시뮬레이션 프로젝트 교안

Version	V1.0
최종수정일	2025.02.18
작성자	김루진

가제보(gazebo)가상환경 구축

가제보

- ROS 시뮤레이션 툴인 가제보를 이용하여 테스트 환경 구축하기
- 터틀봇3를 사용하여 실제 시뮬레이션 구축 및 실제 환경 테스트
- 가상의 공장에서 Pick and Place를 구현하는 시뮬레이션 구축하기

가제보 이해

가제보 신버전 정보

로봇 시뮬레이션은 모든 로봇 공학자의 도구 상자에서 필수적인 도구.
잘 설계된 시뮬레이터를 사용하면 현실적인 시나리오를 사용하여 알고리즘을 신속하게 테스트하고, 로봇을 설계하고, 회귀 테스트를 수행하고, AI 시스템을 훈련.

Gazebo Classic과 Ignition Gazebo (현재는 Gazebo로 통합)라는 두 가지 주요 버전

신규 버전 (Ignition Gazebo)
모듈화된 아키텍처.
여러 독립된 모듈(Ignition Physics, Ignition Rendering, Ignition Transport 등)로 구성, 특정 기능만 교체, 확장

물리 엔진

•**구버전 (Gazebo Classic)**:
•기본 물리 엔진은 ODE (Open Dynamics Engine)를 사용
•다른 엔진을 지원하기 위해 플러그인을 사용할 수 있었습니다. Bullet, Simbody, DART 등과도 통합

•**신규 버전 (Ignition Gazebo)**:
•다양한 물리 엔진과의 더 나은 통합을 제공하며, TPE (Trivial Physics Engine)와 같은 새로운 엔진 도입 성능과 유연성 향상

가제보 이해

1. 셋업
   - 설치하기 싸이트
   - Classic버전 설치
   - Gazebo : Tutorial : Ubuntu (gazebosim.org)

2. 최소 설치 사양
   - CPU: Intel 3세대 i3 또는 그 이상 (AMD도 무방)
   - GPU: Intel HD4000 또는 그 이상 (우분투에서 드라이버가 잘 잡히는.)
   - RAM: 4G (다램익선^^)
   - HDD: 50G 이상의 여유 공간

3. 설치하기

```
# Install Gazebo Classic 11
sudo apt install gazebo11 -y

# Install ROS2-Gazebo integration packages
sudo apt install ros-humble-gazebo-ros-pkgs -y

# Source ROS2 setup script
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc

# Install additional tools (optional but recommended)
sudo apt install python3-colcon-common-extensions -y
```

신버전 가제보

Ubuntu 22.04와 ROS2 Humble에 맞는 Gazebo 설치 방법
ROS2 Humble은 Gazebo 11과 호환되며, 이는 'gz'로 시작하는 새로운 Gazebo 버전

```
$sudo apt update
$sudo apt install -y ros-humble-gazebo-ros-pkgs
```

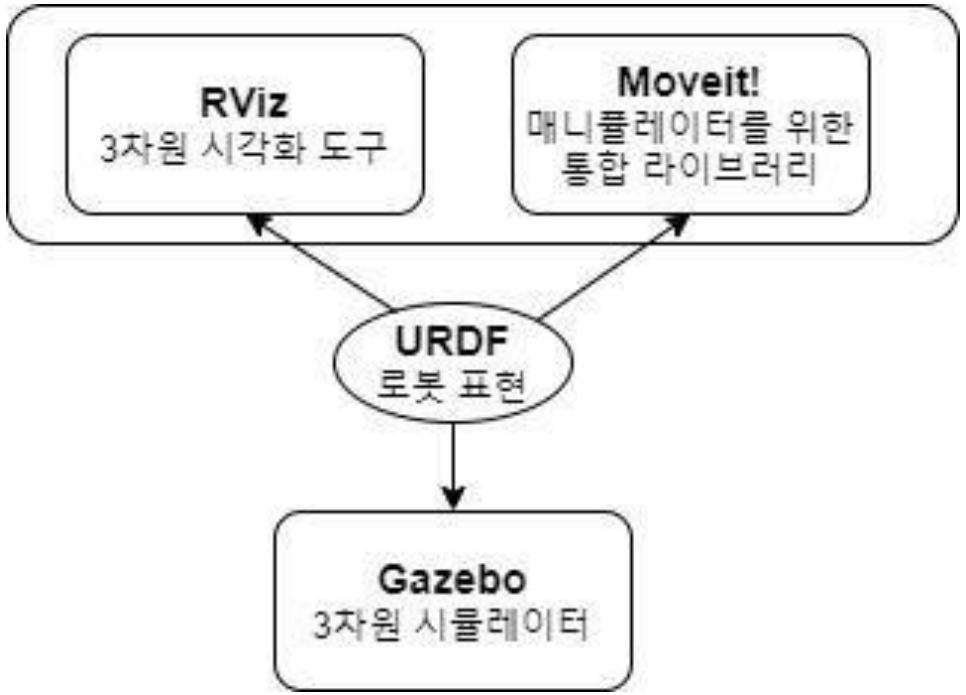
특정 월드로 Gazebo 실행

```
gz sim -v 4 /path/to/your/world/file.sdf
```

```python
import os
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    world_file = os.path.join(
        get_package_share_directory('your_package'),
        'worlds',
        'your_world.sdf'
    )

    return LaunchDescription([
        Node(
            package='gazebo_ros',
            executable='gazebo_sim',
            arguments=[world_file],
            output='screen'
        )
    ])
```

SDF(Simulation Description Format)

URDF는 RViz, Moveit! 그리고 Gazebo에서 모두 사용



Notes: This is the root SDF element.

<sdf> Element
Required: 1
Type:
Default:
Description: SDF base element.

version Attribute
Required: 1
Type: string
Default: 1.5
Description: Version number of the SDF format.

<world> Element
Required: *
Type:
Default:
Description: The world element encapsulates an entire world description including: models, scene, physics, joints, and plugins

<model> Element
Required: *
Type:
Default:
Description: The model element defines a complete robot or any other physical object.

<actor> Element
Required: *
Type:
Default:
Description: A special kind of model which can have a scripted motion. This includes both global waypoint type animations and skeleton animations.

<light> Element
Required: *
Type:
Default:
Description: The light element describes a light source.

가제보 이해

3. 가제보 다양한 기능

동역학 시뮬레이션

ODE, Bullet, Simbody 및 DART를 포함한 여러 고성능 물리 엔진에 액세스할 수 있습니다.

고급 3D 그래픽

OGRE를 활용하는 Gazebo는 고품질 조명, 그림자 및 질감을 포함한 환경의 사실적인 렌더링을 제공합니다.

센서와 소음

레이저 거리 측정기, 2D/3D 카메라, Kinect 스타일 센서, 접촉식 센서, 힘-토크 등에서 선택적으로 노이즈가 있는 센서 데이터를 생성합니다.

플러그인

로봇, 센서 및 환경 제어를 위한 맞춤형 플러그인을 개발합니다. 플러그인은 Gazebo의 API에 대한 직접 액세스를 제공합니다.

로봇 모델

PR2, Pioneer2 DX, iRobot Create, TurtleBot 등 많은 로봇이 제공됩니다. 또는 SDF를 사용하여 직접 구축하십시오.

TCP/IP 전송

원격 서버에서 시뮬레이션을 실행하고 Google Protobufs를 사용하여 소켓 기반 메시지 전달을 통해 Gazebo에 인터페이스합니다.

클라우드 시뮬레이션

CloudSim을 사용하여 Amazon, AWS 및 GzWeb에서 Gazebo를 실행하여 브라우저를 통해 시뮬레이션과 상호 작용할 수 있습니다.

명령줄 도구

광범위한 명령줄 도구는 시뮬레이션 내성 및 제어를 용이하게 합니다.

가제보 프로젝트 디렉토리 구조

가제보 로봇 모델을 spawn하고 ROS를 통해 제어하기위해 필요한 폴더 및 파일 구조

가제보 모델 데이터베이스 폴더 구조에 따른 포맷의 규칙을 따라야 한다.

- launch 폴더는 시뮬레이션을 시작하는 데 사용되는 launch 파일 저장
- worlds 폴더에는 가제보 환경을 정의하는 .world 파일 저장
- models 폴더에는 로봇이나 객체와 같은 개별 모델 파일 저장
- meshes 폴더는 3D 모델의 시각적 및 물리적 표현을 위한 파일 저장
- config 폴더에는 컨트롤러나 기타 설정 파일 저장
- scripts 폴더는 로봇 제어나 시뮬레이션 관련 스크립트 파일 저장
- plugins 폴더에는 가제보의 기능을 확장하는 사용자 정의 플러그인

```
my_gazebo_project/
│
├── CMakeLists.txt
├── package.xml
│
├── launch/
│   └── simulation.launch
│
├── worlds/
│   └── my_world.world
│
├── models/
│   ├── my_robot/
│   │   ├── model.config
│   │   └── model.sdf
│   └── my_object/
│       ├── model.config
│       └── model.sdf
│
├── meshes/
│   ├── visual/
│   │   └── robot_body.dae
│   └── collision/
│       └── robot_body_simplified.stl
│
├── config/
│   └── controller_config.yaml
│
├── scripts/
│   └── control_script.py
│
└── plugins/
    └── my_plugin.cpp
```

가제보 프로젝트 디렉토리 구조

가제보 GUI 인터페이스의 Insert Model을 통해 당신의 모델을 불러올 수 있다

mkdir -p ~/.gazebo/models/my_robot

```
<?xml version="1.0"?>
    <model>
     <name>My Robot</name>
     <version>1.0</version>
     <sdf version='1.4'>model.sdf</sdf>

     <author>
      <name>My Name</name>
      <email>me@my.email</email>
     </author>

     <description>
       My awesome robot.
     </description>
   </model>
```

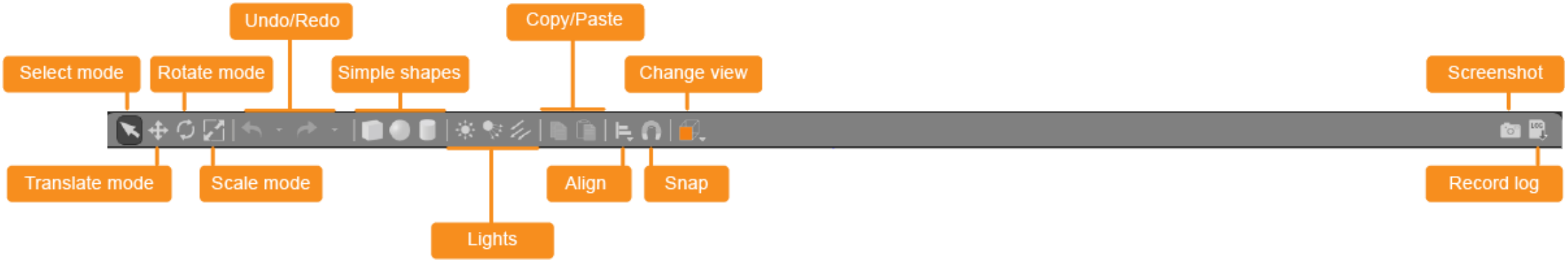
gedit ~/.gazebo/models/my_robot/model.config

```
<?xml version='1.0'?>
      <sdf version='1.4'>
        <model name="my_robot">
        <static>true</static>

          <link name='chassis'>
            <pose>0 0 .1 0 0 0</pose>

            <collision name='collision'>
              <geometry>
                <box>
                  <size>.4 .2 .1</size>
                </box>
              </geometry>
            </collision>

            <visual name='visual'>
              <geometry>
                <box>
                  <size>.4 .2 .1</size>
                </box>
              </geometry>
            </visual>
          </link>

        </model>
      </sdf>
```

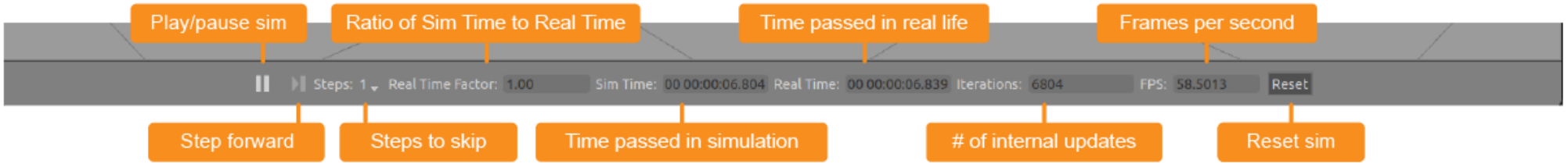
gedit ~/.gazebo/models/my_robot/model.sdf

가제보 툴 이해

1. 툴바(toolbars)



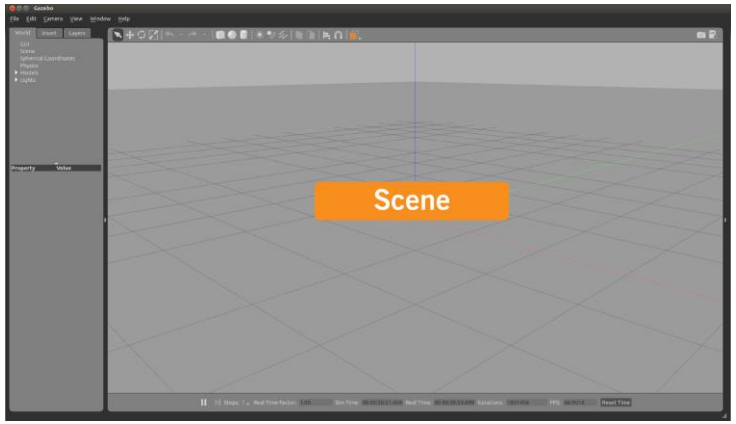
Upper Toolbar

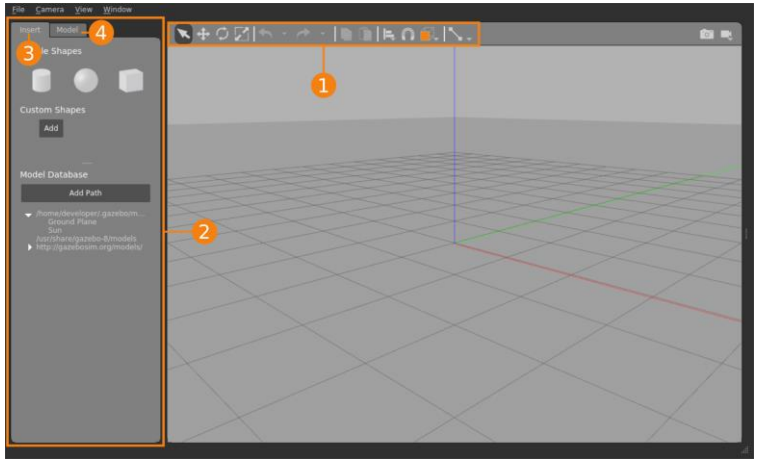


Bottom Toolbar

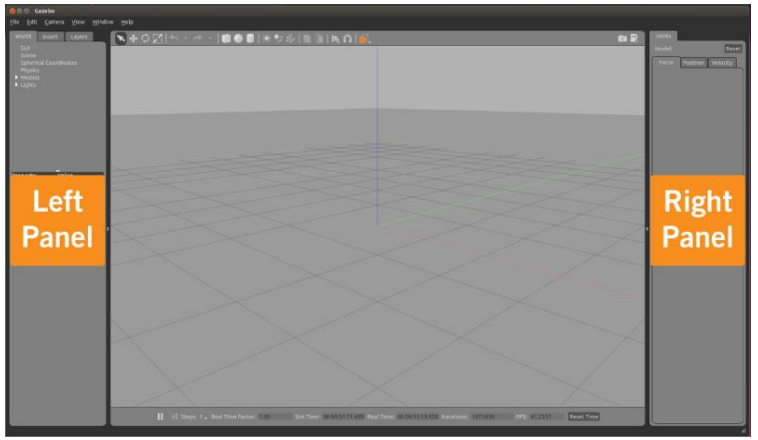
가제보 화면 용어

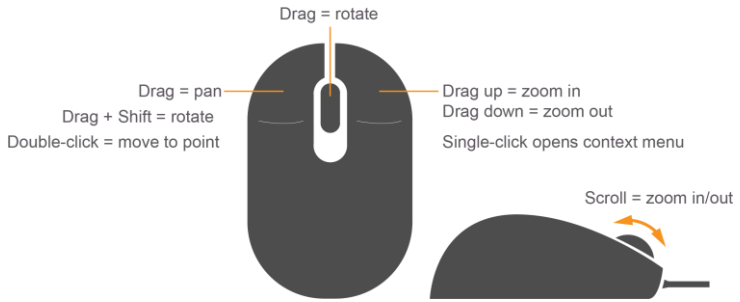
1. 씬(scene)





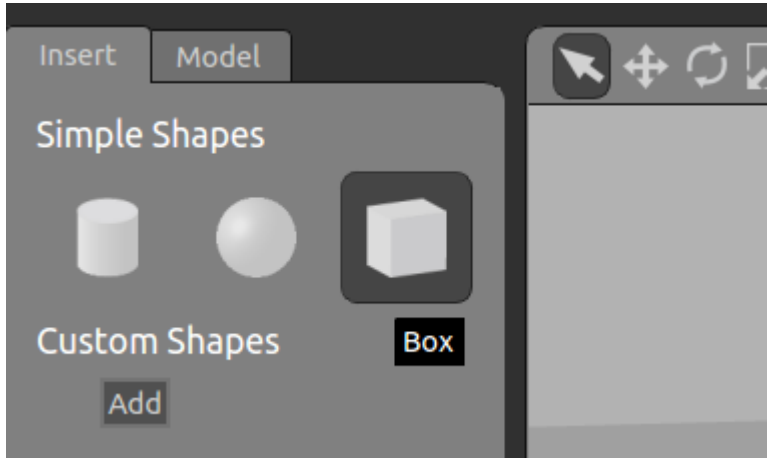
2. 판넬(panel)



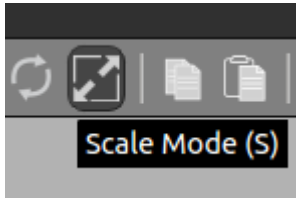


가상환경 만들기

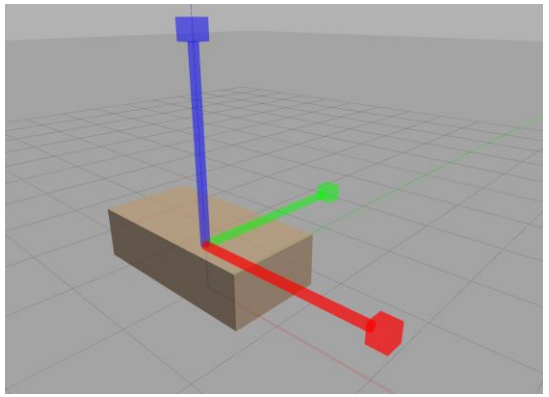
1. 샤시(Chassis)



2. 샤시 크기 조정

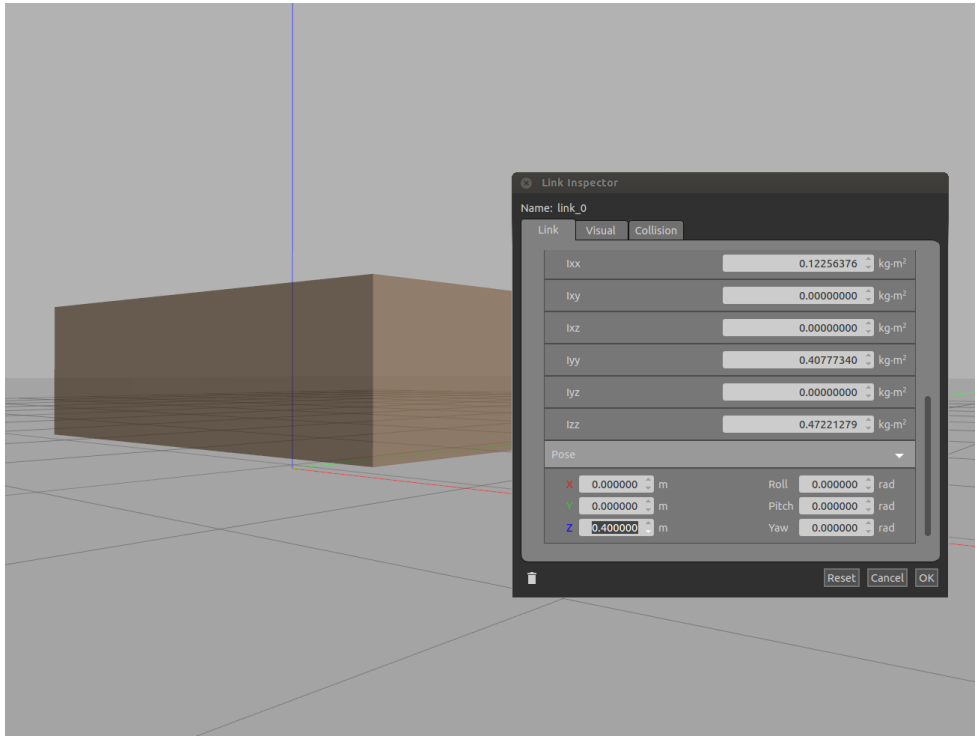


3. 싸이즈 줄이기(파란색으로 납작하게)

