

   - TF 트리 오류:
     - `rqt_tf_tree`에서 연결 확인.
     - `joint_state_publisher_gui` 실행 확인:

       ```
       ros2 node list | grep joint_state_publisher_gui
       ```

12-6. Limo ROS2의 `limo_description` 패키지 분석

12-6-1 수업 개요

참고 자료:

- ROS URDF 위키: http://wiki.ros.org/urdf
- ROS Xacro 위키: http://wiki.ros.org/xacro
- Limo ROS2 저장소: https://github.com/agilexrobotics/limo_ros2
- 실습 참고 저장소: https://github.com/freshmea/kuLimo/tree/main/colcon_ws/src
- Limo ROS2 문서: https://github.com/agilexrobotics/limo_ros2_doc

---

12-6-2 이론: `limo_description` 패키지와 Limo 로봇

Limo ROS2와 `limo_description` 패키지

- **AgileX Limo**: 교육 및 연구용 소형 모바일 로봇으로, ROS 2 Humble(Ubuntu 22.04)와 호환되며, Intel NCU i7 또는 NVIDIA Orin Nano 프로세서를 탑재.
- **특징**:
  - **다중 조향 모드**: 4륜 차동, Omni-Wheel, Tracked, Ackermann 조향 지원.
  - **센서**: EAI T-mini Pro LiDAR, Orbbec Dabai 깊이 카메라, 초음파 센서.
  - **응용**: SLAM, 자율 내비게이션, 장애물 회피, 비전 기반 작업.
- `limo_description` **패키지**:
  - `limo_ros2` 저장소의 서브패키지로, Limo 로봇의 URDF/Xacro 모델을 정의.
  - URDF/Xacro 파일은 로봇의 링크(바디, 바퀴, 센서), 조인트(고정, 연속), 센서(LiDAR, 카메라) 정보를 포함.
  - Gazebo 시뮬레이션과 RViz2 시각화를 지원하도록 `<inertial>`, `<collision>` 태그 포함.

`limo_description`의 Xacro 구조

- **파일 위치**: `limo_ros2/limo_description/urdf`.
- **주요 구성**:

    - **상수**: 치수(바디, 바퀴), 오프셋, 질량 등.
    - **매크로**: 반복적인 링크(바퀴, 센서)와 조인트를 모듈화.
    - **센서 정의**: LiDAR와 깊이 카메라를 `<link>`와 `<joint>`로 통합.
    - **다중 조향 지원**: 조향 모드별 조인트 설정(예: 연속 조인트로 바퀴 회전).
- **비교**: 제공된 `advanced_myfirst.xacro`는 Limo와 유사한 구조(바디, 좌우 다리, 바퀴)를 가지며, 매크로(`leg_macro`, `wheel_macro`)와 조건문(`<xacro:if>`)을 활용.

Xacro 고도화 요소

- **매크로**:
    - `leg_macro`, `wheel_macro`: 좌우 대칭 구조를 단일 호출로 처리.
    - `sensor_macro`(가정): LiDAR와 카메라를 모듈화.
- **상수**: `body_length`, `wheel_radius` 등으로 치수 관리.
- **조건문**: `<xacro:if>`로 시뮬레이션(Gazebo)과 시각화(RViz2) 환경 구분.
- **파일 포함**: `<xacro:include>`로 재질, 센서 설정 분리.

Limo의 다중 조향 모드

- **4륜 차동**: 독립적인 바퀴 회전으로 유연한 이동.
- **Omni-Wheel**: 전방향 이동 가능.
- **Tracked**: 험로 주행에 적합.
- **Ackermann**: 자동차와 유사한 조향.
- URDF/Xacro에서 조향 모드는 `<joint>`의 `type`(예: `continuous`)과 `axis`로 구현.

분석 워크플로우

- `limo_description`의 Xacro 파일 분석 → URDF 변환 → `robot_state_publisher`로 TF 발행 → RViz2 시각화.
- `check_urdf`로 구조 검증, `rqt_tf_tree`로 부모-자식 관계 확인.

12-6-3 실습: `limo_description` 패키지 분석 및 시각화 (1시간 15분)

실습 목표

- `limo_description` 패키지의 Xacro 파일 구조 분석.

- 제공된 `advanced_myfirst.xacro`를 Limo 모델로 확장하여 시각화.
- `check_urdf`와 `rqt_tf_tree`로 URDF와 TF 트리 검증.
- Limo의 다중 조향 모드와 센서 통합 이해.

단계

1. **환경 설정 및 패키지 다운로드**
   - 워크스페이스 생성 및 `limo_description` 패키지 클론:

     ```
     mkdir -p ~/kuLimo/colcon_ws/src
     cd ~/kuLimo/colcon_ws/src
     git clone https://github.com/agilexrobotics/limo_ros2.git
     ```

   - Xacro 및 관련 패키지 설치:

     ```
     sudo apt install ros-humble-xacro ros-humble-joint-state-publisher
     -gui ros-humble-robot-state-publisher
     ```

   - limo_car 패키지안에 폴더 생성 → src, log, worlds
2. **Xacro 파일 분석 및 작성**
   - `limo_description`의 Xacro 파일은 `limo_ros2/limo_description/urdf`에 위치.
   - 내용:

     ```
     <?xml version="1.0"?>
     <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="limo">
      <!-- 상수 정의 -->
      <xacro:property name="body_length" value="0.6"/>
      <xacro:property name="body_radius" value="0.2"/>
      <xacro:property name="leg_length" value="0.6"/>
      <xacro:property name="leg_width" value="0.1"/>
      <xacro:property name="leg_height" value="0.2"/>
      <xacro:property name="wheel_length" value="0.1"/>
      <xacro:property name="wheel_radius" value="0.035"/>
      <xacro:property name="leg_offset_y" value="0.22"/>
      <xacro:property name="leg_offset_z" value="0.25"/>
      <xacro:property name="wheel_offset_z" value="-0.65"/>
      <xacro:property name="lidar_height" value="0.1"/>
     ```

```
  <xacro:property name="camera_offset_z" value="0.3"/>

  <!-- 재질 정의 -->
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>
  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>
  <material name="black">
    <color rgba="0 0 0 1"/>
  </material>

  <!-- 매크로 정의 -->
  <xacro:macro name="default_inertial" params="mass *shape">
    <collision>
      <xacro:insert_block name="shape"/>
    </collision>
    <inertial>
      <mass value="${mass}"/>
      <!-- 단순화된 관성 행렬: 교육용으로 작은 값 사용 -->
      <origin xyz="0 0 0"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" iz
z="0.001"/>
    </inertial>
  </xacro:macro>

  <xacro:macro name="leg_macro" params="side">
    <link name="${side}_leg">
      <visual>
        <geometry>
          <box size="${leg_length} ${leg_width} ${leg_height}"/>
        </geometry>
        <origin rpy="0 1.57 0" xyz="0 0 -0.3"/>
        <material name="white"/>
      </visual>
      <xacro:default_inertial mass="1">
        <geometry>
```

```
        <box size="${leg_length} ${leg_width} ${leg_height}"/>
       </geometry>
     </xacro:default_inertial>
   </link>
   <joint name="base_to_${side}_leg" type="fixed">
     <parent link="base_link"/>
     <child link="${side}_leg"/>
     <xacro:property name="y_offset" value="${-leg_offset_y if side
== 'right' else leg_offset_y}"/>
     <origin xyz="0 ${y_offset} ${leg_offset_z}"/>
   </joint>
 </xacro:macro>

 <xacro:macro name="wheel_macro" params="side">
   <link name="${side}_front_wheel">
     <visual>
       <geometry>
         <cylinder length="${wheel_length}" radius="${wheel_radiu
s}"/>
       </geometry>
       <origin rpy="1.57 0 0" xyz="0 0 0"/>
       <material name="black"/>
     </visual>
     <xacro:default_inertial mass="1">
       <geometry>
         <cylinder length="${wheel_length}" radius="${wheel_radiu
s}"/>
       </geometry>
     </xacro:default_inertial>
   </link>
   <joint name="${side}_front_wheel_joint" type="continuous">
     <axis xyz="0 1 0"/>
     <parent link="${side}_leg"/>
     <child link="${side}_front_wheel"/>
     <origin xyz="0 0 ${wheel_offset_z}"/>
   </joint>
 </xacro:macro>
```

```
  <xacro:macro name="sensor_macro" params="type name offset_
z">
    <link name="${name}">
      <visual>
        <geometry>
          <xacro:if value="${type == 'lidar'}">
            <cylinder length="0.05" radius="0.03"/>
          </xacro:if>
          <xacro:if value="${type == 'camera'}">
            <box size="0.05 0.05 0.03"/>
          </xacro:if>
        </geometry>
        <material name="black"/>
      </visual>
      <xacro:default_inertial mass="0.1">
        <geometry>
          <xacro:if value="${type == 'lidar'}">
            <cylinder length="0.05" radius="0.03"/>
          </xacro:if>
          <xacro:if value="${type == 'camera'}">
            <box size="0.05 0.05 0.03"/>
          </xacro:if>
        </geometry>
      </xacro:default_inertial>
    </link>
    <joint name="base_to_${name}" type="fixed">
      <parent link="base_link"/>
      <child link="${name}"/>
      <origin xyz="0 0 ${offset_z}"/>
    </joint>
  </xacro:macro>

  <!-- 링크 및 조인트 정의 -->
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="${body_length}" radius="${body_radius}"/>
      </geometry>
```

```
    <material name="blue"/>
   </visual>
   <xacro:default_inertial mass="10">
    <geometry>
     <cylinder length="${body_length}" radius="${body_radius}"/>
    </geometry>
   </xacro:default_inertial>
  </link>

  <!-- Gazebo 속성 추가 -->
  <gazebo reference="base_link">
   <material>Gazebo/Blue</material>
   <mu1>0.8</mu1>
   <mu2>0.6</mu2>
  </gazebo>

  <!-- 좌우 다리 및 바퀴 추가 -->
  <xacro:leg_macro side="right"/>
  <xacro:leg_macro side="left"/>
  <gazebo reference="right_leg">
   <material>Gazebo/White</material>
   <mu1>0.7</mu1>
   <mu2>0.5</mu2>
  </gazebo>
  <gazebo reference="left_leg">
   <material>Gazebo/White</material>
   <mu1>0.7</mu1>
   <mu2>0.5</mu2>
  </gazebo>

  <xacro:wheel_macro side="right"/>
  <xacro:wheel_macro side="left"/>
  <gazebo reference="right_front_wheel">
   <material>Gazebo/Black</material>
   <mu1>1.0</mu1>
   <mu2>0.9</mu2>
  </gazebo>
  <gazebo reference="left_front_wheel">
```

```
    <material>Gazebo/Black</material>
    <mu1>1.0</mu1>
    <mu2>0.9</mu2>
  </gazebo>

  <!-- 센서 추가 -->
  <xacro:sensor_macro type="lidar" name="lidar" offset_z="${lidar_
height}"/>
  <gazebo reference="lidar">
    <material>Gazebo/Black</material>
    <sensor type="ray" name="lidar_sensor">
      <ray>
        <range>
          <min>0.1</min>
          <max>10.0</max>
        </range>
      </ray>
      <plugin name="lidar_plugin" filename="libgazebo_ros_laser.s
o">
        <ros>
          <namespace>/lidar</namespace>
          <remapping>~/out:=scan</remapping>
        </ros>
      </plugin>
    </sensor>
  </gazebo>

  <xacro:sensor_macro type="camera" name="camera" offset_z
="${camera_offset_z}"/>
  <gazebo reference="camera">
    <material>Gazebo/Black</material>
    <sensor type="camera" name="camera_sensor">
      <camera>
        <image>
          <width>640</width>
          <height>480</height>
        </image>
      </camera>
```

```
    <plugin name="camera_plugin" filename="libgazebo_ros_camera.so">
      <ros>
        <namespace>/camera</namespace>
        <remapping>~/out:=image_raw</remapping>
      </ros>
    </plugin>
  </sensor>
 </gazebo>
</robot>
```

