ROS2 – Unity 연동 Simulation

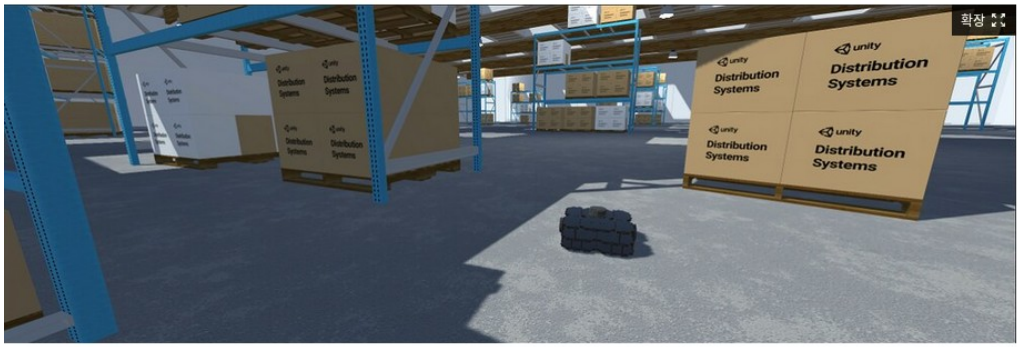
https://blog.unity.com/kr/manufacturing/advance-your-robot-autonomy-with-ros-2-and-unity

Unity 블로그

ROS2와 Unity로 로봇 자율성 향상

작성자: Devin Miller, Sarah Gibson, Anthony Navarro

2021년 8월 13일 산업 분야 | 10 분 소요



유니티에서 기쁜 마음으로 ROS2에 대한 공식 지원을 발표합니다. 강력한 프레임워크와 시뮬레이션을 갖춘 ROS2는 앞으로 다양하고 새로운 사용 사례를 지원할 예정입니다.

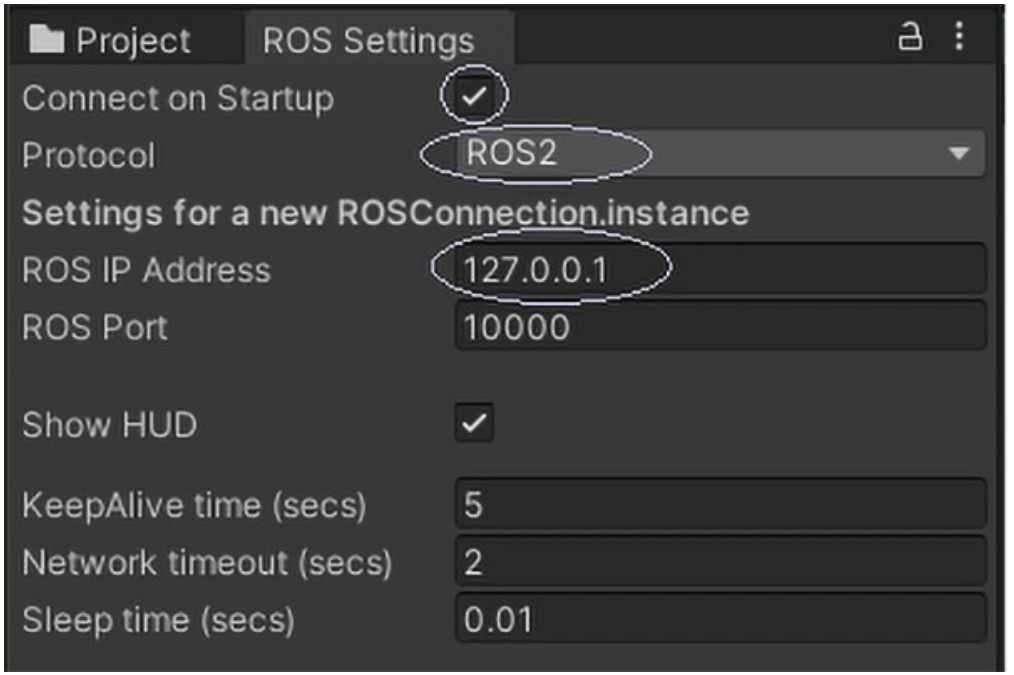
2007년부터 사용되기 시작한 ROS(Robot Operating System, 로봇 운영 체제)는 인기 있는 로봇 애플리케이션 개발 프레임워크입니다. 원래 로보틱스 연구를 촉진할 목적으로 설계되었으나, 곧 산업 및 상업용 로보틱스 분야에서 널리 사용되게 되었습니다. ROS2는 ROS의 안정적인 프레임워크를 기반으로 구축되었으며 멀티 로봇 시스템, 실시간 시스템, 제작 환경과 같은 현대적인 응용 분야에 대한 지원을 개선했습니다. 유니티는 ROS 생태계에 대한 공식 지원을 ROS2까지 확장하려 합니다.