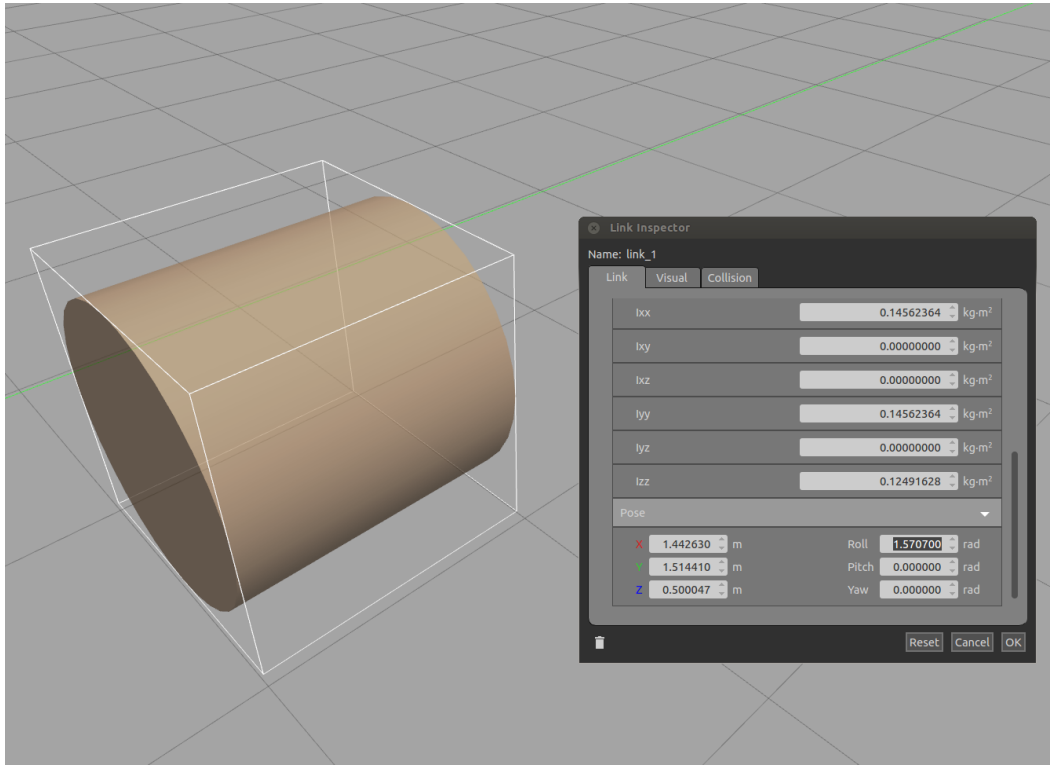


가상환경 만들기

4. Body 사시 만들기

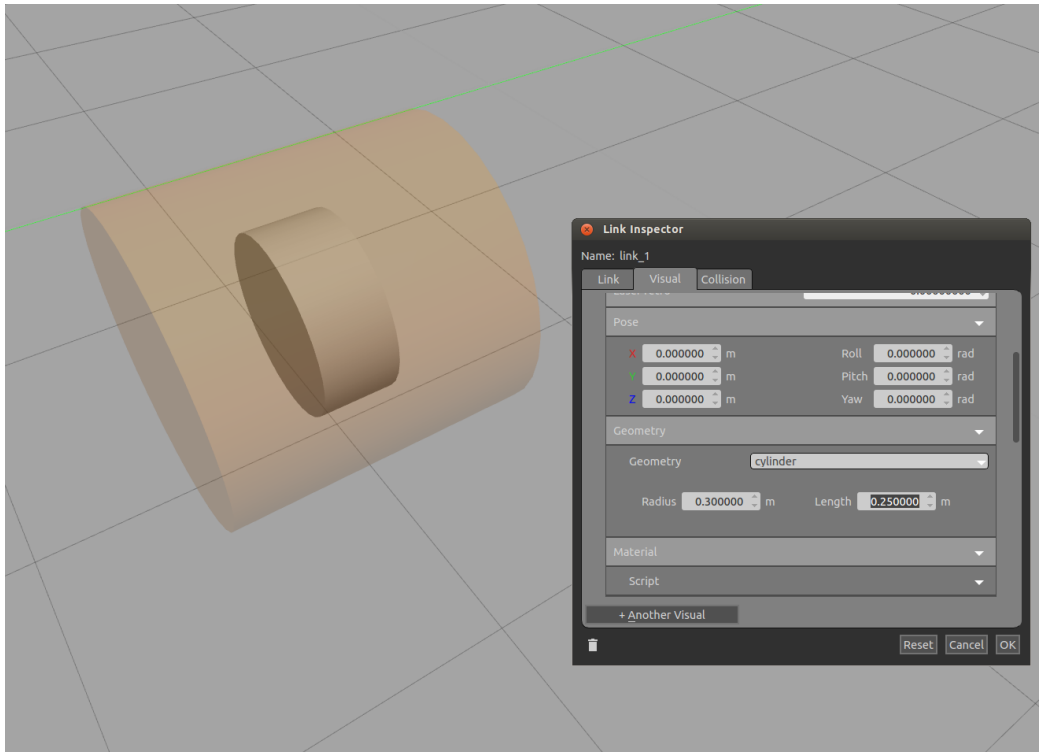


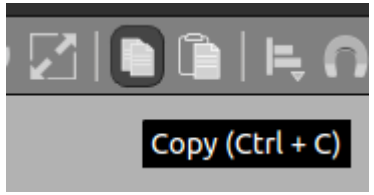
5. Front Wheels

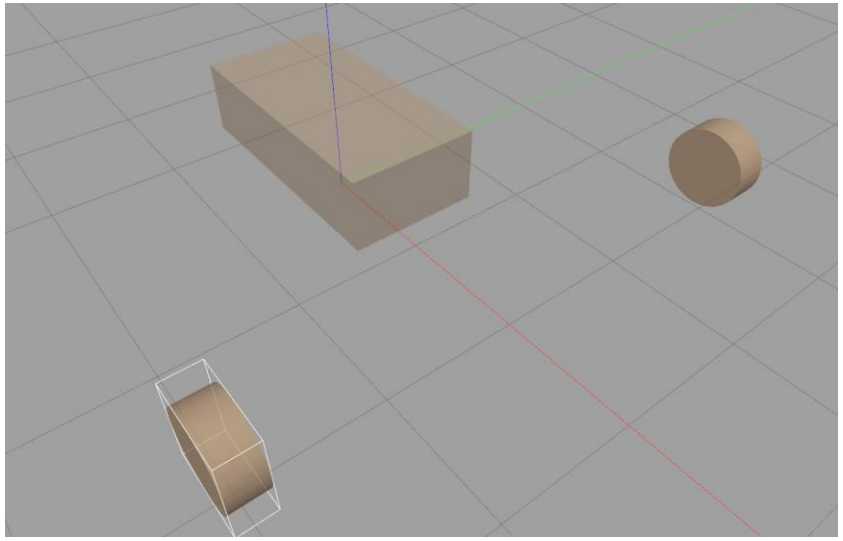


가상환경 만들기

6. 싸이즈 조절 및 복사



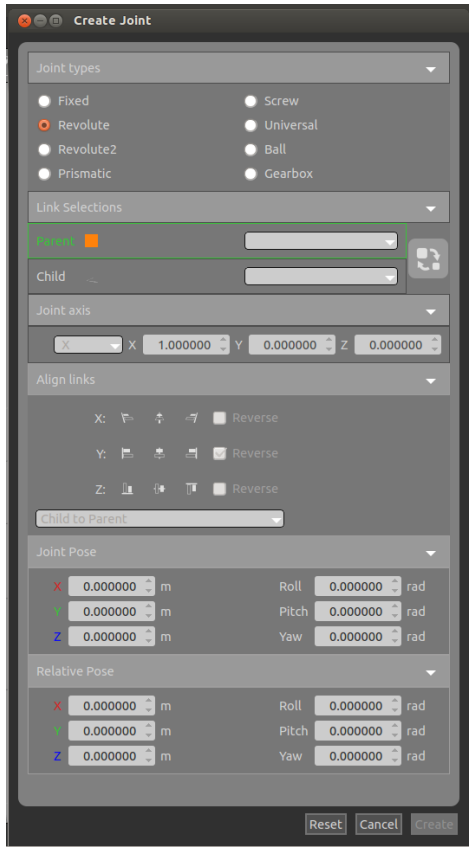


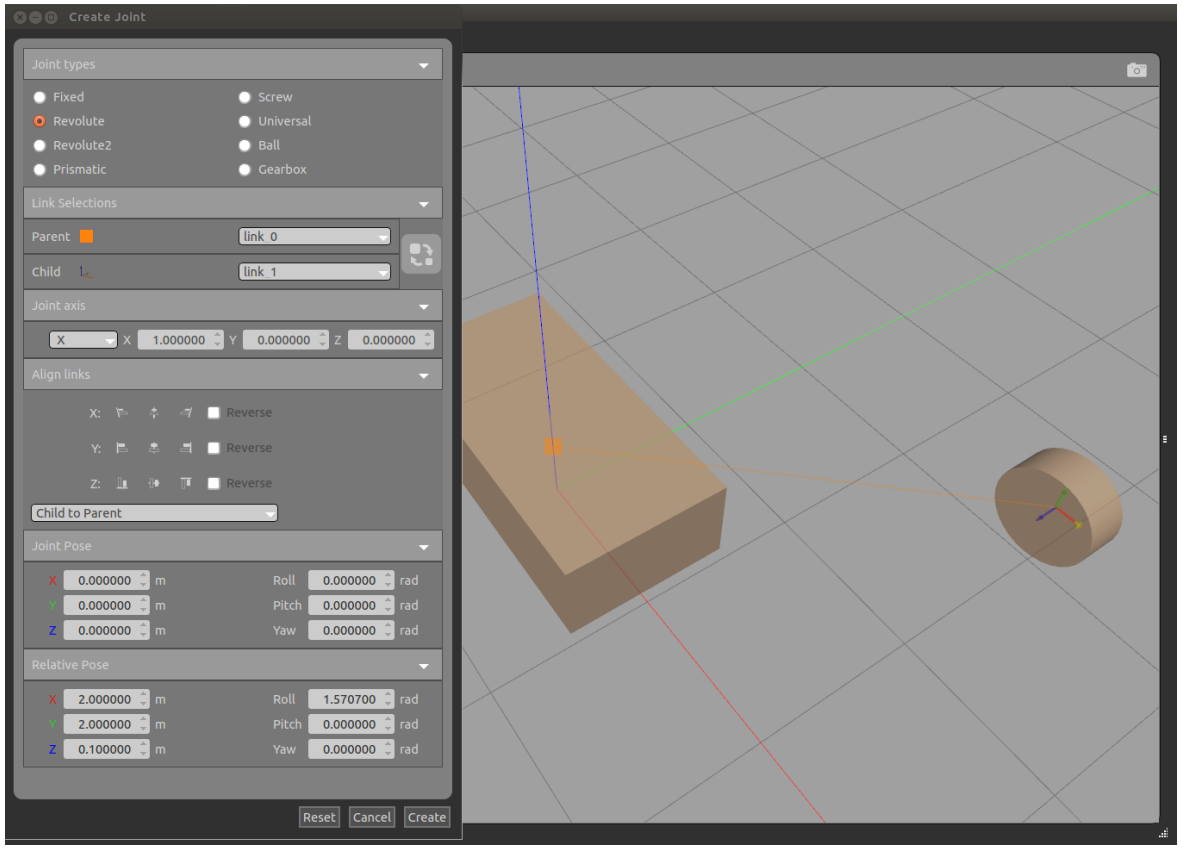


가상환경 만들기

7. 관절(Joint) 추가

상단 도구 모음에서 조인트 아이콘을 클릭하여 조인트 생성 대화 상자를 표시

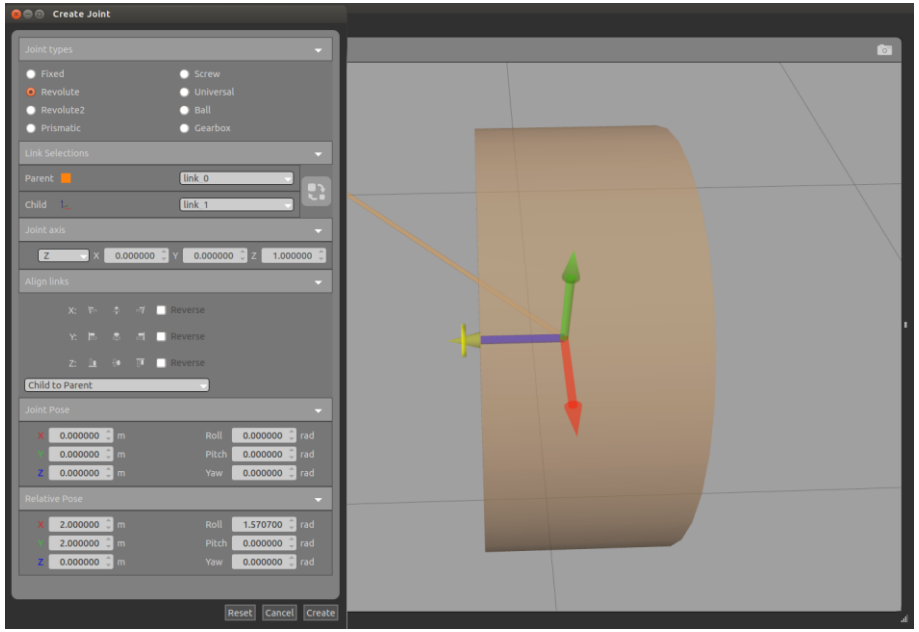




가상환경 만들기

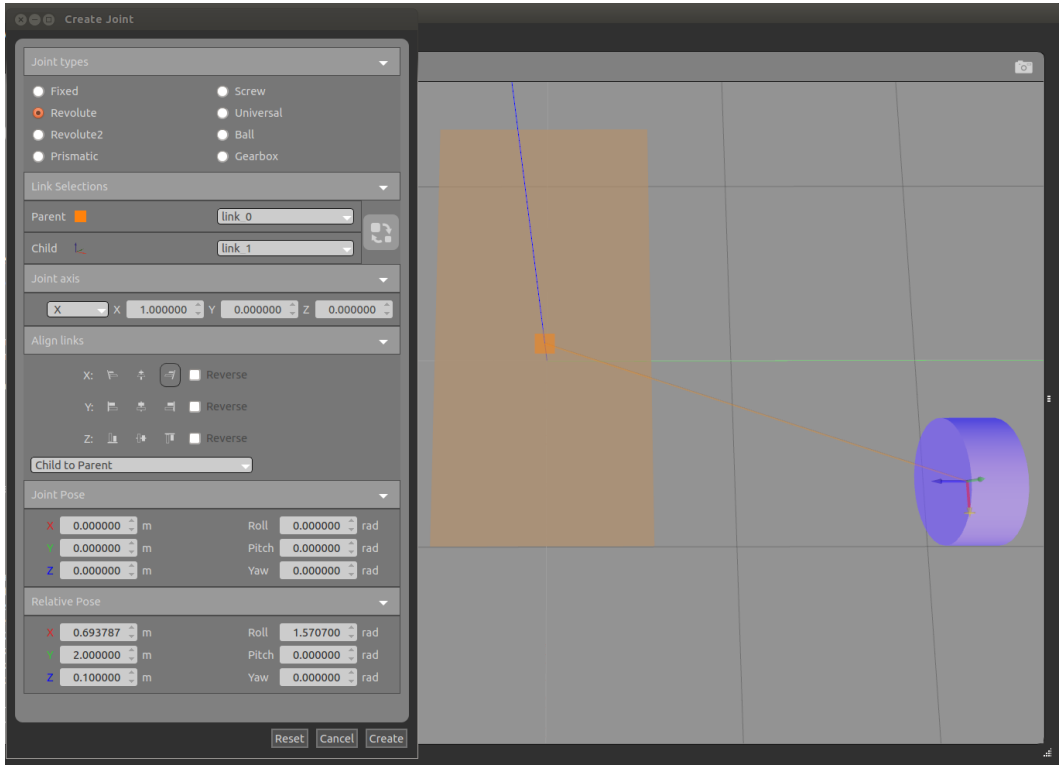
6. 휠 축 맞추기

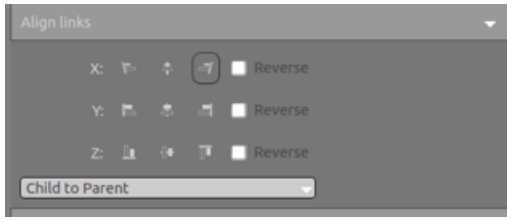
조인트 축 섹션을 변경하고 축을 Z(0, 0, 1)로 변경합니다.