12-7. Gazebo 및 URDF 통합 소개

12-7-1 수업 개요

참고 자료:

- ROS URDF 위키: http://wiki.ros.org/urdf
- Gazebo URDF 튜토리얼: http://classic.gazebosim.org/tutorials?tut=ros_urdf
- ROS 2 Gazebo 통합: http://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Using-URDF-with-Gazebo.html
- Limo ROS2 저장소: https://github.com/agilexrobotics/limo_ros2
- 실습 참고 저장소: https://github.com/freshmea/kuLimo/tree/main/colcon_ws/src

---

12-7-2 이론: Gazebo 및 URDF 통합

Gazebo 개요

- **Gazebo**: 물리 기반 오픈소스 시뮬레이터로, ROS 2와 통합되어 로봇의 동역학과 환경 상호작용을 시뮬레이션.
- **주요 기능**:
  - 물리 엔진(ODE, Bullet 등)을 사용한 충돌, 마찰, 중력 시뮬레이션.

    - 센서(LiDAR, 카메라)와 로봇 동작 시뮬레이션.
    - ROS 2와의 통합: `ros_gz` 패키지로 메시지 교환 및 제어.
- **ROS 2 Humble에서의 Gazebo**:
    - `ros-humble-gazebo-ros-pkgs`로 설치.
    - URDF/Xacro 파일을 활용하여 로봇 모델 로드.
- **응용**: SLAM, 내비게이션, 로봇 제어 테스트.

URDF와 SDF의 차이

- **URDF (Unified Robot Description Format)**:
    - ROS에서 로봇의 기하학적 구조(링크, 조인트)와 시각적 속성(색상, 재질)을 정의.
    - `<link>`: 바디, 바퀴 등.
    - `<joint>`: 고정, 연속, 회전 등.
    - `<visual>`: RViz2 시각화.
    - `<collision>`: 충돌 속성(선택적).
    - 제한: 물리적 속성(마찰, 관성)은 제한적.
- **SDF (Simulation Description Format)**:
    - Gazebo 전용 형식으로, 로봇과 환경(조명, 중력, 마찰)을 정의.
    - URDF보다 풍부한 물리 속성 지원: 마찰 계수, 댐핑, 관성 텐서.
    - `<model>`: 로봇 모델.
    - `<link>`: URDF와 유사하지만 추가 속성(마찰, 질량 중심) 포함.
    - `<plugin>`: Gazebo 플러그인(예: 센서, 제어) 추가.
- **차이점**:
    - **목적**: URDF는 로봇 설명, SDF는 시뮬레이션 환경.
    - **속성**: SDF는 마찰, 댐핑, 센서 노이즈 등 시뮬레이션 전용 속성 포함.
    - **호환성**: URDF는 ROS와 Gazebo에서 사용, SDF는 Gazebo 전용.

Gazebo의 URDF-to-SDF 자동 변환

- Gazebo는 URDF를 직접 로드 가능하며, 내부적으로 SDF로 변환.
- **변환 과정**:

1. URDF 파일 로드(`robot_description` 토픽).
2. `gz sdf` 명령어로 SDF 생성:

```
gz sdf -p physics.urdf >physics.sdf
```

3. Gazebo가 SDF를 사용하여 시뮬레이션 실행.

- **제한**:
  - URDF만으로 마찰, 댐핑 등 부족 → `<gazebo>` 태그로 보완.
  - 복잡한 환경은 SDF로 직접 작성 권장.

`<gazebo>` 태그

- URDF/Xacro에 추가하여 Gazebo의 시뮬레이션 속성을 정의.
- **구문**:

```
<gazebo reference="link_name">
  <!-- 시뮬레이션 속성 -->
</gazebo>
```

- **주요 속성**:
  - **마찰**:

```
<gazebo reference="base_link">
  <mu1>0.8</mu1> <!-- 정지 마찰 계수 -->
  <mu2>0.6</mu2> <!-- 동적 마찰 계수 -->
</gazebo>
```

  - **관성**:

```
<gazebo reference="base_link">
  <inertial>
    <mass>10.0</mass>
    <inertia>
      <ixx>0.001</ixx>
      <iyy>0.001</iyy>
      <izz>0.001</izz>
```

```
    </inertia>
  </inertial>
</gazebo>
```

- **재질**:

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

- **센서**:

```
<gazebo reference="lidar">
  <sensor type="ray" name="lidar_sensor">
    <ray>
      <range>
        <min>0.1</min>
        <max>10.0</max>
      </range>
    </ray>
    <plugin name="lidar_plugin" filename="libgazebo_ros_laser.so"/
>
  </sensor>
</gazebo>
```

- **역할**: URDF의 `<visual>`과 `<collision>`을 보완하여 Gazebo에서 정확한 물리 시뮬레이션 가능.