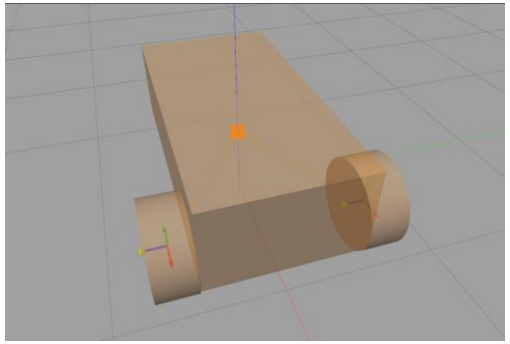


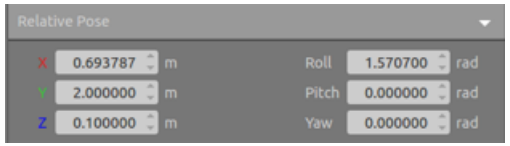
가상환경 만들기

7. 휠을 샤시와 어라인하기 그리고 왼쪽 바퀴도 만들기





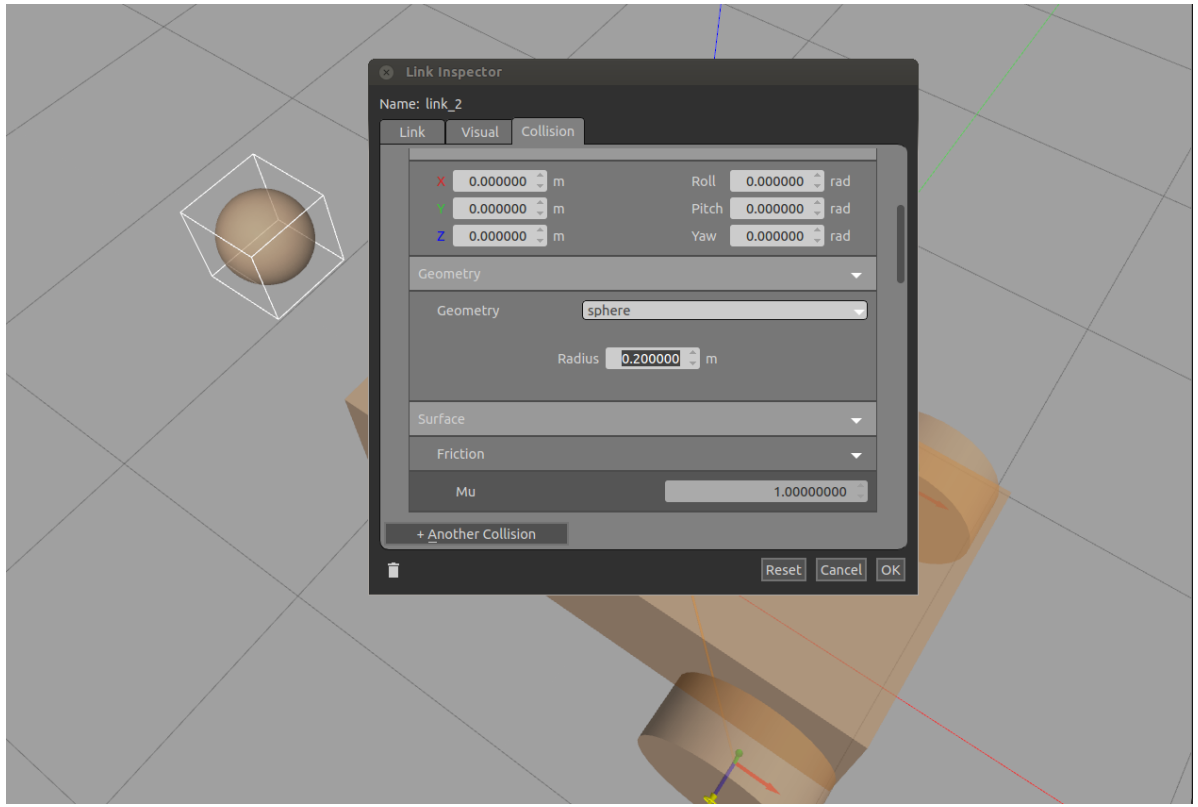


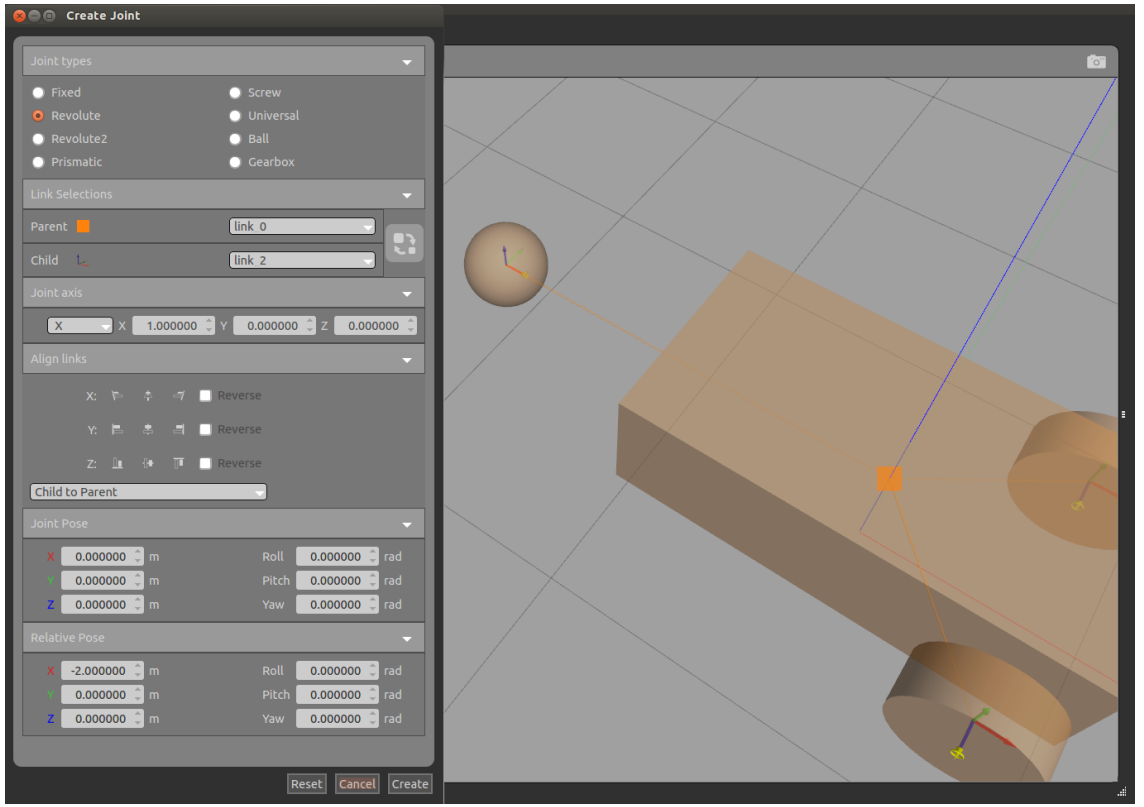


가상환경 만들기

8. 케스트 볼 만들기

기하학 섹션으로 스크롤하여 반경을 0.2m로 변경합니다.

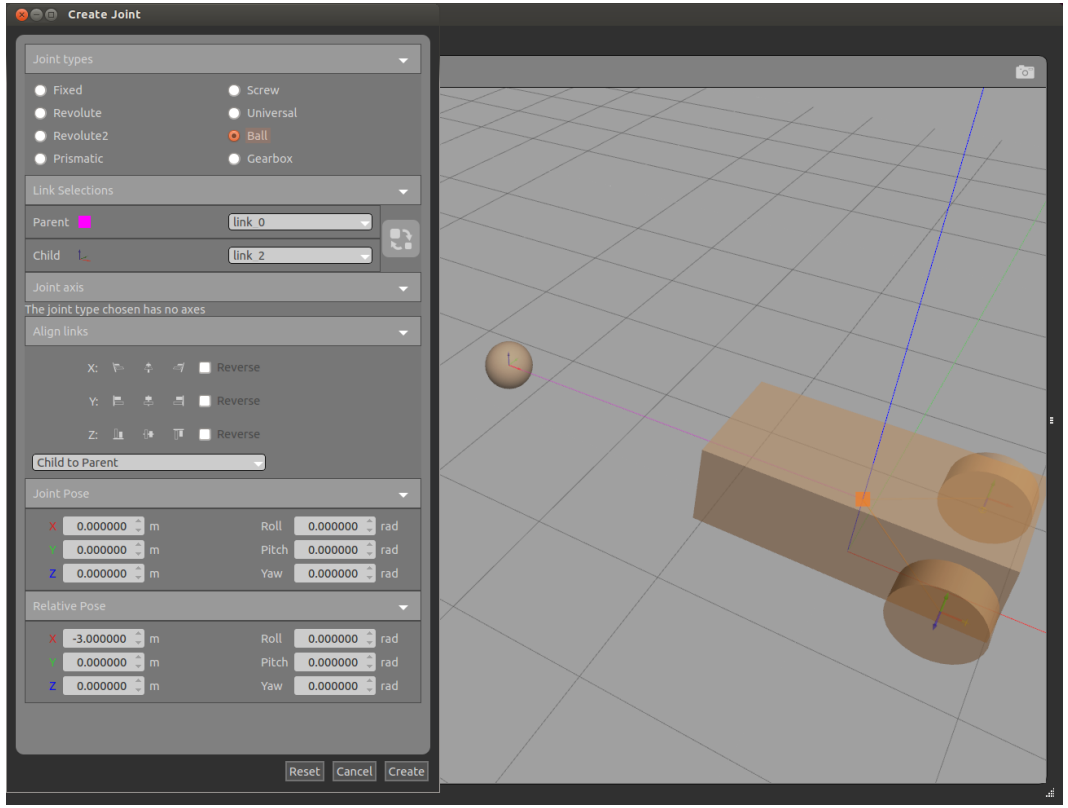


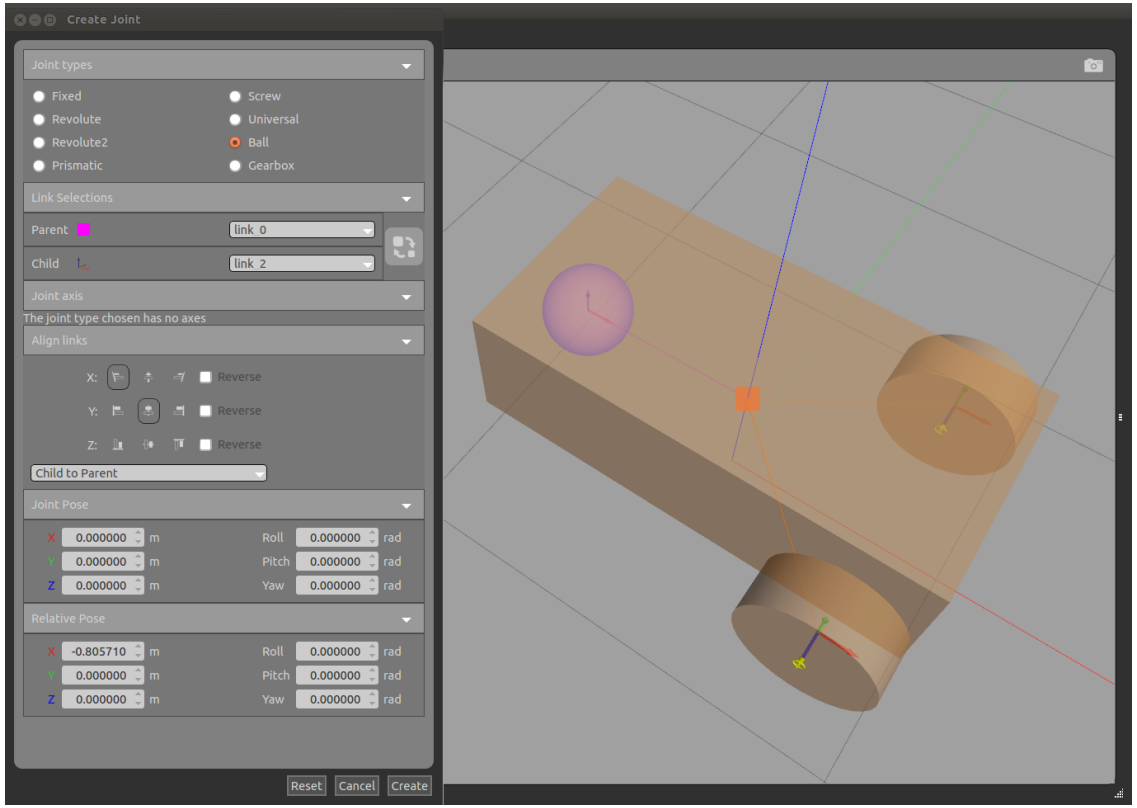


가상환경 만들기

9. 휠을 샤시와 어라인하기

조인트 유형 섹션에서 볼 조인트 옵션을 선택하세요.



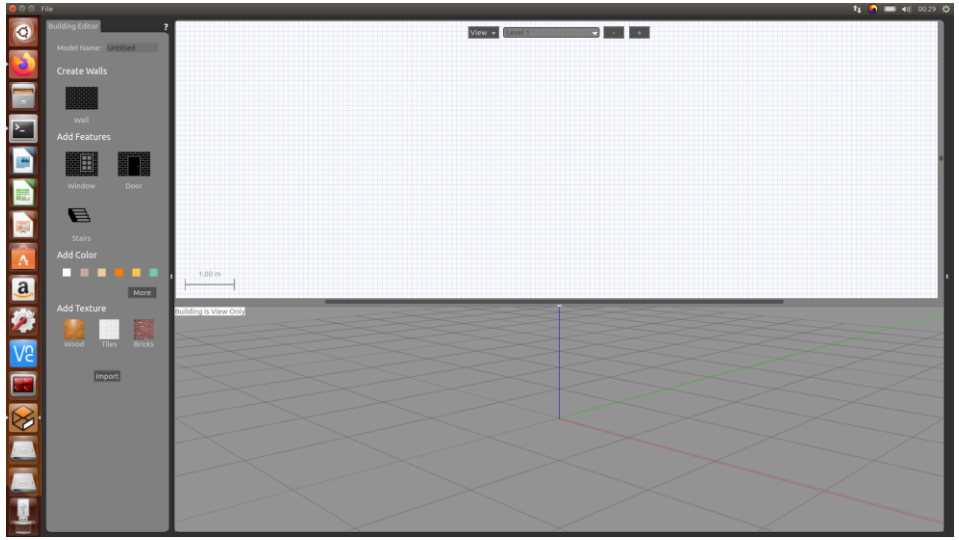


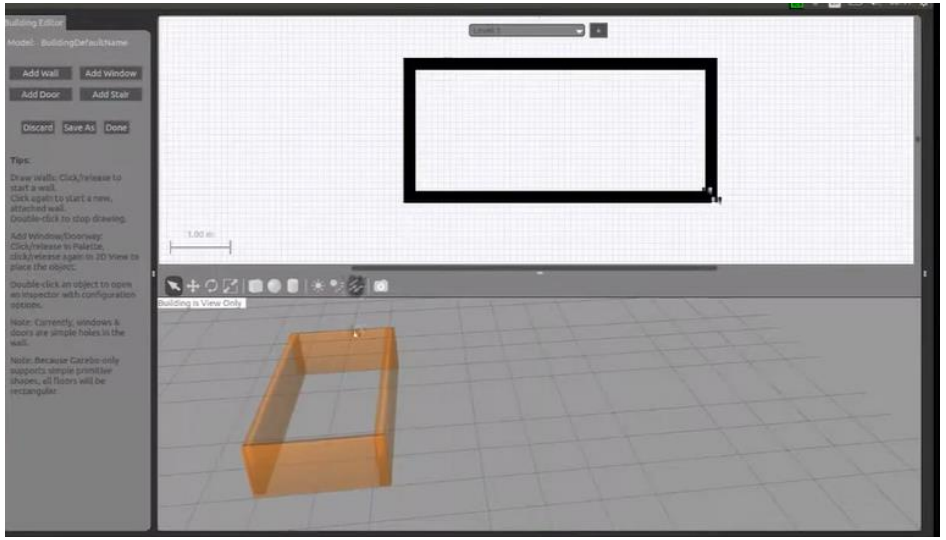
Y Align Center option to center the two links in the Y axis, and select the X Align Min option

가상환경 만들기

3D 맵 만들기

Edit 메뉴에서 building editor 를 클릭, 다음과 같은 building editor화면이 나타난다.

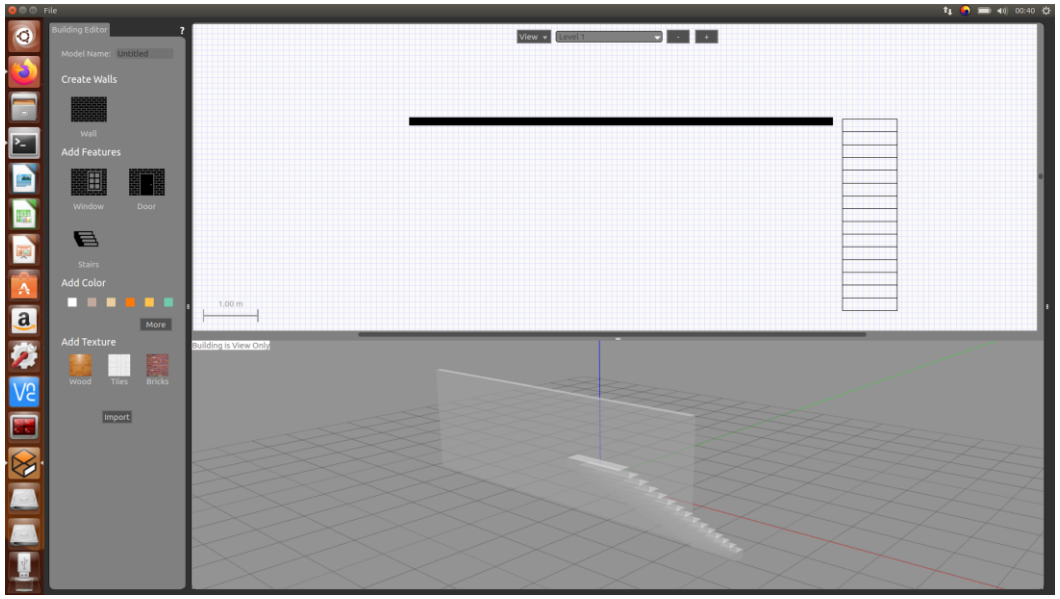




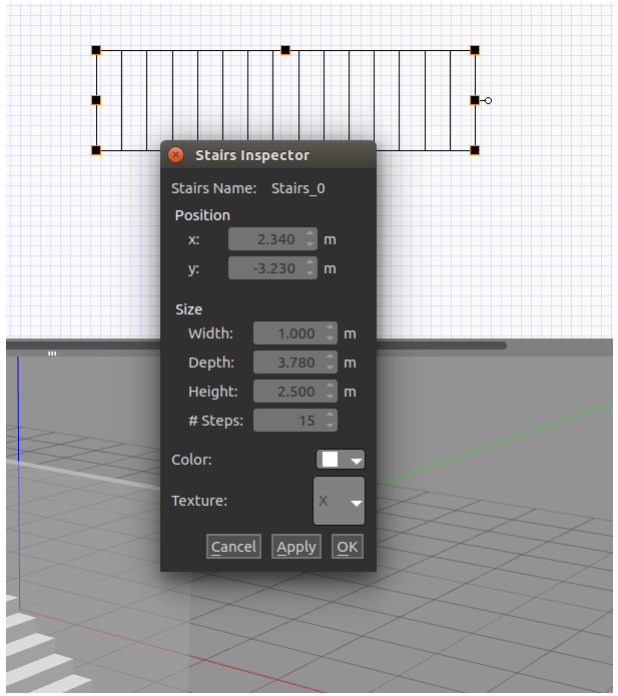
화면에서 왼쪽에는 벽, 문, 계단 등을 만들수 있는 항목들이 있고, 모눈종이처럼 생긴 구역에서 위에서 내려다보는 평면도 관점의 맵을 볼 수 있다. 이 모눈종이 구역에서 벽, 계단 등을 생성, 배치할 수 있다 아래쪽 구역은 실제 gazebo 공간상에 생성된 3차원 맵을 보여준다.

가상환경 만들기

왼쪽의 Create Walls의 Wall을 클릭하고 모눈종이 영역에서 직선을 그려주면, 벽이 생성된다. 직선의 길이, 위치 등을 변경하면 벽의 길이, 위치를 변경할수 있다.



왼쪽에서 Stairs 를 클릭하고, 모눈종이 영역을 클릭해주면 계단이 생성된다.

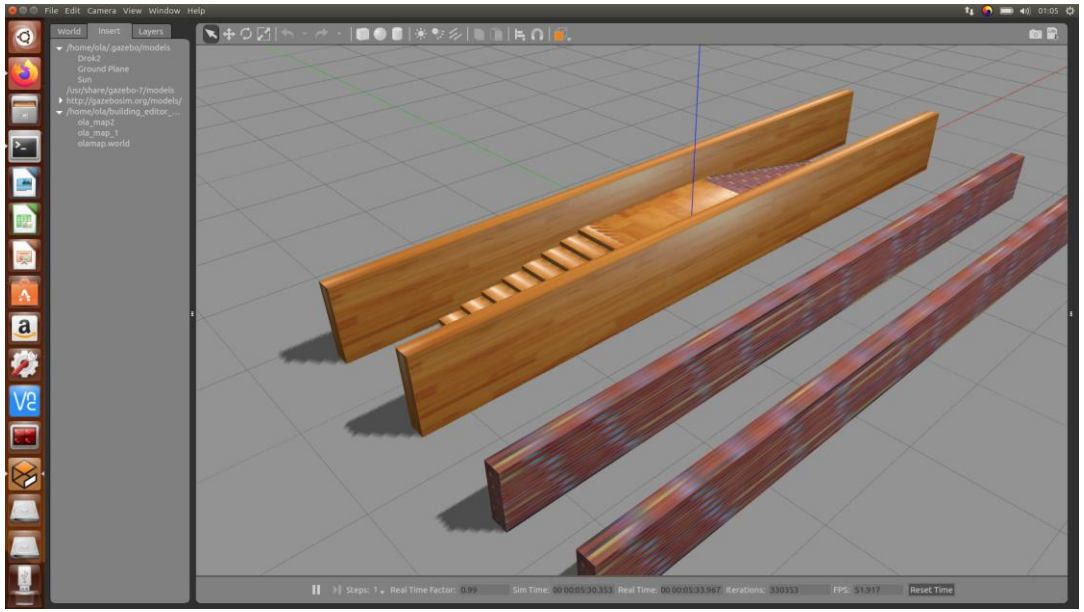


가상환경 만들기

Add texture 기능으로 물체의 표면 질감을 바꿔줄수 있다.

building editor 에서 맵을 다 만들었다면 왼쪽 상단의 File 탭에서 Save as..를 클릭,이름을 바꿔준 뒤, 저장한다.

저장한 뒤 File -> exit building editor 를 눌러 buildig editor 화면에서 나간다.



저장한 뒤 building editor에서 나가면, 가제보 공간에, 우리가 만든 구조물들이 나타나있다.

위 사진은 building editor에서 간단한 계단, 벽 등으로 만든 구조물이다.

이제 이 3D 맵을 *.world 파일로 저장하자.

왼쪽 상단의 File -> save world as 를 클릭한다.