Limo 모델과의 연계

- `limo_description`의 Xacro 파일은 `<gazebo>` 태그를 포함하여 Gazebo 시뮬레이션을 지원.
- 제공된 `limo_model.xacro`를 확장하여 마찰, 관성, 센서 속성을 추가.
- 다중 조향 모드(4륜 차동 등)는 `<joint>`와 `<gazebo>` 태그로 구현.

12-7-3 실습: Gazebo에서 Limo 모델 시뮬레이션

실습 목표

- 가제보 설정
- `limo_description`의 Xacro 파일에 `<gazebo>` 태그 추가.
- Gazebo에서 Limo 로봇 모델 시뮬레이션.
- `check_urdf`와 `rqt_tf_tree`로 URDF와 TF 트리 검증.

단계

1. **환경 설정 및 패키지 설치**
   - Gazebo 및 ROS 2 통합 패키지 설치:

     ```
     sudo apt install ros-humble-gazebo-ros-pkgs ros-humble-ros-gz ro
     s-humble-gazebo-*
     ```

   - 가제보 환경 설정(반드시 ~/.bashrc 에 넣어야 함.

     ```
     source /usr/share/gazebo/setup.bash
     export SVGA_VGPU10=0
     ```

2. **Xacro 파일 수정**
   - `limo_model.xacro`에 `<gazebo>` 태그 추가:

     ```
     nano ~/kuLimo/colcon_ws/src/limo_ros2/limo_description/urdf/limo
     _model.xacro
     ```

3. **패키지 설정 확인**
   - `limo_description`의 `package.xml`에 `gazebo_ros`, `xacro`, `joint_state_publisher_gui`, `robot_state_publisher` 의존성 확인.
   - `setup.py`는 `urdf` 디렉토리의 `.xacro` 파일을 포함.
4. **패키지 빌드 및 실행**
   - Gazebo 실행:

     ```
     ros2 launch limo_description gazebo_models_diff.launch.py
     ```

   - Gazebo에서:
     - Limo 로봇 모델(파란색 원통, 흰색 다리, 검은색 바퀴, LiDAR, 카메라) 확인.

- `JointStatePublisherGui`로 바퀴 회전 테스트.
- ROS 2 토픽 확인:

```
ros2 topic echo /scan  # LiDAR 데이터
ros2 topic echo /limo/depth_camera_link/image_raw  # 카메라 데이터
```

12-8. URDF에 물리적 속성 추가

12-8-1 수업 개요

참고 자료:

- ROS URDF 위키: http://wiki.ros.org/urdf
- Gazebo URDF 튜토리얼: http://classic.gazebosim.org/tutorials?tut=ros_urdf
- ROS 2 Gazebo 통합: http://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Using-URDF-with-Gazebo.html
- Limo ROS2 저장소: https://github.com/agilexrobotics/limo_ros2
- 실습 참고 저장소: https://github.com/freshmea/kuLimo/tree/main/colcon_ws/src

---

12-8-2 이론: URDF에 물리적 속성 추가

`<inertial>` 속성 정의

- **역할**: Gazebo에서 로봇의 동역학(운동, 회전)을 시뮬레이션하기 위해 링크의 질량, 질량 중심, 관성 행렬을 정의.
- **구문**:

```
<inertial>
  <mass value="질량(kg)"/>
  <origin xyz="x y z" rpy="roll pitch yaw"/> <!-- 질량 중심 -->
  <inertia ixx="Ixx" ixy="Ixy" ixz="Ixz" iyy="Iyy" iyz="Iyz" izz="Izz"/>
</inertial>
```

- **속성**:
  - `<mass>`: 링크의 질량(kg).
  - `<origin>`: 질량 중심의 위치(xyz, 미터)와 방향(rpy, 라디안). 생략 시 링크 중심(0,0,0).
  - `<inertia>`: 관성 텐서(kg·m²). 대칭 행렬로 6개 요소(ixx, ixy, ixz, iyy, iyz, izz) 정의.
- **예시** (원통 링크):

```
<inertial>
  <mass value="10.0"/>
  <origin xyz="0 0 0"/>
  <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.
001"/>
</inertial>
```

- **중요성**: 관성 행렬은 회전 동역학에 영향을 미치며, 부정확한 값은 비현실적 동작(예: 과도한 흔들림)을 초래.

`<collision>` 지오메트리 추가

- **역할**: Gazebo에서 링크 간 충돌을 계산하기 위한 지오메트리 정의.
- **구문**:

```
<collision>
  <origin xyz="x y z" rpy="roll pitch yaw"/>
  <geometry>
    <cylinder length="길이" radius="반지름"/>
    <!-- 또는 <box size="x y z"/>, <sphere radius="r"/>, <mesh filename
="..."/> -->
  </geometry>
</collision>
```

- **속성**:
  - `<origin>`: 충돌 지오메트리의 위치와 방향.
  - `<geometry>`: `<visual>`과 유사하지만 단순화된 형태 권장(계산 효율성).
- **예시**:

```
<collision>
  <origin xyz="0 0 0"/>
  <geometry>
    <cylinder length="0.6" radius="0.2"/>
  </geometry>
</collision>
```

- **중요성**: `<collision>`이 없으면 Gazebo에서 충돌 무시 → 비현실적 동작(예: 객체 통과).