참고: https://www.youtube.com/watch?v=7McYSJFAqlU

자동차 모델 만들기

10. 저장 및 리로드

ROS2 환경에서 가제보 모델 다음의 경로에 저장

1. 시스템 전체에서 접근 가능한 공용 가제보 모델 저장소
   `~/.gazebo/models/`: 사용자의 홈 디렉토리 내 `.gazebo/models/` 폴더

2. 특정 ROS2 패키지 내에 속한 모델들을 저장하는 공간
   `<ros2_workspace>/src/<package_name>/models/`: ROS2 워크스페이스의 패키지 내 `models/` 폴더

11. 저장된 모델을 로드

1. ros2 run gazebo_ros spawn_entity.py -entity robot_name   -file   /path/to/your/robot.urdf   -x 0 -y 0 -z 1

자동차 모델 만들기

11. 저장된 모델을 로드

2. launch 파일, `gazebo.spawn_sdf_model()` 또는 `gazebo.spawn_urdf_model()` 함수 사용

```
from gazebo_ros.node import GazeboRosPaths
# 1.번 경로의 모델 로드
gazebo.spawn_sdf_model('my_model', model_xml, '', initial_pose, 'world')

# 2.번 경로의 모델 로드
pkg_path = GazeboRosPaths.get_model_path('my_package', 'my_model')
gazebo.spawn_urdf_model('my_model', pkg_path, '', initial_pose, 'world')
```

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='gazebo_ros',
            executable='spawn_entity.py',
            arguments=[
                '-entity', 'my_robot',
                '-file', '/path/to/your/robot.urdf',
                '-x', '1.0',
                '-y', '2.0',
                '-z', '0.5'
            ],
            output='screen'
        )
    ])
```

메니퓨레이션 시뮬레이션

로봇 모델 패키지 만들기

1. 워크스페이스 폴더를 만들고 그 안으로 이동

```
$ cd ~
$ mkdir -p .gazebo/models/gongjang
$ cd .gazebo/models/gongjang/
```

2. 로봇 모델의 설정 파일을 만들고 편집

```
<?xml version="1.0"?>
  <model>
    <name>ganzang</name>
    <version>1.0</version>
    <sdf version='1.4'>model.sdf</sdf>
    <author>
      <name>sj kim</name>
      <email>*@gmail.com</email>
    </author>
    <description>project ganzang</description>

</model>
```

Model.config

참고: 가제보 설치

```
sudo apt-get install ros-humble-gazebo-ros
```

모델 설계하기

3. 로봇 모델의 실제 내용이 될 파일을 만들고 설계

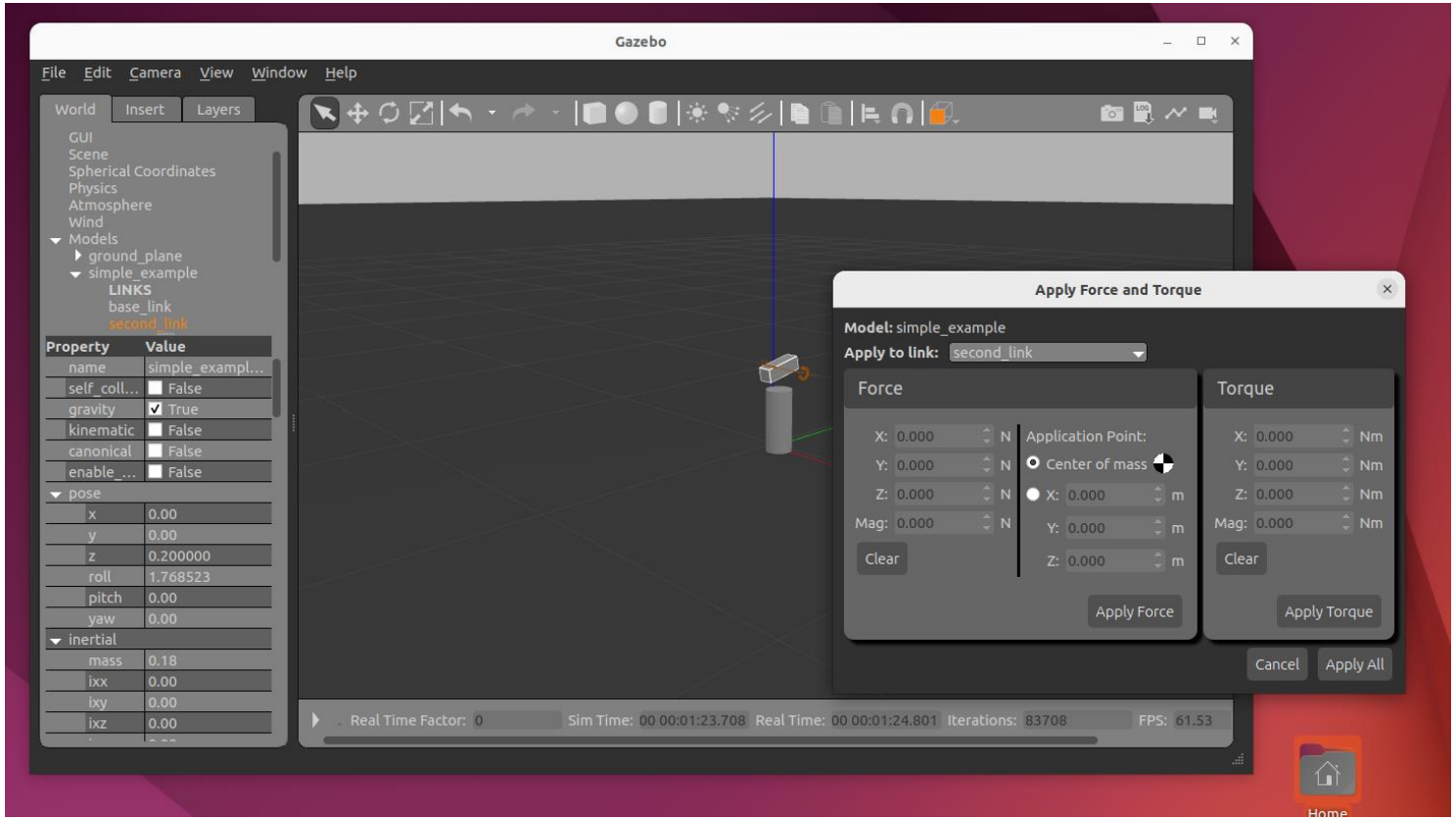
```
<?xml version='1.0'?>
<sdf version='1.4'>
<model name="ganzang">
<static>false</static>

<link name='link0'>
<pose>0 0 0.05 0 0 0</pose>
    <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>
</visual>
</link>
```

```
<link name="link1">
<pose>0 0.1 0.125 0 0 0</pose>
<collision name="collision">
<geometry>
<box>
<size>.05 .3 .05</size>
</box>
</geometry>
</collision>
<visual name="visual">
<geometry>
<box>
<size>.05 .3 .05</size>
</box>
</geometry>
</visual>
</link>
```

```
<joint type="revolute" name="joint0">
<pose>0 0 0 0 0 0</pose>
<child>link0</child>
<parent>world</parent>
<axis>
<xyz>0 0 0</xyz>
</axis>
</joint>

<joint type="revolute" name="joint1">
<pose>0 -0.1 0 0 0 0</pose>
<child>link1</child>
<parent>link0</parent>
<axis>
<limit>
<lower>-1</lower>
<upper>1</upper>
</limit>
<xyz>0 0 10</xyz>
</axis>
</joint>
</model>
```

모델 불러오기

4. 가제보의 왼쪽 페널에서 gongjang 모델을 마우스로 끌어온다.
여러가지 기능을 메뉴를 선택해서 학습하시기 바랍니다.



모델 설계하기

4. 컨트롤러 사용

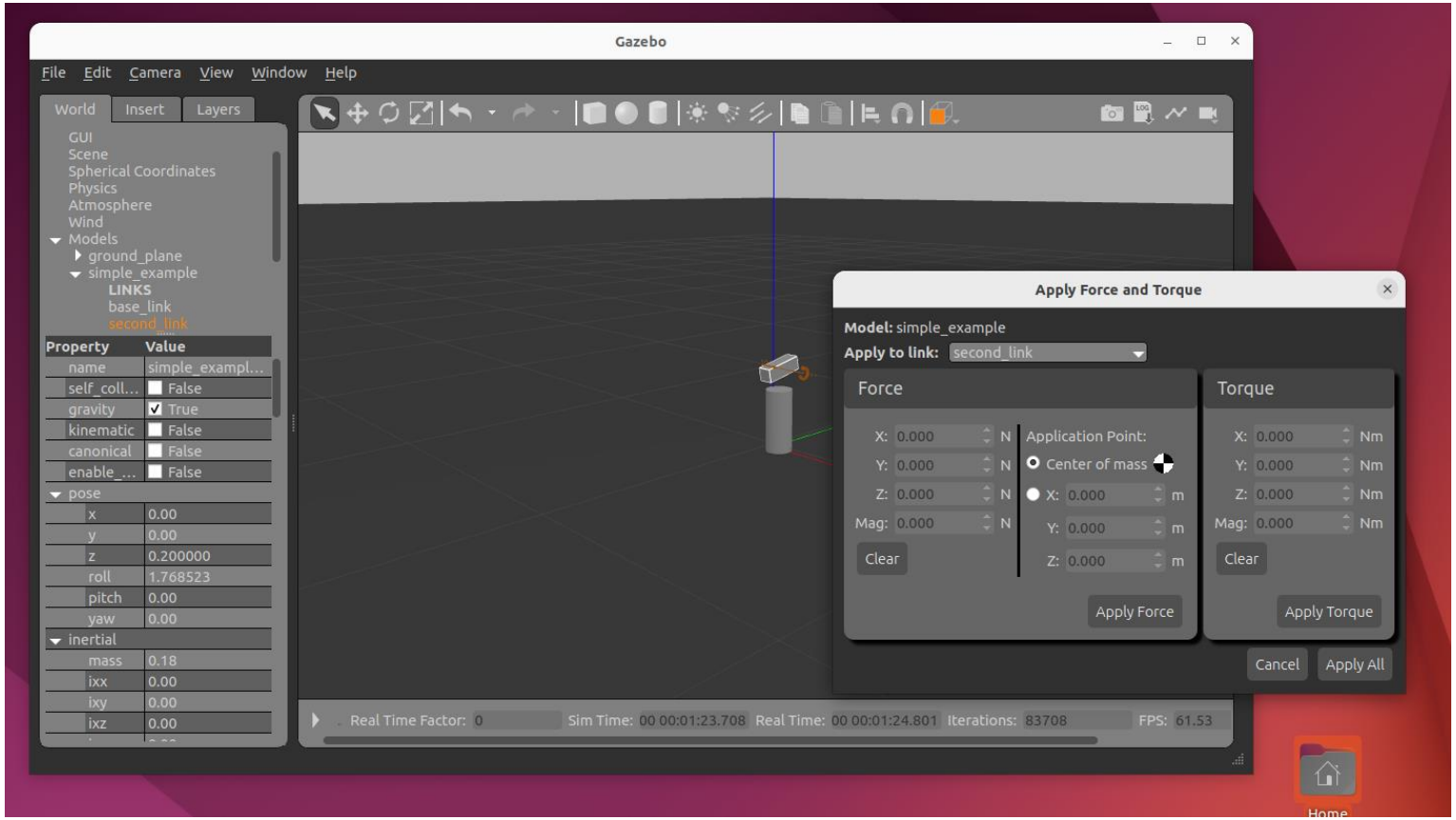
```
<?xml version="1.0"?>
<robot name="simple_example">
 <link name="base_link">
  <inertial>
    <mass value="10" />
    <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
  </inertial>
  <collision>
   <geometry>
    <cylinder radius="0.05" length="0.24" />
   </geometry>
    </collision>
  <visual>
   <geometry>
    <cylinder radius="0.05" length="0.24" />
   </geometry>
  </visual>
 </link>
```

```
<link name="second_link">
  <inertial>
    <mass value="0.18" />
    <inertia ixx="0.0002835" ixy="0.0" ixz="0.0" iyy="0.0002835" iyz="0.0" izz="0.000324" />
  </inertial>
  <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.0" />
  <collision>
   <geometry>
    <box size="0.05 0.05 0.15" />
   </geometry>
  </collision>
  <visual>
   <geometry>
    <box size="0.05 0.05 0.15" />
   </geometry>
  </visual>
</link>
```

```
<joint name="base_to_second_joint" type="continuous">
  <parent link="base_link"/>
  <child link="second_link"/>
  <axis xyz="1 0 0"/>
  <origin xyz="0.0 0.0 0.2" rpy="0.0 0.0 0.0"/>
 </joint>

 <!--           GAZEBO RELATED PART                    -->

 <!-- ROS Control plugin for Gazebo -->
 <gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
   <robotNamespace>/simple_model</robotNamespace>
  </plugin>
 </gazebo>

 <!-- transmission -->
 <transmission name="base_to_second_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <actuator name="motor1">
   <mechanicalReduction>1</mechanicalReduction>
  </actuator>
  <joint name="base_to_second_joint">
   <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
 </transmission>
</robot>
```

힘으로 동작시켜보기

모델을 선택 후 > Apply Force and Torgue 선택
힘(Force)와 토크(Torque) 설정 후 Play 버튼



World

.world 파일

- **Gazebo 시뮬레이션 환경을 정의하는 파일입니다.**
- **여러 개의 모델(로봇, 지형, 센서 등)을 포함할 수 있습니다.**
- **.sdf 파일을 불러오는 역할을 합니다.**

Gazebo의 **환경(월드, 조명, 지면, 로봇, 센서 등)** 을 정의
<include> 태그를 사용하여 **외부 모델(.sdf)** 을 불러옴

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <world name="my_simulation_world">
    <!-- 기본 지면 추가 -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- 조명 추가 -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- 로봇 모델 추가 (외부 SDF 파일 포함) -->
    <include>
      <uri>model://my_robot</uri>
    </include>
  </world>
</sdf>
```

.sdf 파일

단일 개체(로봇, 센서, 지형 등)를 정의하는 파일입니다.
로봇의 링크, 조인트, 센서, 물리적 특성 등을 포함합니다.
.world 파일 내에서 불러올 수도 있고, 개별적으로 실행할 수도 있습니다.

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <model name="my_robot">
    <static>false</static>
    <link name="base_link">
      <visual name="visual">
        <geometry>
          <box>
            <size>0.5 0.5 0.5</size>
          </box>
        </geometry>
      </visual>
      <collision name="collision">
        <geometry>
          <box>
            <size>0.5 0.5 0.5</size>
          </box>
        </geometry>
      </collision>
    </link>
  </model>
</sdf>
```

가제보 모델

Gazebo는 기본적으로 ~/.gazebo/models/ 디렉터리를 모델 경로로 인식

mkdir -p ~/.gazebo/models/my_custom_model

모델 폴더 구조

```
~/.gazebo/models/
└── my_custom_model/
    ├── model.sdf              # SDF 모델 파일
    ├── model.config           # 모델 정보 파일
    ├── materials/
    │   ├── scripts/
    │   │   └── custom_material.material  # 머티리얼 파일
    │   └── textures/
    │       └── my_texture.png  # 텍스처 이미지
    ├── meshes/                # (필요하면) STL 또는 COLLADA(.dae) 모델 파일
    └── README.md               # (선택 사항) 모델 설명 파일
```

[redacted]

.world vs .sdf 핵심 차이점

구분	`.world` 파일	`.sdf` 파일
역할	시뮬레이션 전체 환경 정의	개별 모델(로봇, 센서 등) 정의
내용	지형, 조명, 로봇 포함	특정 개체(로봇, 센서 등)의 링크, 조인트, 센서
불러오기	Gazebo 실행 시 직접 사용	`.world` 파일에서 `<include>`로 참조
예제	`gazebo my_world.world`	`gazebo --verbose my_robot.sdf`

.world 파일을 실행하려면 ROS2의 gazebo_ros 패키지를 사용합니다.

ros2 launch gazebo_ros gzserver.launch.py world:=/absolute/path/to/my_world.world

Gazebo의 .world 파일에서 특정 모델을 직접 지정하는 방법

특정 모델을 .world 파일에서 지정하는 방법
예를 들어, TurtleBot3를 포함한 커스텀 월드를 생성하려면 다음과 같은 .world 파일을 작성합니다.

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <world name="custom_world">
    <!-- 바닥 평면 -->
    <include>
      <uri>model:ground_plane</uri>
    </include>

    <!-- 태양 조명 -->
    <include>
      <uri>model:sun</uri>
    </include>

    <!-- TurtleBot3 로봇 추가 -->
    <include>
      <uri>model:turtlebot3_burger</uri>
      <pose>0 0 0.1 0 0 0</pose>
    </include>

  </world>
</sdf>
```

Gazebo의 .world 파일에서 특정 모델을 직접 지정하는 방법

특정 .world 파일을 포함하는 방법
다른 .world 파일을 현재 월드에 포함하려면 **<include>** 태그 안에 filename 속성을 사용합니다.
예를 들어, 기존 default.world를 custom.world에서 포함하려면:

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <world name="custom_world">
    <!-- 기존 default.world 포함 -->
    <include>
      <uri>file://path/to/default.world</uri>
    </include>
  </world>
</sdf>
```

Gazebo의 .world 파일에서 특정 모델을 직접 지정하는 방법

모델 파일의 위치 확인 및 추가

Gazebo는 기본적으로 /usr/share/gazebo-11/models/ 디렉터리에서 모델을 찾습니다.
사용자 지정 모델을 사용하려면 GAZEBO_MODEL_PATH를 설정해야 합니다.

```
<include>
  <uri>model:my_custom_robot</uri>
</include>
```

직접 .world 파일을 실행하는 방법
저장한 .world 파일을 실행하려면

```
ros2 launch gazebo_ros gzserver.launch.py world:=/absolute/path/to/my_custom_world.world
```

SDF(Scenario Description Format)

SDF(Scenario Description Format, Simulation Description Format)

SDF: Gazebo에서 로봇, 센서, 환경 등을 정의하는 XML 기반의 파일 형식입니다.

ROS2와 함께 사용하면 로봇의 구조, 물리적 특성, 센서 모델 등을 정의하고 시뮬레이션 환경을 설정할 수 있습니다.

SDF는 URDF보다 더 정밀한 시뮬레이션을 지원하며, Gazebo의 물리 엔진과 긴밀하게 연결됩니다.

SDF 주요 요소

1) <world> - 월드 정의
월드는 Gazebo에서 시뮬레이션할 전체 환경을 정의하는 요소입니다.

<model> - 로봇 및 오브젝트 정의
•로봇이나 오브젝트를 정의하는 요소로, <link>, <joint> 등을 포함할 수 있습니다.
•static이 true이면 해당 오브젝트는 고정된 상태가 됩니다.

<link> - 모델의 물리적 요소
•로봇의 구성 요소(바퀴, 몸체 등) 또는 오브젝트의 물리적 특성을 정의합니다.
•<collision>과 <visual>을 포함할 수 있습니다.

SDF

SDF 주요 요소

<joint> - 링크 간 연결
•링크들 사이의 구동 방식(회전, 이동)을 정의하는 요소입니다.
•type="revolute"는 회전 조인트, type="prismatic"는 선형 조인트입니다.

<sensor> - 센서 모델 정의
•카메라, LIDAR, IMU 등의 센서를 정의할 수 있습니다.

<include> - 외부 모델 추가
•Gazebo의 기본 모델을 불러오거나 외부 SDF 파일을 로드할 때 사용됩니다.