Gazebo 전용 속성 (`<gazebo>` 태그)

- **역할**: URDF의 물리적 속성을 보완하여 Gazebo에서 마찰, 재질, 센서 등을 정의.
- **구문**:

```
<gazebo reference="link_name">
  <!-- 마찰, 재질, 센서 등 -->
</gazebo>
```

- **주요 속성**:
  - **마찰**:

```
<gazebo reference="base_link">
  <mu1>0.8</mu1> <!-- 정지 마찰 계수 -->
  <mu2>0.6</mu2> <!-- 동적 마찰 계수 -->
</gazebo>
```

  - **재질**:

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

  - **센서**:

```
<gazebo reference="lidar">
  <sensor type="ray" name="lidar_sensor">
    <ray>
```

```
      <range>
        <min>0.1</min>
        <max>10.0</max>
      </range>
    </ray>
    <plugin name="lidar_plugin" filename="libgazebo_ros_laser.so"/
>
  </sensor>
</gazebo>
```

- **중요성**: `<gazebo>` 태그는 URDF-to-SDF 변환 시 SDF의 물리 속성으로 반영.

URDF-to-SDF 변환

- Gazebo는 URDF를 SDF로 변환하여 시뮬레이션.
- **명령어**:

```
gz sdf -p path/to/my_robot.urdf > output.sdf
```

- **변환 과정**:
  1. URDF의 `<link>`, `<joint>`를 SDF의 `<model>`, `<link>`로 매핑.
  2. `<inertial>`, `<collision>`을 SDF의 물리 속성으로 변환.
  3. `<gazebo>` 태그를 SDF의 추가 속성(마찰, 플러그인)으로 반영.
- **검증**:
  - 변환된 SDF 파일 확인: `cat output.sdf`.
  - Gazebo 실행으로 동작 확인.

Limo 모델과의 연계

- `limo_description`의 Xacro 파일은 `<inertial>`, `<collision>`, `<gazebo>` 태그를 포함.
- 제공된 `limo_model.xacro`를 확장하여 물리적 속성 추가.
- Limo의 다중 조향 모드(4륜 차동 등)는 `<joint>`와 `<gazebo>`로 구현.

12-8-3 실습: Limo 모델에 물리적 속성 추가 및 시뮬레이션

실습 목표

- `limo_model.xacro`에 `<inertial>`, `<collision>`, `<gazebo>` 태그 추가.
- Gazebo에서 Limo 로봇 모델 시뮬레이션.
- URDF-to-SDF 변환 테스트.
- `check_urdf`와 `rqt_tf_tree`로 검증.

단계

1. **패키지 설정 확인**
   - `limo_description`의 `package.xml`에 `gazebo_ros`, `xacro`, `joint_state_publisher_gui`, `robot_state_publisher` 의존성 확인.
   - `setup.py`는 `urdf` 디렉토리의 `.xacro` 파일 포함.
2. **URDF-to-SDF 변환 테스트**
   - Xacro를 URDF로 변환:

     ```
     cd ~/kuLimo/colcon_ws/src/limo_ros2/limo_description/urdf
     ros2 run xacro xacro limo_four_diff.xacro > /tmp/limo_model.urdf
     ```

   - URDF-to-SDF 변환:

     ```
     gz sdf -p /tmp/limo_model.urdf > /tmp/limo_model.sdf
     ```

   - SDF 파일 확인:

     ```
     cat /tmp/limo_model.sdf
     ```

   - 예상 출력: `<model name="limo">`로 시작, `<link>`, `<joint>`, 마찰 속성 포함.
3. 만들어진 sdf 활용 (Gazebo에서 파일로드)
   - world 파일에 inclue :
     ex)

```
<include>
  <uri>model://turtlebot3_autorace_2020/traffic_stop</uri>
  <pose> -1.35 1.04 0.125 0 -0 -1.57</pose>
</include>
```

- spawn_entity.py 를 이용한 모델 로드(gazebo_ros 패키지)
- gazebo gui 를 이용한 모델 로드

12-9 ROS 2 Humble로 Gazebo

12-9-1. Gazebo 시작 및 URDF 모델 스폰

ROS 2 Humble에서 Gazebo를 실행하고 URDF 모델을 로드하려면 런치 파일을 작성하여 Gazebo 서버를 시작하고 `spawn_entity` 노드를 호출합니다.

런치 파일 작성

다음은 빈 Gazebo 월드와 함께 URDF 모델을 스폰하는 예제 런치 파일입니다. 이 파일은 `your_package/launch/spawn_robot.launch.py`에 저장됩니다.

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionS
ource
from launch_ros.actions import Node

def generate_launch_description():
    # 패키지 경로
    gazebo_ros_pkg = get_package_share_directory('gazebo_ros')
    your_package = get_package_share_directory('your_package')

    # URDF 파일 경로
    urdf_file = os.path.join(your_package, 'urdf', 'my_robot.urdf')

    # robot_description 파라미터에 URDF 로드
    with open(urdf_file, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
```

```
        # Gazebo 서버 실행
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                os.path.join(gazebo_ros_pkg, 'launch', 'gzserver.launch.py')
            ),
            launch_arguments={'world': os.path.join(your_package, 'worlds', 'e
mpty.world')}.items()
        ),

        # Gazebo 클라이언트 실행 (GUI)
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                os.path.join(gazebo_ros_pkg, 'launch', 'gzclient.launch.py')
            )
        ),

        # robot_description 파라미터 설정
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'robot_description': robot_desc}]
        ),

        # spawn_entity 노드로 로봇 스폰
        Node(
            package='gazebo_ros',
            executable='spawn_entity.py',
            name='spawn_entity',
            output='screen',
            arguments=['-topic', '/robot_description', '-entity', 'my_robot', '-x',
'0', '-y', '0', '-z', '0']
        )
    ])
```

- Gazebo GUI에서 로봇이 올바르게 렌더링되는지 확인.

- ROS 2 토픽 확인:

```
ros2 topic list
```

  - `/robot_description`, `/tf`, `/tf_static` 등이 표시되어야 함.
- `rviz2`로 로봇 시각화 (선택):