```
ros2 launch gazebo_ros gazebo.launch.py world:=/path/to/my_world.sdf
```

Sdf 만들기 실습

로봇 모델 만들기 준비

폴더를 만들고 시작
 cd ~
mkdir –p .gazebo/models/myrobot
cd .gazebo/models/myrobot
vi model.config

```
<?xml version="1.0"?>
  <model>
     <name>myrobot</name>
     <version>1.0</version>
     <sdf version='1.4'>model.sdf</sdf>
     <author>
        <name>kim rujin</name>
        <email>rujinkim32@gmail.com</email>
     </author>
     <description>project rokey</description>
</model>
```

model.config

Sdf 파일 모델 파일

static은 이 모델이 움직이는 것인지 아닌지를 표시,앞으로 움직일 '로봇'인지 바닥 위에 놓여질 '장애물'인지를 설정.
True이면 이 후에 플러그인이나 파라메터를 전달해도 움직이 않음.
로봇 모델일 경우엔 false로 입력

```
<?xml version='1.0'?>
<sdf version='1.4'>
 <model name="myrobot">
 <static>false</static>

 <link name='link0'>
 <pose>0 0 0.05 0 0 0</pose>
     <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>
 </visual>
 </link>
```

```
<link name="link1">
 <pose>0 0.1 0.125 0 0 0</pose>

 <collision name="collision">
 <geometry>
 <box>
 <size>.05 .3 .05</size>
 </box>
 </geometry>
 </collision>
 <visual name="visual">
 <geometry>
 <box>
 <size>.05 .3 .05</size>
 </box>
 </geometry>
 </visual>
 </link>

 <joint type="revolute" name="joint0">
 <pose>0 0 0 0 0 0</pose>
 <child>link0</child>
 <parent>world</parent>
 <axis>
 <xyz>0 0 0</xyz>
 </axis>
 </joint>
```

```
 <joint type="revolute" name="joint1">
 <pose>0 -0.1 0 0 0 0</pose>
 <child>link1</child>
 <parent>link0</parent>
 <axis>
 <limit>
   <lower>-1</lower>
  <upper>1</upper>
 </limit>
 <xyz>0 0 10</xyz>
 </axis>
 </joint>

 </model>
</sdf>
```

Model.sdf

Sdf 만들기 실습

world, static, dynamic 객체

```xml
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <!-- World 요소 -->
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://ground_plane</uri>
    </include>
```

```xml
<!-- Static 모델 예제 -->
<model name="static_box">
  <static>true</static>
  <link name="link">
    <pose>0 0 0.5 0 0 0</pose>
    <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>
    </visual>
  </link>
</model>
```

```xml
<!-- Dynamic 모델 예제 -->
<model name="dynamic_sphere">
  <static>false</static>
  <link name="link">
    <pose>0 0 5 0 0 0</pose>
    <collision name="collision">
      <geometry>
        <sphere>
          <radius>0.5</radius>
        </sphere>
      </geometry>
    </collision>
    <visual name="visual">
      <geometry>
        <sphere>
          <radius>0.5</radius>
        </sphere>
      </geometry>
    </visual>
    <inertial>
      <mass>1.0</mass>
      <inertia>
        <ixx>0.1</ixx>
        <ixy>0.0</ixy>
        <ixz>0.0</ixz>
        <iyy>0.1</iyy>
        <iyz>0.0</iyz>
        <izz>0.1</izz>
      </inertia>
    </inertial>
  </link>
</model>
```

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

Inertial

Gazebo는 로봇 부품(즉, 링크)의 무게와 관성을 알아야 하는데, 이는 이 무게가 어떻게 분포되는지를 측정

```
<inertial>
  <mass value="2.275"/>
  <origin rpy="0 0 0" xyz="0.0 0.0 0.25"/>
  <inertia ixx="0.049443313556" ixy="0.0" ixz="0.0" iyy="0.049443313556" iyz="0.0" izz="0.004095"/>
</inertial>
```

$$\begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

effort로봇에 적용할 수 있는 최대 힘(또는 토크)을 결정하는데, 이는 주로 모터의 힘에 따라 달라집니다.

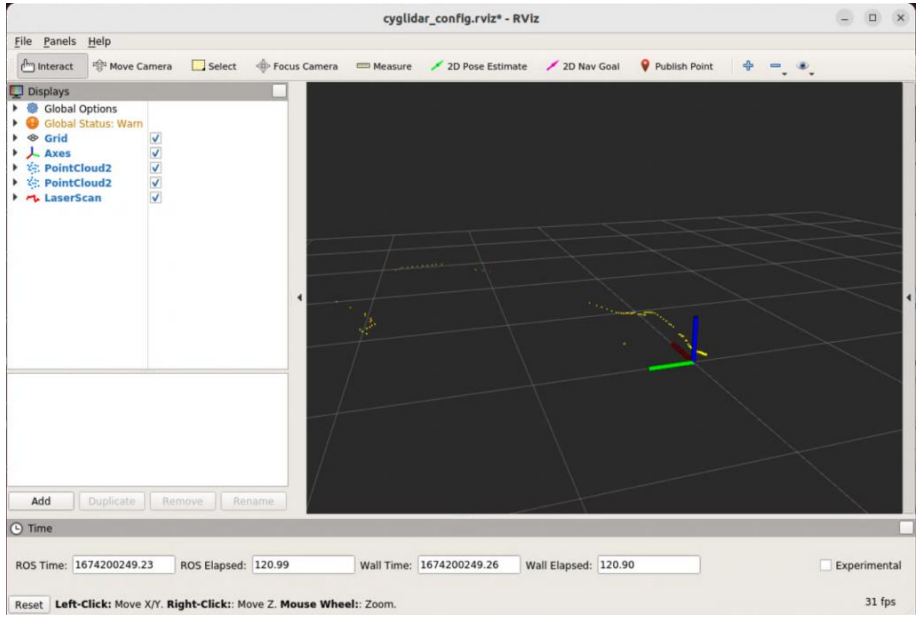
태그 <origin>는 각각 링크의 질량 중심 또는 중력 중심을 정의합니다.
이는 질량에 따라 가중치가 부여된 시스템의 모든 부분의 평균 위치입니다. 가속도를 계산할 때 유용한 속성입니다.

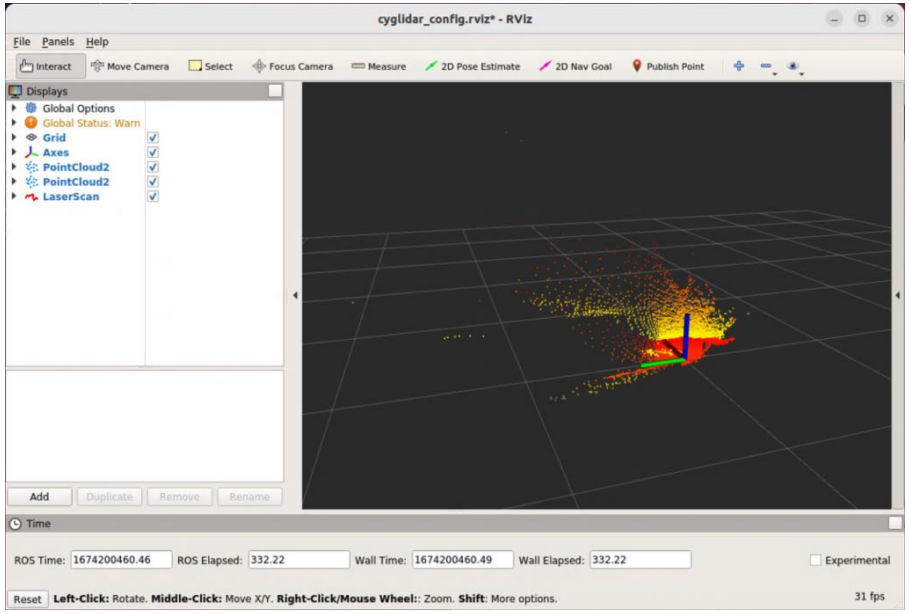
실습과제 2 –라이다 시뮬레이션

라이다 정보 시각화

라이다(LIDAR/LiDAR, light detection and ranging” 또는 “laser imaging, detection, and ranging”의 약자))는 레이저 펄스를 쏘고 반사되어 돌아오는 시간을 측정하여 반사체의 위치좌표를 측정하는 레이다 시스템

2D LiDAR는 관찰 지점으로부터 2차원 상의 평면 정보를 제공합니다.
즉 센서와 평행한 평면을 기준으로 물체와의 거리가 얼마나 떨어져 있는지 확인





패키지 만들기, 기본 모델 만들기

라이다 기능을 모델에 추가하고 gazebo에서 실행하고 그 정보를 Rviz2에서 시각화 확인할 수 있다.

```
$ cd ~/sim_ws
$ cp src/urdf_tutorial/urdf/robot_3.xacro src/urdf_tutorial/urdf/robot_4.xacro
```

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.substitutions import LaunchConfiguration
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node
import xacro

def generate_launch_description():
    use_sim_time = LaunchConfiguration("use_sim_time")

    pkg_path = os.path.join(get_package_share_directory("urdf_tutorial"))
    xacro_file = os.path.join(pkg_path, "urdf", "robot_4.xacro")
    robot_description = xacro.process_file(xacro_file)
    params = {"robot_description": robot_description.toxml(), "use_sim_time": use_sim_time}

    return LaunchDescription(
        [
            DeclareLaunchArgument(
                "use_sim_time", default_value="false", description="use sim time"
            ),
            Node(
                package="robot_state_publisher",
                executable="robot_state_publisher",
                output="screen",
                parameters=[params],
            ),
        ]
    )
```

Robot_state_publisher는 robot_description을 토픽으로

런치파일 만들기

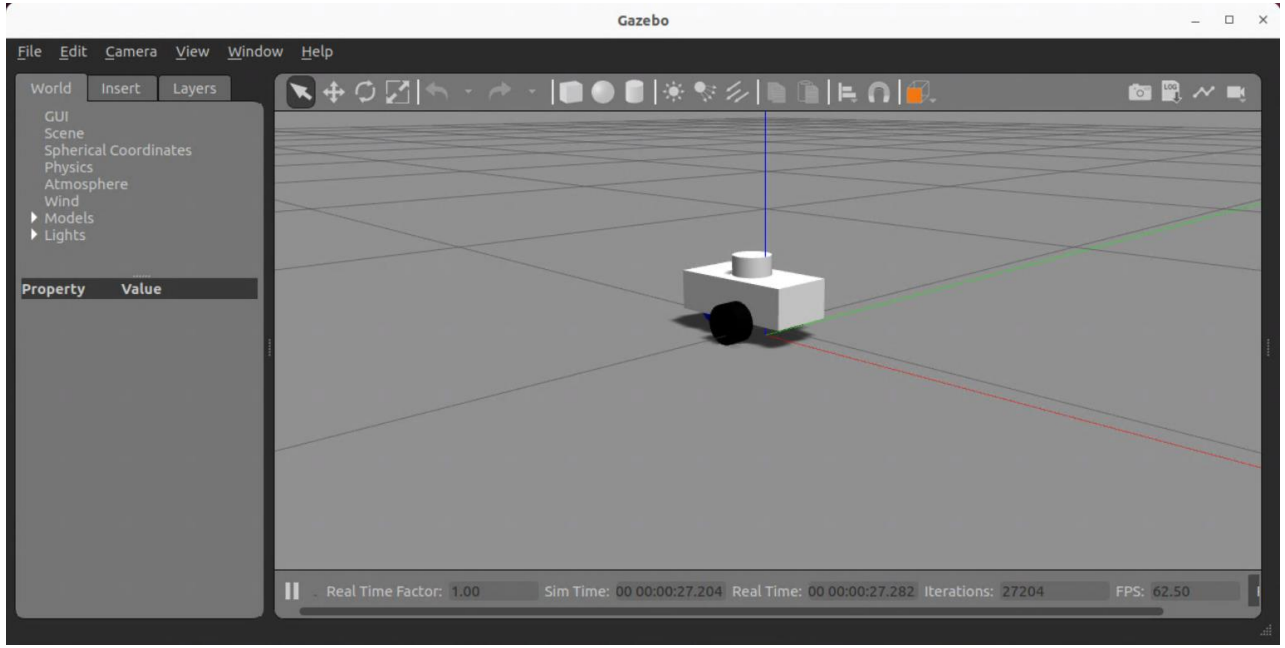
```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node


def generate_launch_description():
    package_name = "urdf_tutorial"

    rsp = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            [os.path.join(get_package_share_directory(package_name), "launch", "robot_4.launch.py")]
        ),
        launch_arguments={"use_sim_time": "true"}.items(),
    )

    # Include the Gazebo launch file, provided by the gazebo_ros package
    gazebo = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            [os.path.join(get_package_share_directory("gazebo_ros"), "launch", "gazebo.launch.py")]
        ),
    )
```

```
# Run the spawner node from the gazebo_ros package.
    spawn_entity = Node(
        package="gazebo_ros",
        executable="spawn_entity.py",
        arguments=["-topic", "robot_description", "-entity", "with_robot"],
        output="screen",
    )

    # Launch them all!
    return LaunchDescription(
        [
            rsp,
            gazebo,
            spawn_entity,
        ]
    )
```

패키지 빌드 후 가제보 실행

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

<!-- LiDAR -->
   <joint name="laser_joint" type="fixed">
      <parent link="body"/>
      <child link="laser_frame"/>
      <origin xyz="0.1 0 0.075" rpy="0 0 0"/>
   </joint>

   <link name="laser_frame">
      <visual>
         <geometry>
            <cylinder radius="0.03" length="0.03"/>
         </geometry>
         <material name="red"/>
      </visual>
      <collision>
         <geometry>
            <cylinder radius="0.03" length="0.03"/>
         </geometry>
      </collision>
   </link>
```

```
$ cd ~/sim_ws/
$ colcon build --symlink-install
$ source install/setup.bash
$ ros2 launch urdf_tutorial lidar.launch.py
```



라이다 정보 모델 만들기

로봇 모델에 라이다 모델 추가

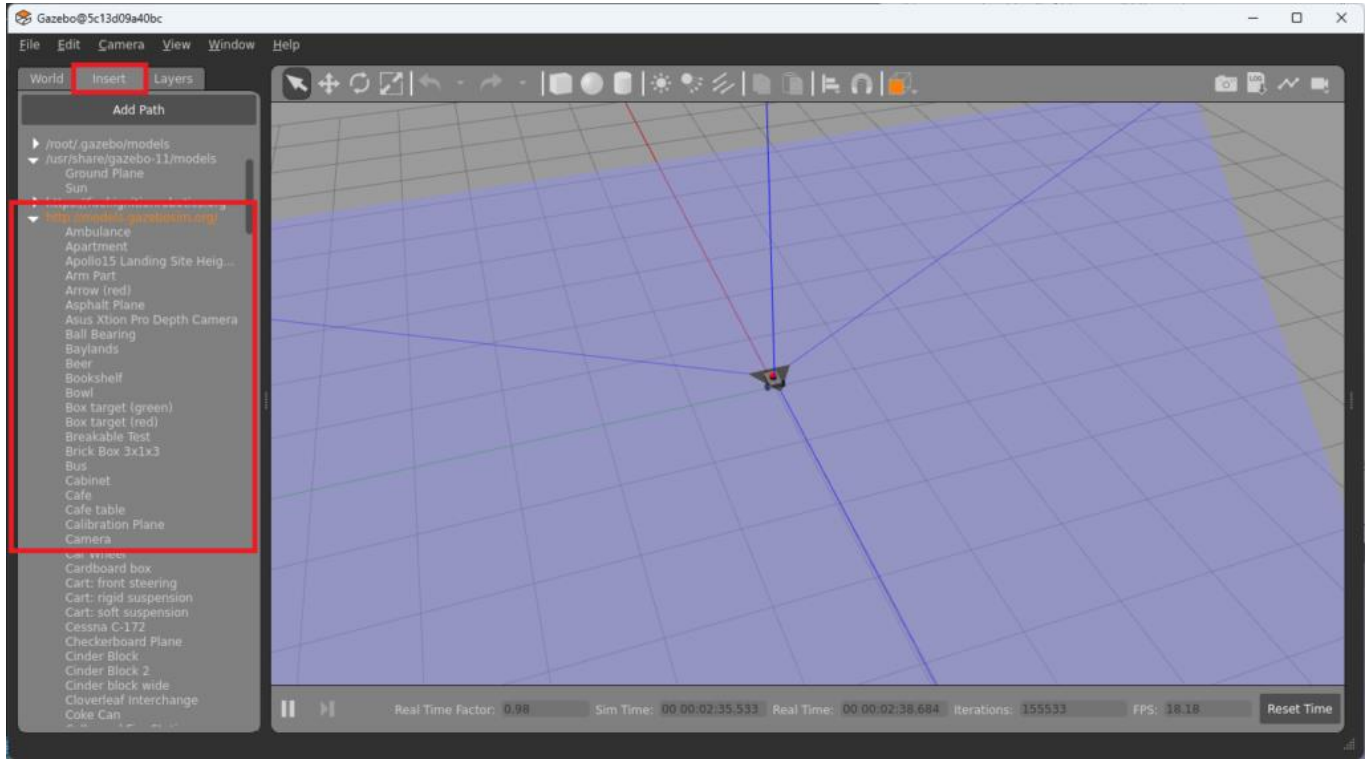
'src/urdf_tutorial/urdf/lidar.xacro' 파일
- update_rate: 초당 10회 정보를 제공합니다.
- samples: 검색 구간 내에서 4개의 광선을 발사합니다.
- min_angle, max_angle: 2D LiDAR가 스캔할 구간입니다.
- range min/max: 2D LiDAR가 스캔 가능한 거리의 최소, 최대 값입니다.
- argument: ros에 전달할 topic 이름입니다.
- output_type: 출력 형식입니다.
- frame_name: 2D LiDAR가 부착된 센서 이름입니다.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <gazebo reference="laser_frame">
    <material>Gazebo/Red</material>
    <sensor name="laser" type="ray">
      <pose>0 0 0 0 0 0</pose>
      <visualize>true</visualize>
      <update_rate>10</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>4</samples>
            <min_angle>-3.14</min_angle>
            <max_angle>3.14</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.3</min>
          <max>12</max>
        </range>
      </ray>
      <plugin name="laser_controller" filename="libgazebo_ros_ray_sensor.so">
        <ros>
          <argument>~/out:=scan</argument>
        </ros>
        <output_type>sensor_msgs/LaserScan</output_type>
        <frame_name>laser_frame</frame_name>
      </plugin>
    </sensor>
  </gazebo>

</robot>
```

```
<xacro:include filename="lidar.xacro"/>
```

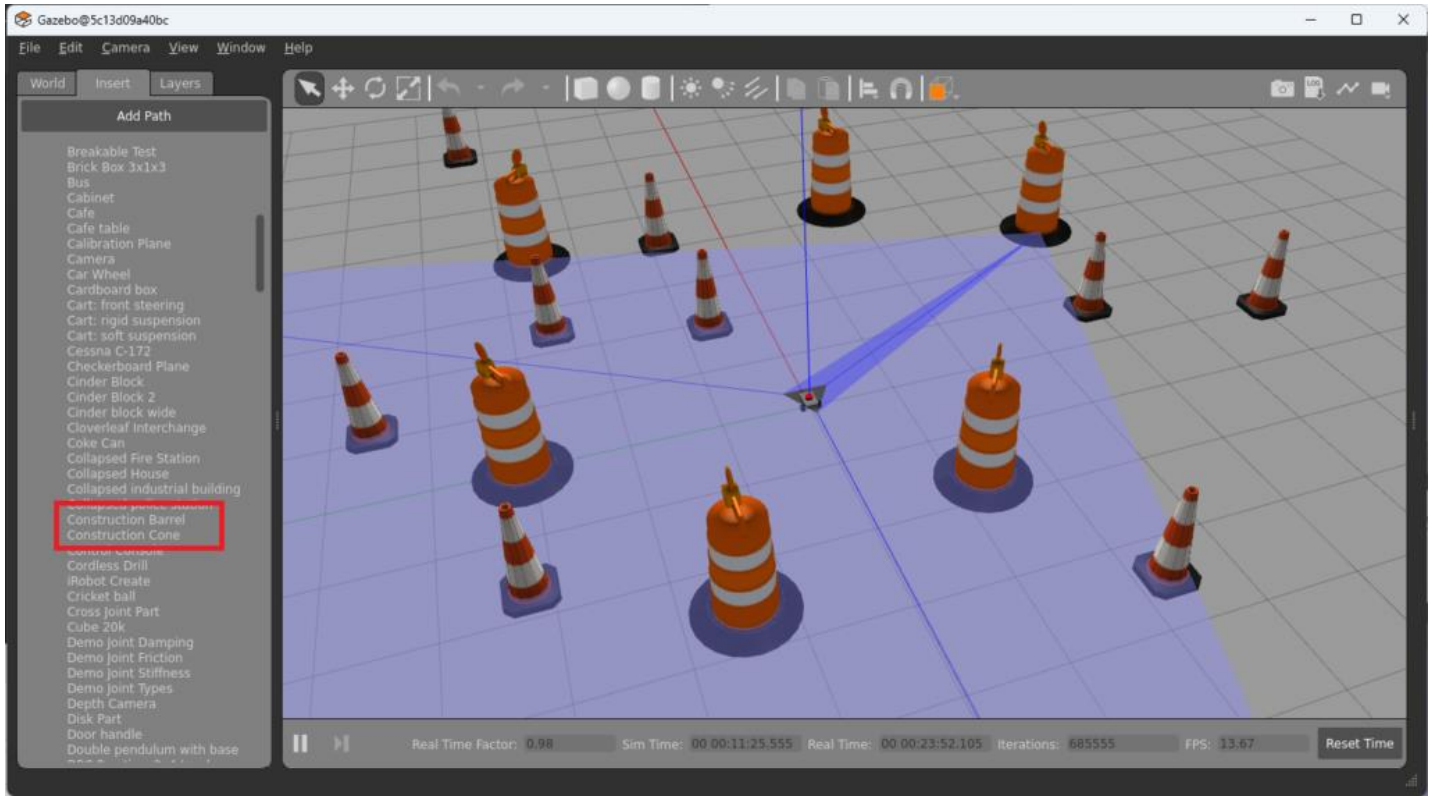
가제보 이해

시뮬레이션을 위한 월드 만들기, 로봇 제거 후 월드 저장



•Gazebo에서 'Insert' 탭을 누르고 'Insert model …'을 누른 상태로 5분 정도 기다리면 설치 가능한 모델 목록이 나타납니다.