```
ros2 run rviz2 rviz2
```

  - `RobotModel` 디스플레이 추가, `Topic`을 `/robot_description`으로 설정.

12-9-2. 흔한 문제 해결

(1) 관성 태그 누락

- **증상**: 로봇이 Gazebo에 로드되지 않거나 비정상적으로 움직임.
- **원인**: URDF의 `<inertial>` 태그가 누락되었거나 잘못된 값.
- **해결**:
  - `<inertial>` 태그에 `<mass>`와 `<inertia>` 요소 추가.
  - 예: 위 URDF의 `<inertial>` 참조.
  - 관성 행렬 값이 양수이고 적절한지 확인.

(2) 모델이 보이지 않음

- **원인**: `<visual>` 또는 `<collision>` 태그 누락, Gazebo 플러그인 문제.
- **해결**:
  - `<visual>`과 `<collision>` 태그가 있는지 확인.
  - Gazebo 로그 확인:

```
gzserver --verbose
```

(3) spawn_entity 실패

- **원인**: `robot_description` 파라미터가 설정되지 않음, URDF 파싱 오류.
- **해결**:

- `ros2 param get /robot_description`으로 파라미터 확인.
- URDF 파일의 XML 문법 점검:

```
check_urdf /path/to/my_robot.urdf
```

(4) Gazebo가 실행되지 않음

- **원인**: 패키지 누락, 그래픽 드라이버 문제.
- **해결**:
  - `ros-humble-gazebo-ros-pkgs` 재설치.
  - 그래픽 드라이버 확인:

```
glxinfo | grep "OpenGL version"
```

- **다른 위치에 로봇 스폰**: 런치 파일의 `x`, `y`, `z` 인자 변경.
- **복잡한 URDF**: 바퀴 달린 로봇 URDF 작성 후 스폰.
- **월드 파일 추가**: `empty.world` 대신 커스텀 월드 파일 작성.
- ROS 2 Humble 문서: http://docs.ros.org/en/humble/
- Gazebo Classic 문서: http://gazebosim.org/
- `ros_gz` GitHub: https://github.com/gazebosim/ros_gz

12-10 Gazebo에서 커스텀 월드 파일 작성하기

이 교육 자료는 Gazebo 시뮬레이션 환경에서 커스텀 월드 파일을 작성하는 방법을 안내합니다. `<include>` 태그를 사용해 기본 모델(지면, 조명)을 추가하고, SDF 형식으로 정적 객체(벽, 테이블, 박스 등)를 정의하며, 작성한 월드 파일을 실행하는 과정을 다룹니다.

학습 목표

- `<include>` 태그를 사용해 Gazebo 모델 추가
- SDF 형식으로 정적 객체 정의
- 커스텀 월드 파일 작성 및 실행

1. Gazebo 월드 파일의 기본 구조

Gazebo 월드 파일은 XML 기반의 SDF(Simulation Description Format)로 작성됩니다. 월드 파일은 시뮬레이션 환경의 구성 요소(모델, 조명, 물리 설정 등)를 정의합니다.

기본 구조는 다음과 같습니다:

```
<?xml version="1.0" ?>
<sdf version="1.7">
  <world name="my_world">
    <!-- 월드 구성 요소: 지면, 조명, 모델 등 -->
  </world>
</sdf>
```

2. `<include>` 태그로 모델 추가

`<include>` 태그를 사용하면 Gazebo에 내장된 모델(예: `ground_plane`, `sun`)을 쉽게 추가할 수 있습니다.

예제: 지면과 조명 추가

```
<include>
  <uri>model://ground_plane</uri>
</include>
<include>
  <uri>model://sun</uri>
</include>
```

- `ground_plane`: 기본 지면 모델로, 시뮬레이션의 바닥 역할을 합니다.
- `sun`: 기본 조명 모델로, 월드에 빛을 제공합니다.

3. SDF로 정적 객체 추가

정적 객체(예: 벽, 테이블, 박스)는 `<model>` 태그를 사용해 정의합니다. 정적 객체는 `static` 속성을 `true`로 설정하여 물리적 움직임을 방지합니다.

예제: 정적 박스 추가

다음은 크기 1x1×1인 정적 박스를 원점에 배치하는 코드입니다:

```
<model name="static_box">
  <static>true</static>
  <pose>0 0 0.5 0 0 0</pose>
  <link name="link">
    <collision name="collision">
      <geometry>
        <box>
          <size>1 1 1</size>
        </box>
      </geometry>
    </collision>
    <visual name="visual">
      <geometry>
        <box>
          <size>1 1 1</size>
        </box>
      </geometry>
      <material>
        <script>
          <uri>file://media/materials/scripts/gazebo.material</uri>
          <name>Gazebo/Grey</name>
        </script>
      </material>
    </visual>
  </link>
</model>
```

- `<pose>`: 객체의 위치와 회전(x, y, z, roll, pitch, yaw)을 지정합니다.
- `<static>true</static>`: 객체를 정적으로 만들어 물리적 상호작용을 방지합니다.
- `<geometry>`: 객체의 형상을 정의합니다(여기서는 박스).
- `<material>`: 객체의 색상 또는 텍스처를 지정합니다.

4. 완전한 월드 파일 예제

다음은 지면, 조명, 정적 박스를 포함한 완전한 월드 파일 예제입니다:

```xml
<?xml version="1.0" ?>
<sdf version="1.7">
  <world name="my_world">
    <!-- 지면 추가 -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <!-- 조명 추가 -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- 정적 박스 추가 -->
    <model name="static_box">
      <static>true</static>
      <pose>0 0 0.5 0 0 0</pose>
      <link name="link">
        <collision name="collision">
          <geometry>
            <box>
              <size>1 1 1</size>
            </box>
          </geometry>
        </collision>
        <visual name="visual">
          <geometry>
            <box>
              <size>1 1 1</size>
            </box>
          </geometry>
          <material>
            <script>
              <uri>file://media/materials/scripts/gazebo.material</uri>
              <name>Gazebo/Grey</name>
            </script>
          </material>
        </visual>
      </link>
    </model>
```