가제보 이해

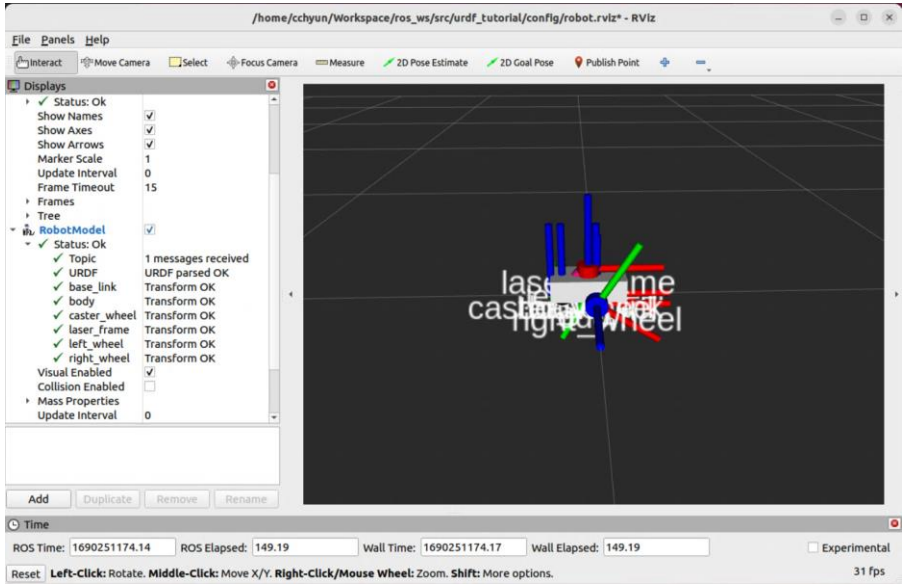


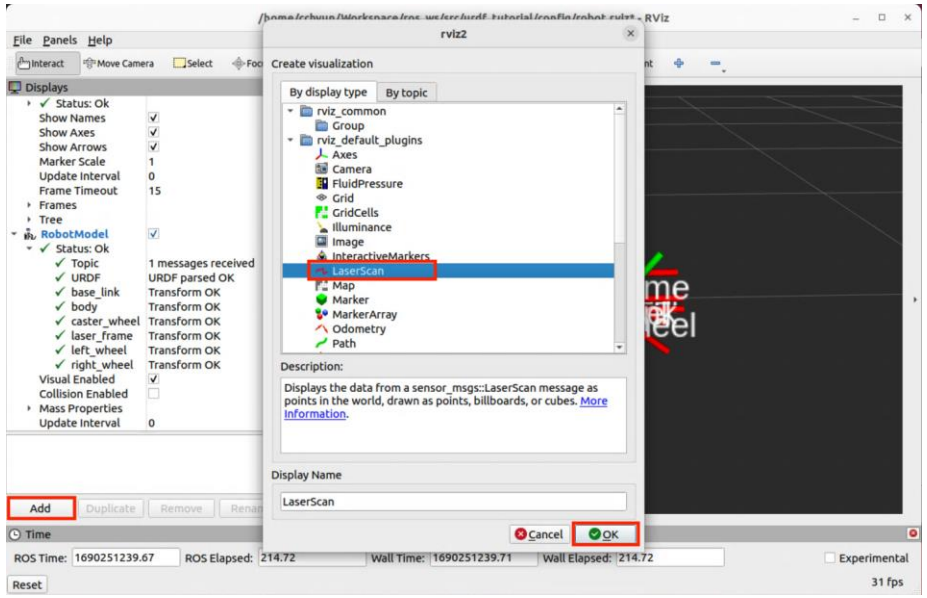
•목록에서 'with-bot'을 선택 후 마우스 메뉴 버튼을 누른 후 팝업메뉴에서 'Delete'를 눌러서 with-bot을 제거

•'src/urdf_tutorial/config' 폴더에 'with_robot.world'라는 파일 명으로 저장합니다.

RVIZ2 실행하기

```
$ ros2 launch urdf_tutorial lidar.launch.py world:=src/urdf_tutorial/config/with_robot.world
```



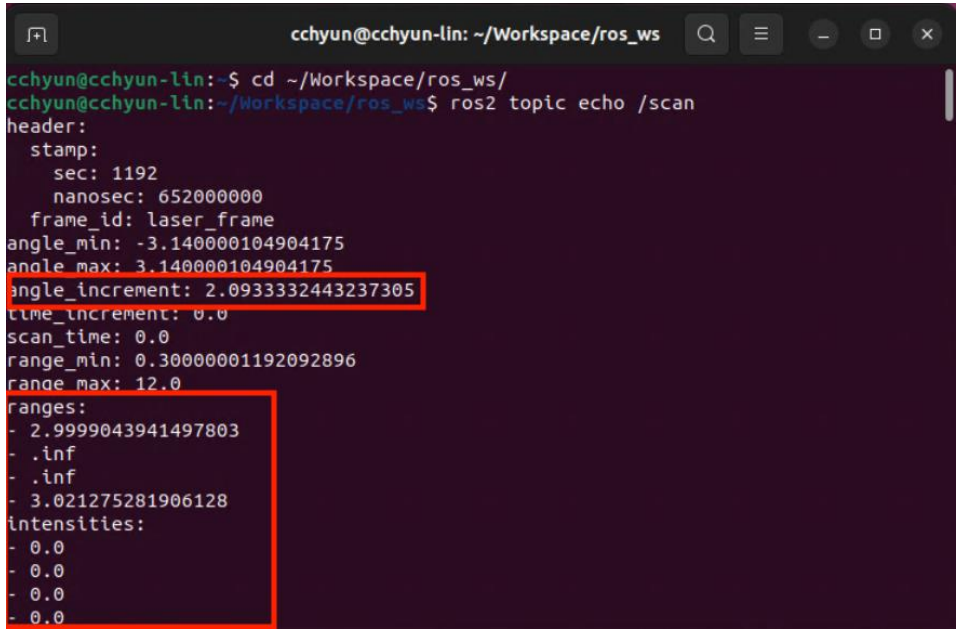


카메라 시뮬레이션

/scan 토픽 확인

- 2D LiDAR의 samples 수를 4에서 360으로 늘려보고 수신되는 정보를 기반으로 키보드로 주행을 해 보는 과정
- 우선 모든 터미널에서 'CTRL+C'를 눌러서 실행 중인 모든 프로그램을 종료.
- 'src/urdf_tutorial/urdf/lidar.xacro' 파일에서 'laser_frame'의 samples를 360으로 변경.

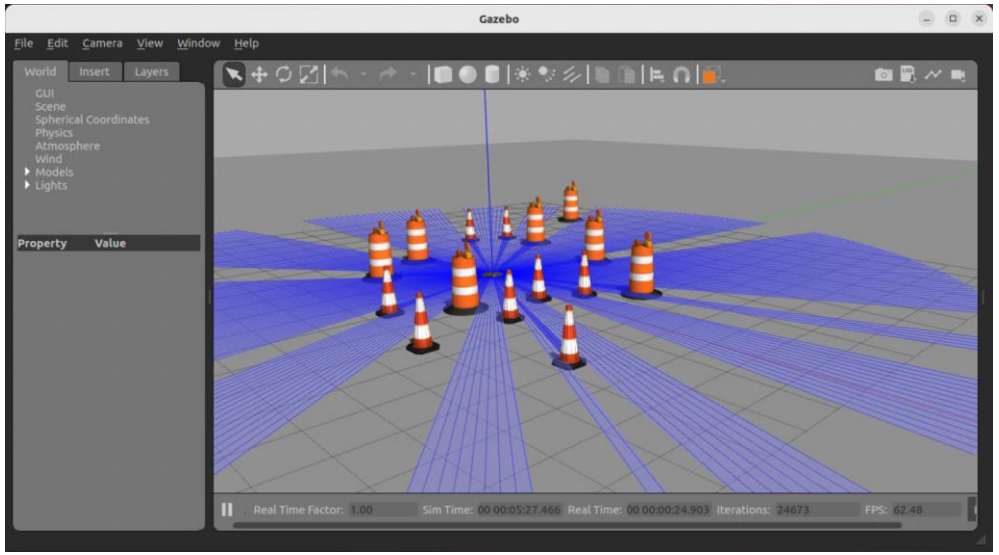
```
$ cd ~/Workspace/ros_ws/ $ ros2 topic echo /scan
```

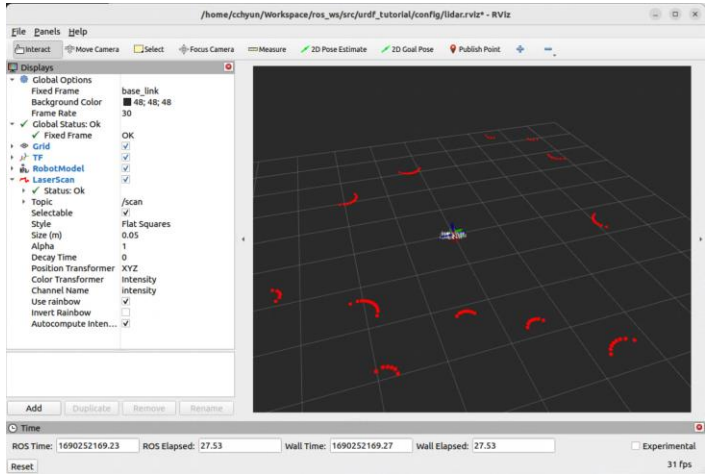


시뮬레이션 수행하기

- 2D LiDAR의 samples 수를 4에서 360으로 늘려보고 수신되는 정보를 기반으로 키보드로 주행을 해 보는 과정
- 우선 모든 터미널에서 'CTRL+C'를 눌러서 실행 중인 모든 프로그램을 종료합니다.
- 'src/urdf_tutorial/urdf/lidar.xacro' 파일에서 'laser_frame'의 samples를 360으로 변경합니다.

```
$ colcon build --symlink-install $ ros2 launch urdf_tutorial
lidar.launch.py world:=src/urdf_tutorial/config/with_robot.world
```






```
$ ros2 run teleop_twist_keyboard
teleop_twist_keyboard
```

실습과제 3- Moveit2

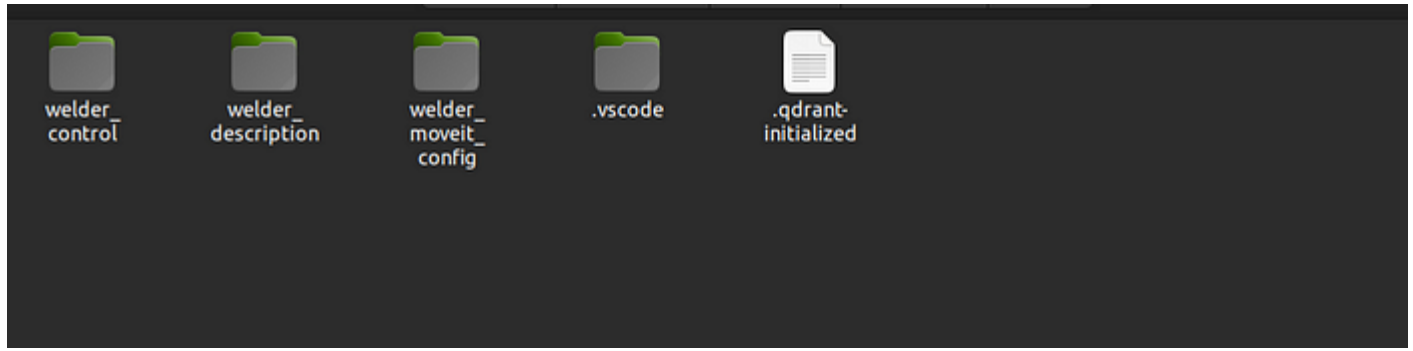
Moveit2 실습

시뮬레이션 및 시각화를 위한 Gazebo, 모션 계획을 위한 MoveIt2, 프로그래밍을 위한 Python을 통합하여 로봇 개발을 위한 포괄적인 환경을 제공
아래 링크는 내 GitHub에 있는 파일의 공개 저장소

필수 구성 요소
Ubuntu 22.04
ROS2 Humble
Gazebo
Moveit2

https://kolkemboi.medium.com/simulate-6-dof-robot-arm-in-ros2-gazebo-and-moveit2-a171c7e9b0ad

Moveit2 실습

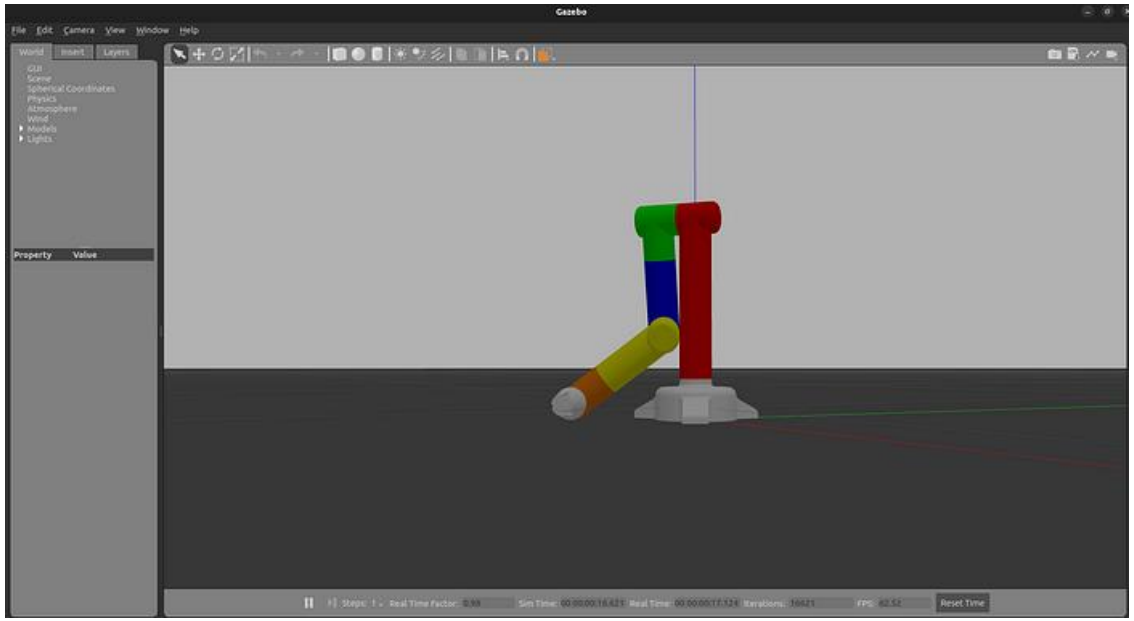


시스템 아키텍처
6개 DoF 로봇은 관절을 위한 6개 관절을 특징으로 하는 로봇 팔 구성을 사용합니다.
ROS 2 아키텍처 내에서 여러 노드를 사용하여 관절 컨트롤러, 센서 인터페이스, 모션 계획 모듈을 포함한 시스템의 다양한 측면을 제어합니다.
Gazebo는 시뮬레이션 환경 역할을 하며 현실적인 물리 및 센서 피드백을 제공합니다.
Autodesk Fusion을 사용하여 Gazebo에서 시뮬레이션
로봇 모델을 개발하고 ROS2용 URDF 내보내기 기능을 사용하여 URDF로 내보냈습니다.

Moveit2 실습

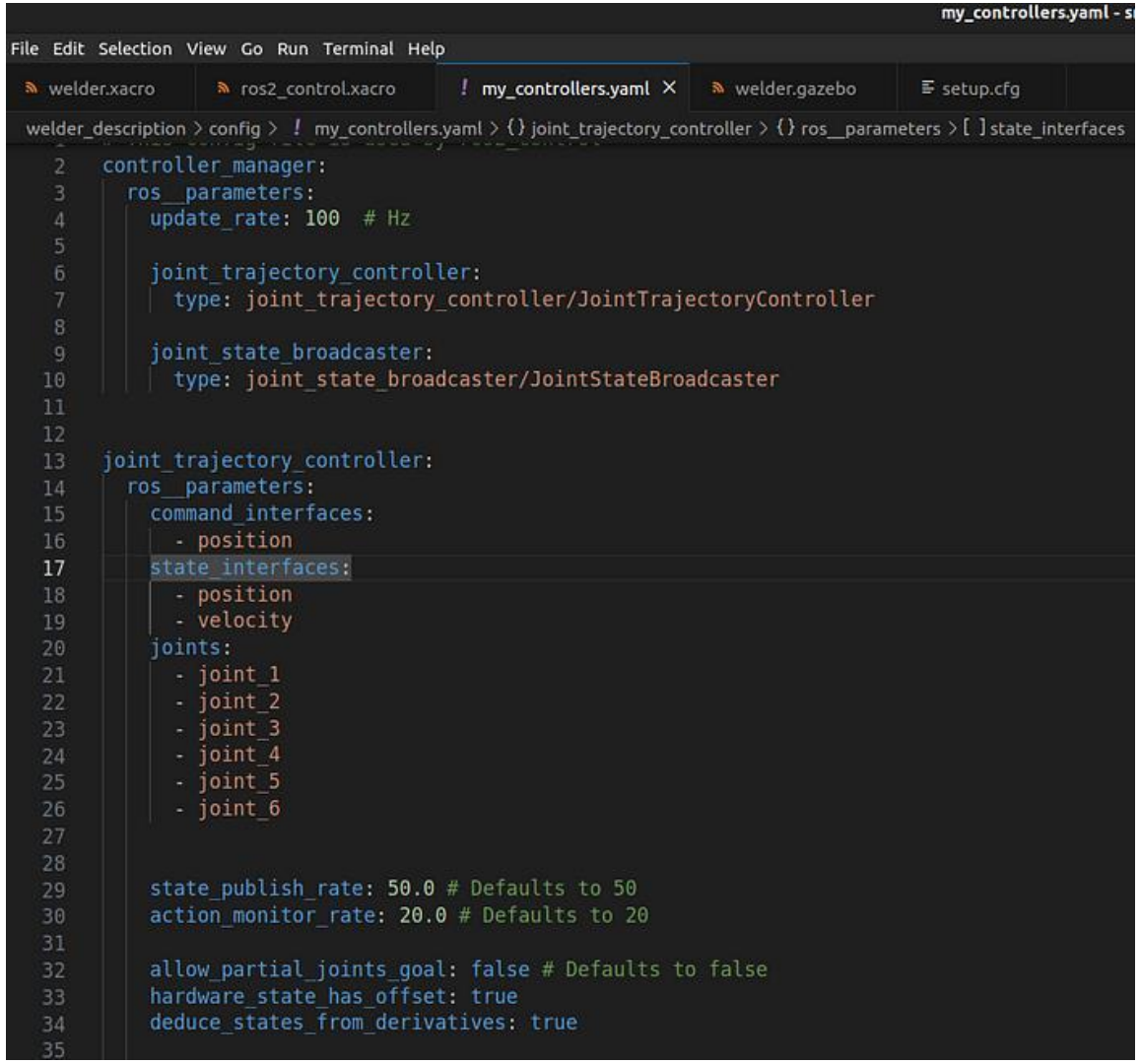
시뮬레이션을 위해 Gazebo로 가져왔습니다. 월드 링크를 추가하여 월드의 땅에 고정

로봇은 시뮬레이션된 세계에서 제어가 부족하므로 ros2_control을 사용하여 제어를 추가
조인트, 명령 및 상태 인터페이스를 정의하고 gazebo.launch.py에 컨트롤러 런처를 추가하여 모델을 생성하고 로봇의 컨트롤러를 시작



Moveit2 실습

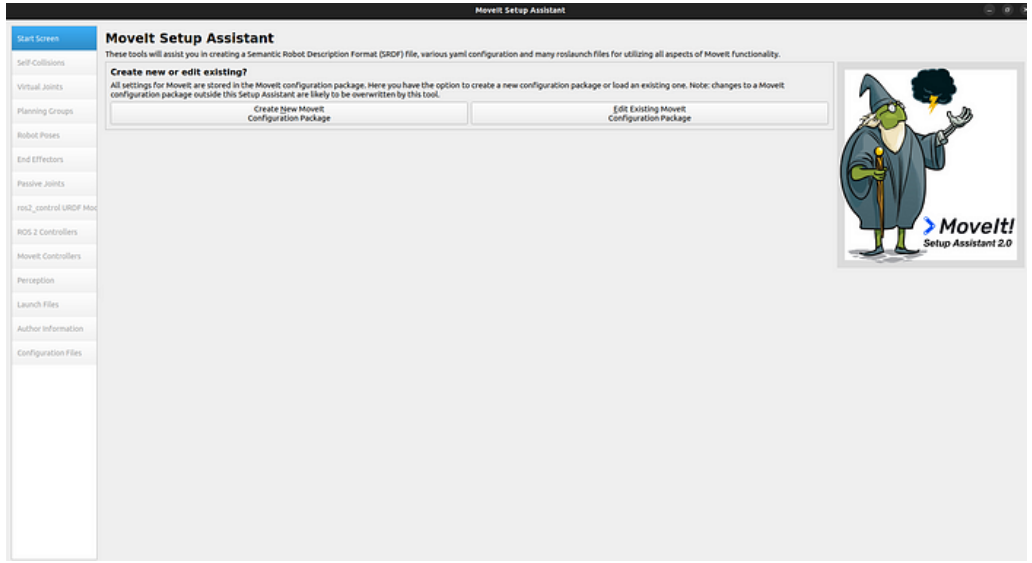

Activities
Visual Studio Code
File Edit Selection View Go Run Terminal Help
welder.xacro
ros2_control.xacro
my_controllers.yaml
welder.gazebo
welder_description > urdf > ros2_control.xacro
<?xml version="1.0" ?>
<robot name="welder" xmlns:xacro="http://www.ros.org/wiki/xacro" >
<ros2_control name="welder" type="system">
<hardware>
<plugin>gazebo_ros2_control/GazeboSystem</plugin>
</hardware>
<joint name="joint_1">
<command_interface name="position">
<param name="min">-3.14</param>
<param name="max">3.14</param>
</command_interface>
<state_interface name="position">
<param name="initial_value">0.0</param>
</state_interface>
<state_interface name="velocity"/>
<state_interface name="effort"/>
</joint>
<joint name="joint_2">
<command_interface name="position">
<param name="min">-3.14</param>
<param name="max">3.14</param>
</command_interface>
<state_interface name="position">
<param name="initial_value">0.0</param>
</state_interface>
<state_interface name="velocity"/>
<state_interface name="effort"/>
</joint>
<joint name="joint_3">
<command_interface name="position">
<param name="min">-3.14</param>
<param name="max">3.14</param>
</command_interface>
<state_interface name="position">
<param name="initial_value">0.0</param>
main
ROS2.humble
Git Graph


my_controllers.yaml - s
File Edit Selection View Go Run Terminal Help
welder.xacro
ros2_control.xacro
my_controllers.yaml
welder.gazebo
setup.cfg
welder_description > config > my_controllers.yaml > {} joint_trajectory_controller > {} ros__parameters > [ ] state_interfaces
controller_manager:
ros__parameters:
update_rate: 100 # Hz
joint_trajectory_controller:
type: joint_trajectory_controller/JointTrajectoryController
joint_state_broadcaster:
type: joint_state_broadcaster/JointStateBroadcaster
joint_trajectory_controller:
ros__parameters:
command_interfaces:
- position
state_interfaces:
- position
- velocity
joints:
- joint_1
- joint_2
- joint_3
- joint_4
- joint_5
- joint_6
state_publish_rate: 50.0 # Defaults to 50
action_monitor_rate: 20.0 # Defaults to 20
allow_partial_joints_goal: false # Defaults to false
hardware_state_has_offset: true
deduce_states_from_derivatives: true