```
  </world>
</sdf>
```

파일 저장

위 코드를 `my_world.world` 파일로 저장합니다(예: `~/gazebo_worlds/my_world.world`).

5. 월드 파일 실행

작성한 월드 파일을 Gazebo에서 실행하려면 다음 명령어를 사용합니다:

```
ros2 launch gazebo_ros gazebo.launch.py world:=~/gazebo_worlds/my_wo
rld.world
```

- `gazebo.launch.py`: Gazebo를 ROS 2와 함께 실행하는 런치 파일입니다.
- `world:=<파일 경로>`: 실행할 월드 파일의 경로를 지정합니다.

12-11 터틀봇3 시뮬레이션 추가하기

이 섹션에서는 터틀봇3 로봇을 Gazebo 시뮬레이션 환경에 추가하고 제어하는 방법을 설명합니다. 터틀봇3의 URDF 모델을 사용하여 시뮬레이션을 구성하고, ROS 2 명령을 통해 로봇을 제어하는 과정을 다룹니다.

학습 목표

- 터틀봇3 URDF 모델을 Gazebo 시뮬레이션에 통합하기
- ROS 2 명령어를 사용하여 시뮬레이션된 터틀봇3 제어하기
- 센서 데이터 수집 및 시각화 방법 이해하기

12-11-1 패키지 설치 및 환경 설정

```
cd ~/kuLimo/colcon_ws/src
git clone -b humble https://github.com/ROBOTIS-GIT/turtlebot3_simulations.gi
sudo apt install ros-humble-turtlebot3-msgs
```

```
sudo apt install ros-humble-turtlebot3-teleop

# 이 내용은 ~/.bashrc 에 추가
export TURTLEBOT3_MODEL=burger
```

기본 터틀봇3 simulation 실행

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
ros2 run turtlebot3_teleop teleop_keyboard
```

터틀봇3의 gazebo 환경에 limo_ros2 모델 로드 하기

```
import os

from launch import LaunchDescription
from launch.actions import (
    DeclareLaunchArgument,
    ExecuteProcess,
    IncludeLaunchDescription,
)
from launch.conditions import IfCondition, UnlessCondition
from launch.launch_description_sources import PythonLaunchDescriptionSou
from launch.substitutions import Command, LaunchConfiguration, PythonExpr
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():

    # Constants for paths to different files and folders
    gazebo_models_path = "models"
    package_name = "limo_description"
    world_package_name = "turtlebot3_gazebo" # 이부분 변경
    robot_name_in_model = "limo_description"
    rviz_config_file_path = "rviz/urdf.rviz"
    urdf_file_path = "urdf/limo_four_diff.xacro"
```

```
world_file_path = "worlds/turtlebot3_world.world" # 이부분 변경

# Pose where we want to spawn the robot
spawn_x_val = "0.0"
spawn_y_val = "0.0"
spawn_z_val = "0.0"
spawn_yaw_val = "0.00"

############ You do not need to change anything below this line #####

# Set the path to different files and folders.
pkg_gazebo_ros = FindPackageShare(package="gazebo_ros").find("gazebo
pkg_share = FindPackageShare(package=package_name).find(package_na
world_pkg_share = FindPackageShare(package=world_package_name).find
    world_package_name
) # 이부분 변경
default_urdf_model_path = os.path.join(pkg_share, urdf_file_path)
default_rviz_config_path = os.path.join(pkg_share, rviz_config_file_path)
world_path = os.path.join(world_pkg_share, world_file_path) # 이부분 변경
gazebo_models_path = os.path.join(pkg_share, gazebo_models_path)
os.environ["GAZEBO_MODEL_PATH"] = gazebo_models_path

# Launch configuration variables specific to simulation
use_sim_time = LaunchConfiguration("use_sim_time", default="true")
gui = LaunchConfiguration("gui")
headless = LaunchConfiguration("headless")
namespace = LaunchConfiguration("namespace")
rviz_config_file = LaunchConfiguration("rviz_config_file")
urdf_model = LaunchConfiguration("urdf_model")
use_namespace = LaunchConfiguration("use_namespace")
use_robot_state_pub = LaunchConfiguration("use_robot_state_pub")
use_rviz = LaunchConfiguration("use_rviz")
use_simulator = LaunchConfiguration("use_simulator")
world = LaunchConfiguration("world")

# Declare the launch arguments
declare_use_sim_time_cmd = DeclareLaunchArgument(
    name="use_sim_time",
```

```
        default_value="True",
        description="Use simulation (Gazebo) clock if true",
    )

    declare_use_joint_state_publisher_cmd = DeclareLaunchArgument(
        name="gui",
        default_value="True",
        description="Flag to enable joint_state_publisher_gui",
    )

    declare_namespace_cmd = DeclareLaunchArgument(
        name="namespace", default_value="", description="Top-level namespac
    )

    declare_use_namespace_cmd = DeclareLaunchArgument(
        name="use_namespace",
        default_value="False",
        description="Whether to apply a namespace to the navigation stack",
    )

    declare_rviz_config_file_cmd = DeclareLaunchArgument(
        name="rviz_config_file",
        default_value=default_rviz_config_path,
        description="Full path to the RVIZ config file to use",
    )

    declare_simulator_cmd = DeclareLaunchArgument(
        name="headless",
        default_value="False",
        description="Whether to execute gzclient",
    )

    declare_urdf_model_path_cmd = DeclareLaunchArgument(
        name="urdf_model",
        default_value=default_urdf_model_path,
        description="Absolute path to robot urdf file",
    )
```

```
    declare_use_robot_state_pub_cmd = DeclareLaunchArgument(
        name="use_robot_state_pub",
        default_value="True",
        description="Whether to start the robot state publisher",
    )

    declare_use_rviz_cmd = DeclareLaunchArgument(
        name="use_rviz", default_value="True", description="Whether to start RV
    )

    declare_use_simulator_cmd = DeclareLaunchArgument(
        name="use_simulator",
        default_value="True",
        description="Whether to start the simulator",
    )

    declare_world_cmd = DeclareLaunchArgument(
        name="world",
        default_value=world_path,
        description="Full path to the world model file to load",
    )

    # Subscribe to the joint states of the robot, and publish the 3D pose of each
    start_robot_state_publisher_cmd = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        parameters=[
            {
                "robot_description": Command(["xacro ", urdf_model]),
                "use_sim_time": use_sim_time,
            }
        ],
    )

    # Publish the joint states of the robot
    start_joint_state_publisher_cmd = Node(
        package="joint_state_publisher",
        executable="joint_state_publisher",
```

```
    name="joint_state_publisher",
    condition=UnlessCondition(gui),
    parameters=[{"use_sim_time": use_sim_time}],
)

start_joint_state_publisher_gui_node = Node(
    condition=IfCondition(gui),
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
    name="joint_state_publisher_gui",
    parameters=[{"use_sim_time": use_sim_time}],
)
# start_dummy_sensors=Node(
#   package='dummy_sensors',
#   node_executable='dummy_joint_states',
#   output='screen')

# Launch RViz
start_rviz_cmd = Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="screen",
    arguments=["-d", rviz_config_file],
)

# Start Gazebo server
start_gazebo_server_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        os.path.join(pkg_gazebo_ros, "launch", "gzserver.launch.py")
    ),
    condition=IfCondition(use_simulator),
    launch_arguments={"world": world}.items(),
)

# Start Gazebo client
start_gazebo_client_cmd = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
```

```
        os.path.join(pkg_gazebo_ros, "launch", "gzclient.launch.py")
    ),
    condition=IfCondition(PythonExpression([use_simulator, " and not ", head
)

# Launch the robot
spawn_entity_cmd = Node(
    package="gazebo_ros",
    executable="spawn_entity.py",
    arguments=[
        "-entity",
        robot_name_in_model,
        "-topic",
        "robot_description",
        "-x",
        spawn_x_val,
        "-y",
        spawn_y_val,
        "-z",
        spawn_z_val,
        "-Y",
        spawn_yaw_val,
    ],
    output="screen",
)

# Create the launch description and populate
ld = LaunchDescription()

# Declare the launch options
ld.add_action(declare_use_sim_time_cmd)
ld.add_action(declare_use_joint_state_publisher_cmd)
ld.add_action(declare_namespace_cmd)
ld.add_action(declare_use_namespace_cmd)
ld.add_action(declare_rviz_config_file_cmd)
ld.add_action(declare_simulator_cmd)
ld.add_action(declare_urdf_model_path_cmd)
ld.add_action(declare_use_robot_state_pub_cmd)
```

```
    ld.add_action(declare_use_rviz_cmd)
    ld.add_action(declare_use_simulator_cmd)
    ld.add_action(declare_world_cmd)

    # Add any actions
    ld.add_action(start_gazebo_server_cmd)
    ld.add_action(start_gazebo_client_cmd)
    ld.add_action(spawn_entity_cmd)
    ld.add_action(start_robot_state_publisher_cmd)
    ld.add_action(start_joint_state_publisher_gui_node)
    # ld.add_action(start_dummy_sensors)
    ld.add_action(start_rviz_cmd)

    return ld
```

worldpath 를 따로 추가하여 turtlebot3_world.world 파일로 로드 한다.

만들어진 환경에 모델 추가하기

```
<?xml version="1.0"?>
<sdf version="1.6">
  <world name="default">

    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sun</uri>
    </include>

    <scene>
      <shadows>false</shadows>
    </scene>

    <gui fullscreen='0'>
      <camera name='user_camera'>
        <pose frame=''>0.319654 -0.235002 9.29441 0 1.5138 0.009599</pose>
```

```
    <view_controller>orbit</view_controller>
    <projection_type>perspective</projection_type>
  </camera>
</gui>

<physics type="ode">
  <real_time_update_rate>1000.0</real_time_update_rate>
  <max_step_size>0.001</max_step_size>
  <real_time_factor>1</real_time_factor>
  <ode>
    <solver>
      <type>quick</type>
      <iters>150</iters>
      <precon_iters>0</precon_iters>
      <sor>1.400000</sor>
      <use_dynamic_moi_rescaling>1</use_dynamic_moi_rescaling>
    </solver>
    <constraints>
      <cfm>0.00001</cfm>
      <erp>0.2</erp>
      <contact_max_correcting_vel>2000.000000</contact_max_correcting_v
      <contact_surface_layer>0.01000</contact_surface_layer>
    </constraints>
  </ode>
</physics>

<model name="turtlebot3_world">
  <static>1</static>
  <include>
    <uri>model://turtlebot3_world</uri>
  </include>
</model>

<include>
  <uri>model://turtlebot3_autorace_2020/traffic_stop</uri>
  <pose> -1.35 1.04 0.125 0 -0 -1.57</pose>
</include>
```

```
  </world>
</sdf>
```

<include> 태그를 추가하여 sdf 형식의 파일이나 폴더를 로드한다.