Moveit2 실습

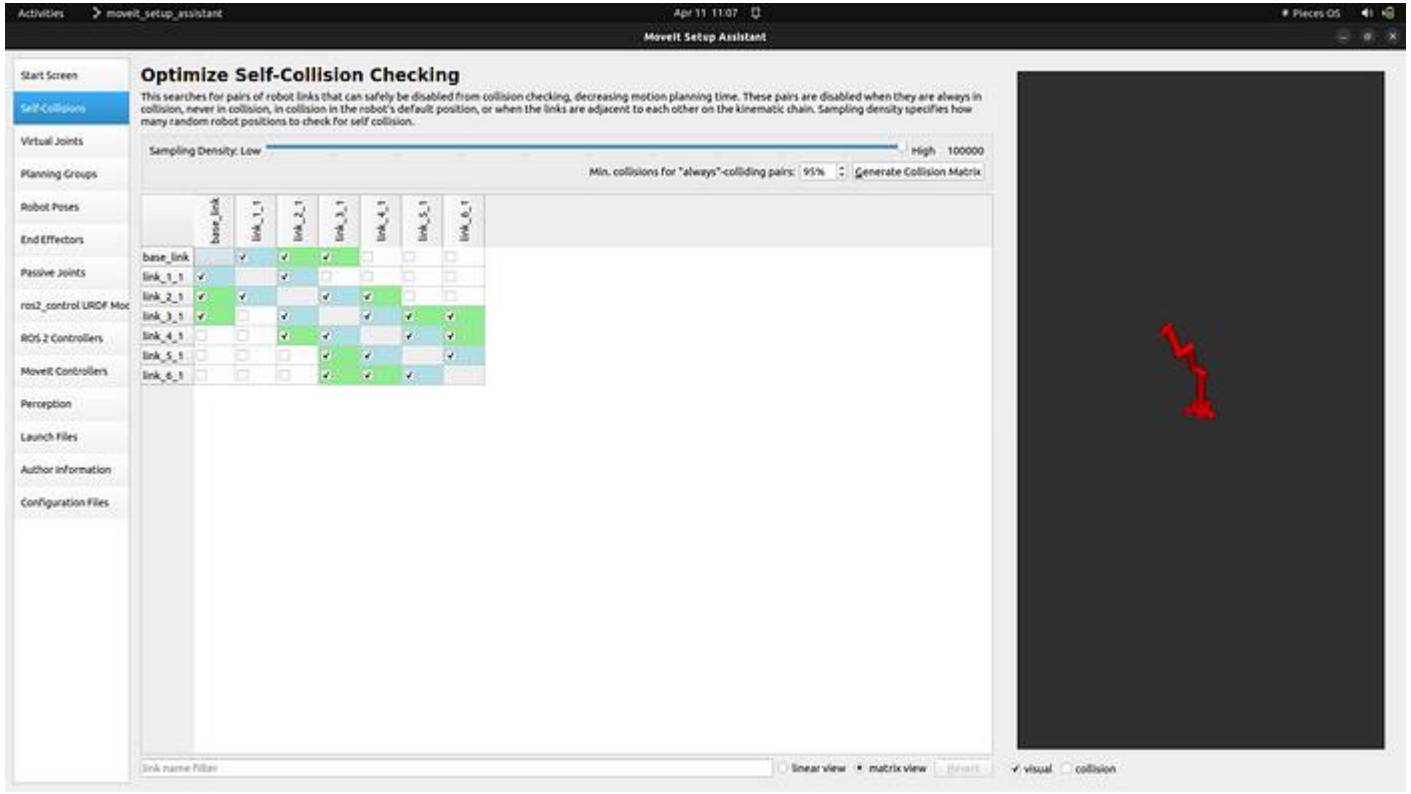
MoveIt2를 사용한 모션 플래닝

MoveIt2는 모션 플래닝 및 조작 작업을 위한 강력한 프레임워크를 제공
로봇의 운동학 모델은 MoveIt2와 인터페이스되어 구성 공간에서 경로 플래닝이 가능
Rapidly-exploring Random Trees(RRT)를 포함한 다양한 모션 플래닝 알고리즘이 구현되어 충돌 없는 궤적을 생성



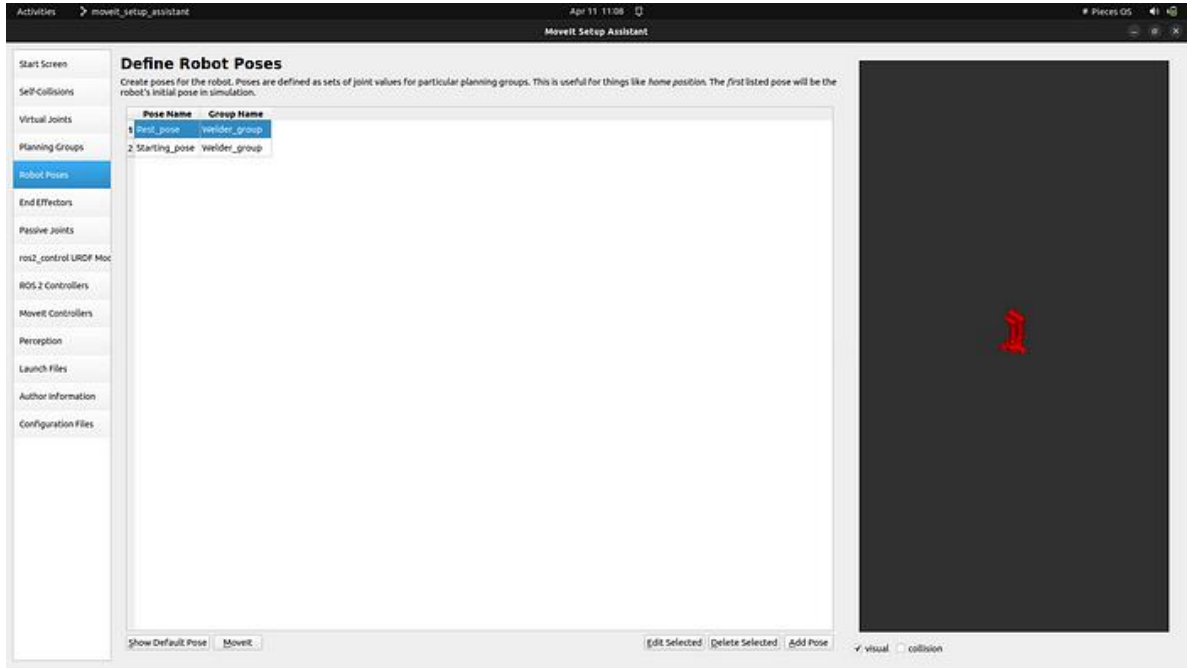
Moveit2 실습

시스템이 어떤 링크가 충돌하는지에 대한 계산을 줄이기 위해 충돌 행렬을 생성합니다.



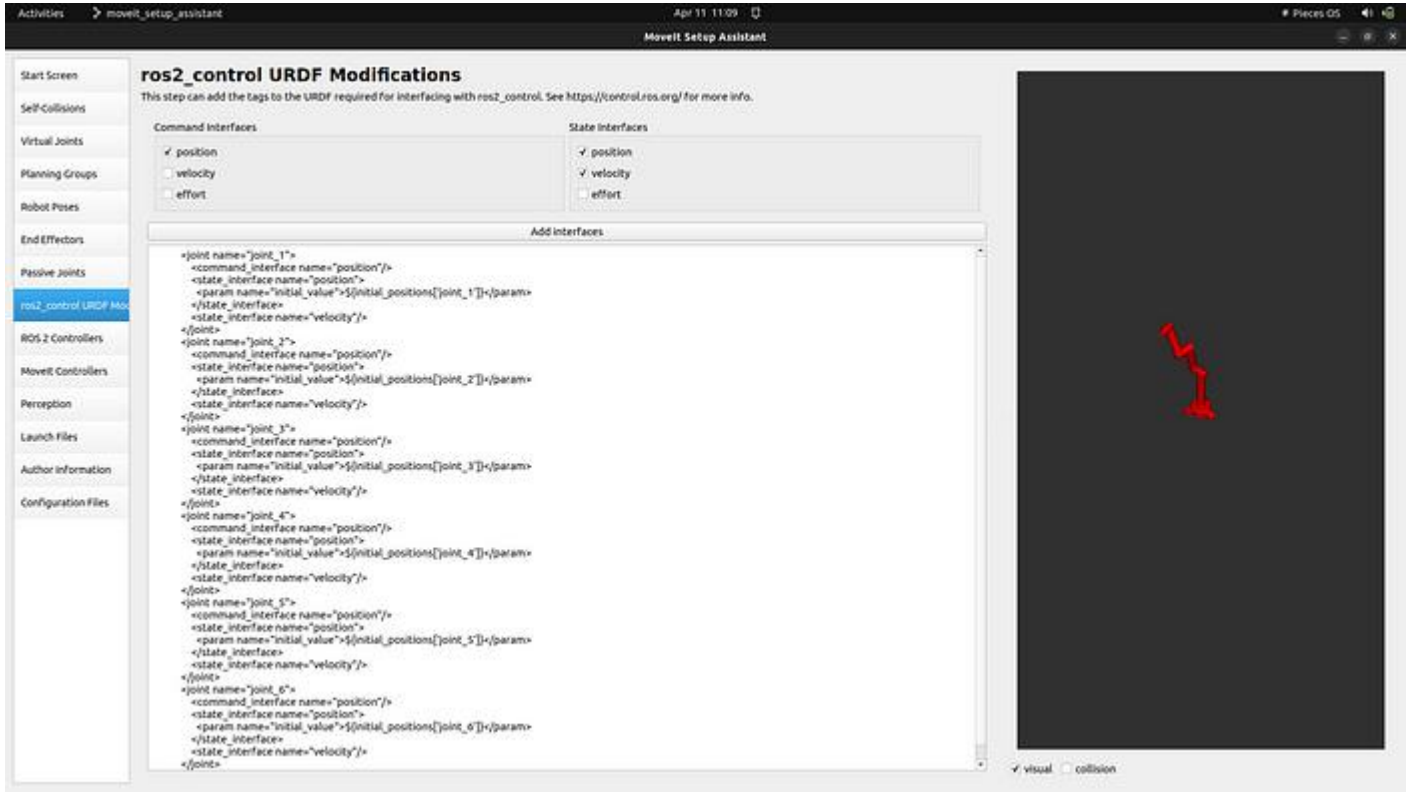
Moveit2 실습

RRT와 KDLkinematic 솔버를 사용하여 계획 그룹을 생성합니다.
이 단계에서는 Moveit에서 제어할 링크를 선택합니다.



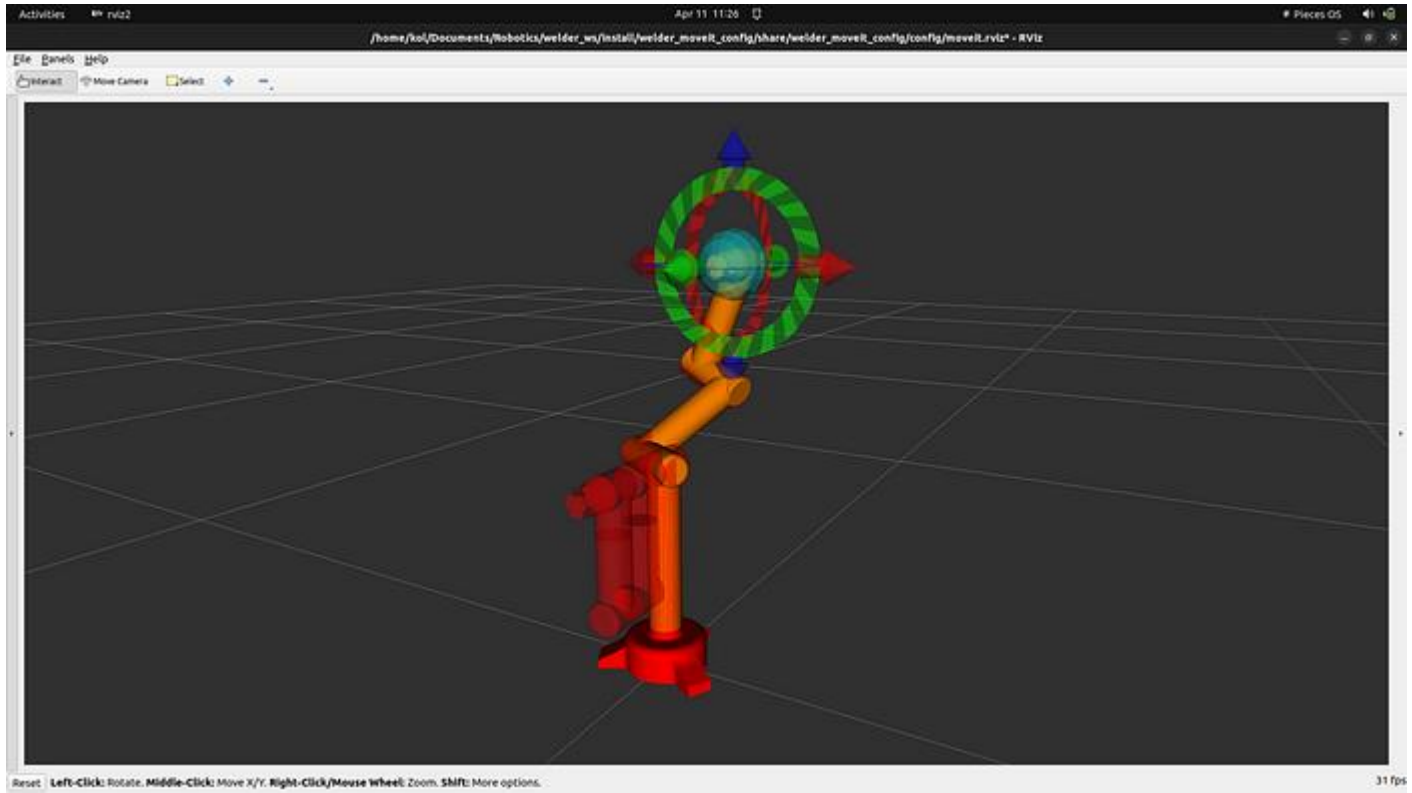
Moveit2 실습

로봇에 휴식 포즈와 시작 포즈의 두 가지 포즈를 제공합니다.



Moveit2 실습

moveit config 폴더에 ros2_control 인터페이스 추가



SRDF (Semantic Robot Description Format)

SRDF

- URDF와 같은 로봇의 의미론적 정보(semantic information)를 정의하는 XML 파일 포맷
- 주로 MoveIt!과 같은 로봇 모션 계획 소프트웨어에서 사용
- SRDF는 로봇의 움직임에 대한 의미론적 정보
  - 어떤 조인트를 그룹으로 묶을지, 플래닝 그룹, 엔드 이펙터 정의 등

SRDF 특징

- **플래닝 그룹** - 로봇의 특정 조인트들을 그룹화하여 모션 플래닝을 위해 사용하는데, 이를 SRDF에서 정의.
- **자유도 제약 및 제한** - 특정 조인트에 대한 운동 범위 제한 및 제약 조건을 정의.
- **MoveIt!와 통합** - 주로 MoveIt!에서 모션 플래닝을 위해 사용.

SDF (Simulation Description Format)

SDF

- **복잡한 시뮬레이션 환경 지원**:
  로봇뿐만 아니라 전체 시뮬레이션 환경(장애물, 지형 등) 정의

- **다양한 물리적 특성**:
  URDF보다 더 복잡한 물리 엔진 지원 및 모델 정의 가능
  (예: 다양한 마찰력, 관성 모멘트 등을 정교하게 설정)

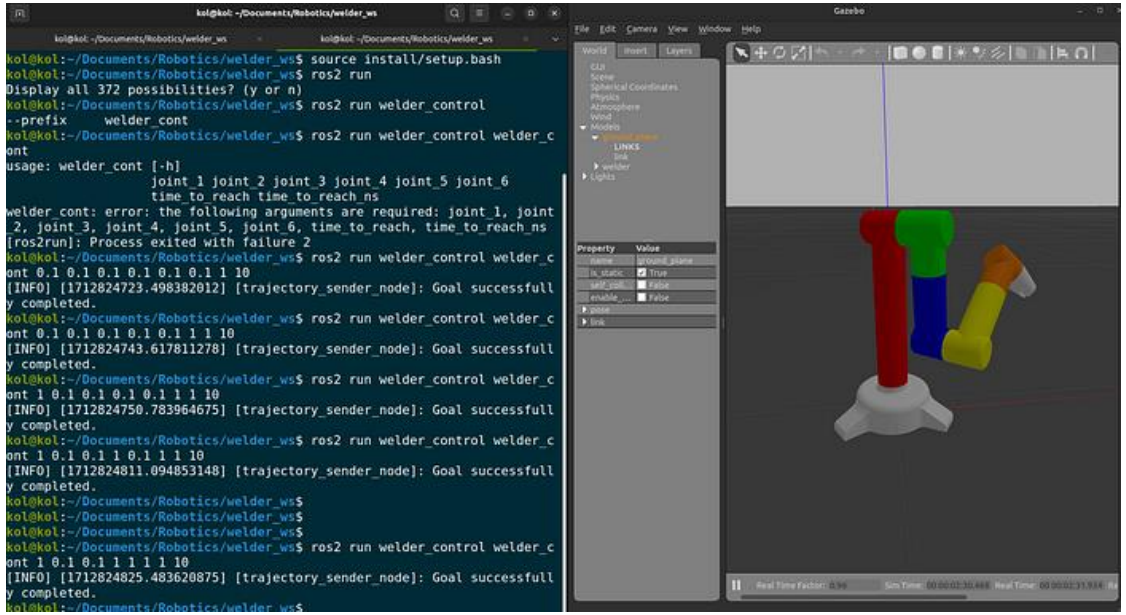
- **Gazebo에서 주로 사용**:
  Gazebo 시뮬레이터와 통합
  고급 물리 시뮬레이션을 위해 설계

```
<sdf version="1.6">
  <model name="example_robot">
    <link name="base_link">
      <visual>
        <geometry>
          <box size="1 1 1"/>
        </geometry>
      </visual>
      <collision>
        <geometry>
          <box size="1 1 1"/>
        </geometry>
      </collision>
    </link>
    <joint name="joint1" type="revolute">
      <parent>base_link</parent>
      <child>link1</child>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>
  </model>
</sdf>
```

Moveit2 실습

Python으로 프로그래밍하기

생성된 Moveit 파일을 RVIZ에서 시각화하고 로봇과 상호작용하며 다양한 위치에서 포즈
Python 스크립트는 ROS 2 환경 내에서 로봇의 동작을 제어하는 데 활용
궤적 생성, 역 운동학 계산 및 피드백 제어를 위한 함수는 Python으로 구현
이를 통해 프로그래밍의 유연성과 ROS 2 노드와의 쉬운 통합이 가능
스크립트는 각 관절의 관절 위치에 대한 6개 인수와 지속 시간에 대한 2개 인수를 포함하여 8개의 인수를 필요

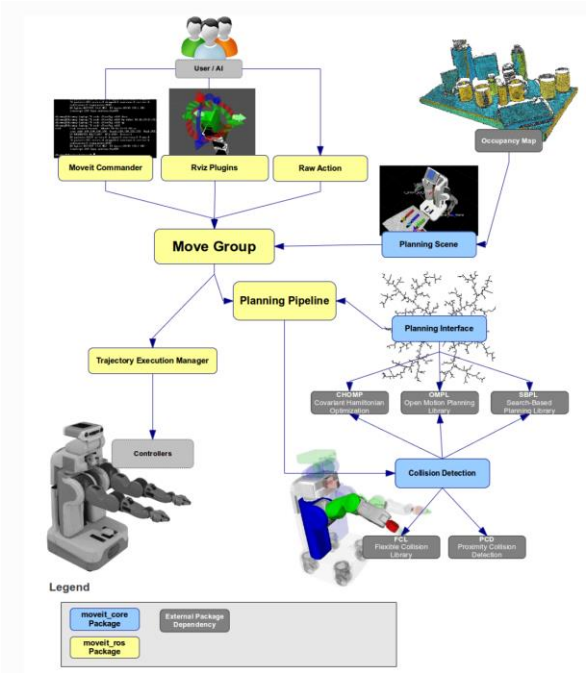


Moveit2

Moveit 시스템 구조

MoveIt은 로봇의 모션 플래닝, 제어, 시뮬레이션, 충돌 감지 등을 지원하는 강력한 로봇 소프트웨어 플랫폼으로, ROS(로봇 운영체제) 기반에서 동작

MoveIt은 모션 플래닝을 위해 다양한 알고리즘과 플러그인 기반 구조를 사용하며, 로봇의 경로 계획, 제어, 시각화 등의 작업을 통합하여 수행



- **로봇 모델 및 환경 (URDF, SRDF)**
  로봇의 구조와 환경을 정의하는 파일들을 바탕으로 MoveIt이 로봇과 환경을 이해하고 작업.
- **Planning Scene**
  로봇과 환경의 상태를 유지하고, 충돌 감지 및 환경 변화를 실시간 반영.
- **Motion Planning Pipeline**
  경로를 계획하고 최적화하며, 키네마틱 솔버와 충돌 감지 시스템을 통해 안전한 경로를 계산.
- **Controller Manager**
  경로를 실제 로봇에 적용하여 움직임을 제어.
- **Rviz 및 사용자 상호작용**
  사용자에게 로봇의 상태를 시각적으로 보여주고, 목표를 설정하거나 경로를 모니터링
- **Perception과 Grasping**
  환경 인식을 통해 동적으로 경로를 수정하고, 물체 조작.

Moveit 기능

1. Robot Model (로봇 모델)

- MoveIt은 로봇의 모델을 기반으로 작업을 수행.
- **URDF(Unified Robot Description Format)** 또는 **SRDF(Semantic Robot Description Format)** 정의.
- URDF는 로봇의 물리적 구조(링크, 조인트 등)를 정의, SRDF는 로봇의 키네마틱 체인, 그룹화된 링크, 제약 사항 등을 설명.
- 모션 플래닝, 경로 생성, 충돌 감지 등 다양한 작업에 사용.

2. Motion Planning (모션 플래닝)

- OMPL (Open Motion Planning Library)를 사용하여 다양한 알고리즘 기반으로 경로를 계획.
- 로봇의 목표 위치에 도달하는 최적 경로를 계산
- 이 과정에서 충돌을 피하고, 로봇의 운동학적 제약을 고려.
- 사용 가능한 플래닝 알고리즘에는 **RRT, RRT*, PRM, CHOMP, STOMP, TrajOpt** 등

Moveit 기능

3. Kinematics Solver (키네마틱 솔버)

- MoveIt은 정방향 키네마틱스 (Forward Kinematics)와 **역방향 키네마틱스 (Inverse Kinematics, IK)** 솔버를 사용 로봇의 위치와 자세 계산.
- 정방향 키네마틱스는 로봇의 조인트 값을 기반으로 각 링크의 위치를 계산, 역방향 키네마틱스는 목표 위치에 도달하기 위해 각 조인트의 값 계산.
- MoveIt은 IKFast, KDL 등 다양한 IK 솔버를 지원.

4. MoveIt Setup Assistant

- MoveIt을 설정하고 URDF 파일을 기반으로 **시스템 구성**을 쉽게 할 수 있도록 돕는 GUI 도구.
- 로봇 모델을 설정하고, 키네마틱 체인 및 그룹 설정, 플래닝 그룹 설정, 충돌 감지 설정 .
- 시뮬레이션 및 실제 로봇에서 사용할 수 있도록 MoveIt의 주요 구성 요소들을 자동으로 생성.

Moveit 기능

5. Motion Planning Pipeline (모션 플래닝 파이프라인)

- 모션 플래닝은 **파이프라인 구조**로 처리.
- 로봇의 현재 상태와 목표 상태를 입력받아, 플래닝 요청시 다양한 플래닝 알고리즘을 통해 경로 계산.
- 이 파이프라인에는 **경로 필터링**, **경로 수정**, **키네마틱 계산**, **충돌 감지** 등의 단계가 포함.
- 최종적으로 안전한 경로를 만들어 로봇이 실행할 수 있도록 명령 전달.

6. Controller Manager (컨트롤러 매니저)

- MoveIt은 플래닝된 경로를 로봇이 실제로 따를 수 있도록 **로봇 컨트롤러**와 통신.
- ROS 2의 **controller_manager**를 통해 제어 명령을 로봇의 하드웨어로 전송, 로봇이 경로 따라 동작시킴.
- 로봇의 조인트 트레저(Joint Trajectory Controller)와 같은 컨트롤러를 사용하여 경로 제어.

Moveit 기능

7. Planning Scene (플래닝 씬)

- 로봇과 환경을 정의하는 데이터 구조.
- 로봇 모델, 환경의 장애물, 그리고 이들이 상호작용하는 방식 등을 정의.
- 로봇의 현재 상태(포즈, 링크의 위치 등)와 환경의 상태를 포함하여 실시간으로 로봇의 상태를 추적하며, **충돌 감지** 및 **모션 플래닝**에 사용.

8. Collision Detection (충돌 감지)

- 로봇이 경로를 계획할 때 주변 환경 및 로봇 자신의 다른 링크와 충돌하지 않도록 확인하는 중요한 기능
- MoveIt은 FCL(Flexible Collision Library)와 **Bullet** 같은 충돌 감지 라이브러리를 사용.
- 경로 계획 중에 실시간으로 충돌 여부를 감지하고, 충돌을 피하는 경로 생성.

Moveit 기능

9. Perception (인식)

- 카메라 또는 LIDAR 등의 **센서 데이터**를 이용해 주변 환경을 인식하고, 이를 모션 플래닝 반영
- **Octomap**과 같은 맵핑 라이브러리를 통해 3D 환경에서 장애물을 감지하고, 로봇의 경로 계획에 반영.

10. Rviz와의 통합

- 사용자는 **Rviz**의 인터페이스를 통해 경로 계획과 시각화를 직관적으로 수행.
- move_group은 Rviz와 상호작용하여 실시간 경로 계획 및 상태 업데이트를 처리.

TF 이해

ROS에서 기본적으로 제공해주는 메세지 타입 중, 공간과 관련된 메세지 타입

geometry_msgs/PoseWithCovarianceStamped Message

File: geometry_msgs/PoseWithCovarianceStamped.msg

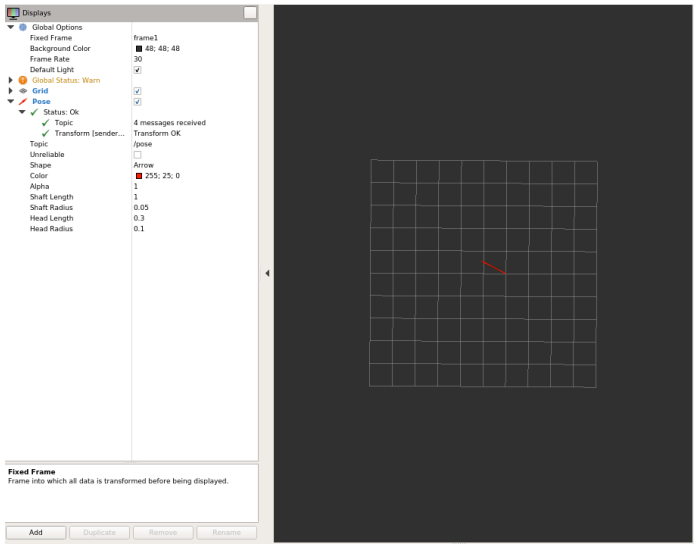
Raw Message Definition

```
# This expresses an estimated pose with a reference coordinate frame and timestamp

Header header
PoseWithCovariance pose
```

Compact Message Definition

```
std_msgs/Header header
geometry_msgs/PoseWithCovariance pose
```

autogenerated on Wed, 02 Mar 2022 00:06:53

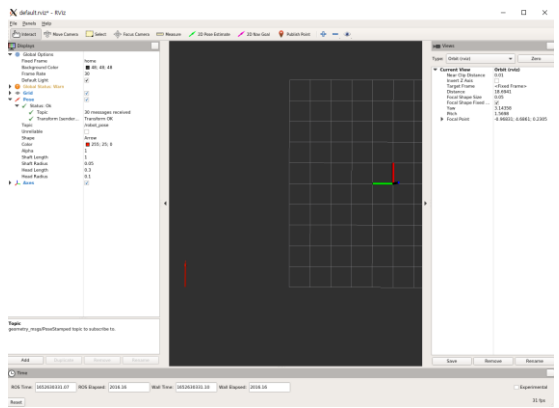


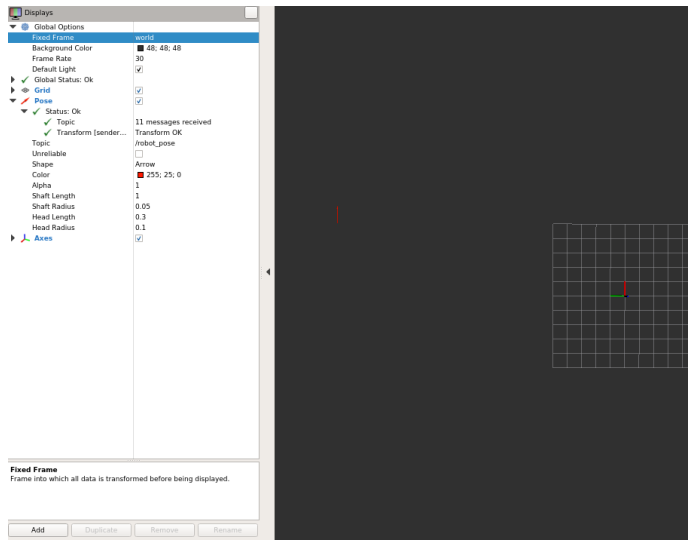

```
#!/usr/bin/python

import rospy
import random
import math
from tf.transformations import quaternion_from_euler
from geometry_msgs.msg import PoseStamped


if __name__ == '__main__':
    rospy.init_node("tf_test")

    pub = rospy.Publisher("pose", PoseStamped, queue_size=1)

    r = rospy.Rate(1)
    while not rospy.is_shutdown():

        msg = PoseStamped()

        msg.header.stamp = rospy.Time.now()
        msg.header.frame_id = "frame1"

        msg.pose.position.x = random.randint(0, 5)
        msg.pose.position.y = random.randint(0, 5)
        msg.pose.position.z = 0

        quat = quaternion_from_euler(
            0., 0., math.radians(random.randint(0, 180)))

        msg.pose.orientation.x = quat[0]
        msg.pose.orientation.y = quat[1]
        msg.pose.orientation.z = quat[2]
        msg.pose.orientation.w = quat[3]

        pub.publish(msg)

        r.sleep()
```

TF 이해

```
import rospy
import tf
from geometry_msgs.msg import PoseStamped

class Robot(object):
  def __init__(self):
    position = PoseStamped()

    position.header.frame_id = "home"

    position.pose.position.x = -5
    position.pose.position.y = 10

    position.pose.orientation.w = 1

    self.position = position
    self.pub = rospy.Publisher("robot_pose", PoseStamped, queue_size=1)

  def publishPose(self):
    self.position.header.stamp = rospy.Time.now()

    self.pub.publish(self.position)

if __name__ == '__main__':
  rospy.init_node("tf_pub")

  robot = Robot()

  r = rospy.Rate(1)
  while not rospy.is_shutdown():

    robot.publishPose()

    r.sleep()
```

```
tf_broadcaster =
tf.TransformBroadcaster()

  r = rospy.Rate(1)
  while not rospy.is_shutdown():

    tf_broadcaster.sendTransform(
      translation=[10, 10, 0],
      rotation=[0., 0., 0., 1],
      time=rospy.Time.now(),
      child="home",
      parent="world"
    )

    robot.publishPose()

    r.sleep()
```





터틀봇3 자기 위치 찾기

```
#include <ros/ros.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_listener.h>

class TurtlebotLocationTracker {
private:
   ros::NodeHandle nh;
   ros::Subscriber amcl_pose_sub;
   ros::Subscriber odom_sub;
   tf::TransformListener tf_listener;

   void amclPoseCallback(const
geometry_msgs::PoseWithCovarianceStamped::ConstPtr& msg) {
      ROS_INFO("AMCL Position:");
      ROS_INFO("  Position: (x: %.2f, y: %.2f)",
            msg->pose.pose.position.x,
            msg->pose.pose.position.y);

      // 쿼터니언을 RPY로 변환
      tf::Quaternion q(
         msg->pose.pose.orientation.x,
         msg->pose.pose.orientation.y,
         msg->pose.pose.orientation.z,
         msg->pose.pose.orientation.w
      );
      tf::Matrix3x3 m(q);
      double roll, pitch, yaw;
      m.getRPY(roll, pitch, yaw);

      ROS_INFO("  Orientation (yaw): %.2f degrees", yaw * 180.0 / M_PI);
   }
```

```
 void odometryCallback(const
nav_msgs::Odometry::ConstPtr& msg) {
      ROS_INFO("Odometry Position:");
      ROS_INFO("  Position: (x: %.2f, y: %.2f, z: %.2f)",
            msg->pose.pose.position.x,
            msg->pose.pose.position.y,
            msg->pose.pose.position.z);

      // 쿼터니언을 RPY로 변환
      tf::Quaternion q(
         msg->pose.pose.orientation.x,
         msg->pose.pose.orientation.y,
         msg->pose.pose.orientation.z,
         msg->pose.pose.orientation.w
      );
      tf::Matrix3x3 m(q);
      double roll, pitch, yaw;
      m.getRPY(roll, pitch, yaw);

      ROS_INFO("  Orientation:");
      ROS_INFO("    Roll: %.2f degrees", roll * 180.0 / M_PI);
      ROS_INFO("    Pitch: %.2f degrees", pitch * 180.0 / M_PI);
      ROS_INFO("    Yaw: %.2f degrees", yaw * 180.0 / M_PI);
   }
```

```
   void getTFTransform() {
      try {
         tf::StampedTransform transform;
         tf_listener.lookupTransform("/map", "/base_footprint",
ros::Time(0), transform);

         ROS_INFO("TF Transform:");
         ROS_INFO("  Position: (x: %.2f, y: %.2f, z: %.2f)",
               transform.getOrigin().x(),
               transform.getOrigin().y(),
               transform.getOrigin().z());

         // 쿼터니언을 RPY로 변환
         tf::Quaternion q = transform.getRotation();
         tf::Matrix3x3 m(q);
         double roll, pitch, yaw;
         m.getRPY(roll, pitch, yaw);

         ROS_INFO("  Orientation:");
         ROS_INFO("    Roll: %.2f degrees", roll * 180.0 / M_PI);
         ROS_INFO("    Pitch: %.2f degrees", pitch * 180.0 / M_PI);
         ROS_INFO("    Yaw: %.2f degrees", yaw * 180.0 / M_PI);

      } catch (tf::TransformException &ex) {
         ROS_ERROR("TF Transform error: %s", ex.what());
      }
   }
```

터틀봇3 자기 위치 찾기

```
public:
  TurtlebotLocationTracker() {
    // AMCL 포즈 구독
    amcl_pose_sub = nh.subscribe("/amcl_pose", 10,
&TurtlebotLocationTracker::amclPoseCallback, this);

    // 오도메트리 구독
    odom_sub = nh.subscribe("/odom", 10,
&TurtlebotLocationTracker::odometryCallback, this);
  }

  void run() {
    ros::Rate rate(1.0);  // 1Hz로 위치 업데이트
    while (ros::ok()) {
      ros::spinOnce();

      // TF 변환 확인
      getTFTransform();

      rate.sleep();
    }
  }
};
```

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "turtlebot3_location_tracker");

  TurtlebotLocationTracker tracker;
  tracker.run();

  return 0;
}
```

FT로 서로 위치 추종

두 개의 터틀봇이 서로를 따라가도록 한다.

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <turtlesim/Pose.h>
#include <tf/transform_broadcaster.h>
#include <tf/transform_listener.h>
#include <cmath>
class TurtleFollower {
private:
  ros::NodeHandle nh;
  ros::Publisher turtle1_vel_pub;
  ros::Publisher turtle2_vel_pub;
  ros::Subscriber turtle1_pose_sub;
  ros::Subscriber turtle2_pose_sub;
  tf::TransformBroadcaster br;
  tf::TransformListener listener;
  turtlesim::Pose turtle1_pose;
  turtlesim::Pose turtle2_pose;

  void turtle1PoseCallback(const turtlesim::Pose::ConstPtr& msg) {
    turtle1_pose = *msg;

    // Broadcast turtle1's transform
    tf::Transform transform;
    transform.setOrigin(tf::Vector3(msg->x, msg->y, 0.0));
    tf::Quaternion q;
    q.setRPY(0, 0, msg->theta);
    transform.setRotation(q);
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle1"));
  }

  void turtle2PoseCallback(const turtlesim::Pose::ConstPtr& msg) {
    turtle2_pose = *msg;

    // Broadcast turtle2's transform
    tf::Transform transform;
    transform.setOrigin(tf::Vector3(msg->x, msg->y, 0.0));
    tf::Quaternion q;
    q.setRPY(0, 0, msg->theta);
    transform.setRotation(q);
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle2"));
  }

  double getDistance(double x1, double y1, double x2, double y2) {
    return std::sqrt(std::pow(x1 - x2, 2) + std::pow(y1 - y2, 2));
  }

  double getAngleBetweenPoses(const turtlesim::Pose& pose1, const turtlesim::Pose& pose2) {
    return std::atan2(pose2.y - pose1.y, pose2.x - pose1.x);
  }

public:
  TurtleFollower() {
    // Subscribers for pose
    turtle1_pose_sub = nh.subscribe("/turtle1/pose", 10, &TurtleFollower::turtle1PoseCallback, this);
    turtle2_pose_sub = nh.subscribe("/turtle2/pose", 10, &TurtleFollower::turtle2PoseCallback, this);

    // Publishers for velocity
    turtle1_vel_pub = nh.advertise<geometry_msgs::Twist>("/turtle1/cmd_vel", 10);
    turtle2_vel_pub = nh.advertise<geometry_msgs::Twist>("/turtle2/cmd_vel", 10);
  }
```

FT로 서로 위치 추종

```
void update() {
    // Turtle2 follows Turtle1
    if (turtle1_pose.x != 0 && turtle2_pose.x != 0) {
      double distance = getDistance(turtle1_pose.x, turtle1_pose.y,
                                    turtle2_pose.x, turtle2_pose.y);
      double target_angle = getAngleBetweenPoses(turtle2_pose, turtle1_pose);

      geometry_msgs::Twist cmd_vel;

      // Linear velocity proportional to distance
      cmd_vel.linear.x = distance * 0.5;

      // Angular velocity to align with target
      double angle_diff = target_angle - turtle2_pose.theta;
      // Normalize angle
      while (angle_diff > M_PI) angle_diff -= 2 * M_PI;
      while (angle_diff < -M_PI) angle_diff += 2 * M_PI;

      cmd_vel.angular.z = angle_diff * 1.0;

      // Limit velocities
      cmd_vel.linear.x = std::min(cmd_vel.linear.x, 1.0);
      cmd_vel.angular.z = std::min(std::max(cmd_vel.angular.z, -1.5), 1.5);

      turtle2_vel_pub.publish(cmd_vel);
    }
  }
};
```

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "turtle_follower");

  TurtleFollower follower;

  ros::Rate rate(10);
  while (ros::ok()) {
    ros::spinOnce();
    follower.update();
    rate.sleep();
  }

  return 0;
}
```

수고하셨습니다.