최근의 로보틱스는 개발자가 직접 정의한 엄격한 규칙 없이 알고리즘이 결정을 내리는 "자율성" 연구 및 개발로 초점이 옮겨가고 있으며, 개발 과정은 실제 세계의 테스트보다 유연하고 빠른 실험이 가능한 시뮬레이션을 통해 지원됩니다. 유니티는 Robotics-Nav2-SLAM 예제를 개발하여, Unity와 ROS2로 자율 주행 로보틱스(AMR)에 동시 위치 인식 및 매핑(SLAM)과 내비게이션을 시뮬레이션하는 방법을 시연했습니다.

ROS2 지원

클릭 한 번으로 ROS2 지원

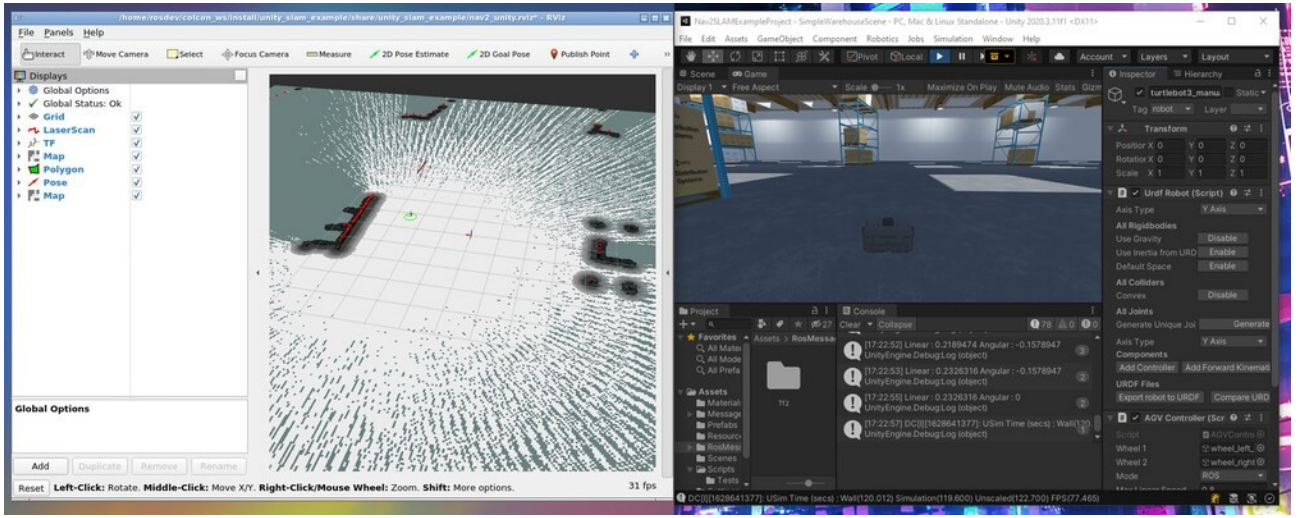
Unity 프로젝트를 ROS2로 간편하게 전환할 수 있습니다. ROS-TCP-Connector패키지에 ROS와 ROS2 통합 간에 패키지를 전환할 수 있는 드롭다운 메뉴가 추가되었습니다. 프로토콜을 변경하면 Unity에서 자동으로 선택된 메시지 정의와 직렬화 프로토콜에 따라 패키지를 다시 컴파일합니다. 해당 기능을 테스트하려면 프로젝트에 변경 사항을 직접 적용하거나 Robotics-Nav2-SLAM 예제 저장소를 이용해 보세요. Nav2 내비게이팅 및 매핑 튜토리얼에서 Unity를 센서 및 주행 거리 측정 정보의 시뮬레이션 소스로 사용하는 데 필요한 컴포넌트가 포함되어 있습니다.



예제 프로젝트

이 예제 프로젝트는 Unity를 사용하여 ROS2로 구동되는 내비게이션 시스템을 시뮬레이션하는 방법을 시연합니다. 내비게이션의 개념은 간단명료하며 자율 로보틱스의 맥락에서도 크게 달라지지 않습니다. 내비게이션 알고리즘의 목표는 *현 위치*에서 *원하는 위치*로 이동하는 경로를 찾는 것입니다. 하지만 현 위치에서 원하는 위치로 이동하려면 우선 SLAM, 즉 동시 위치 인식과 매핑을 수행해야 합니다. SLAM은 로봇의 현재 위치와 이전 위치를 파악하도록 구축된 알고리즘의 모음을 설명합니다. 인간은 감각 기관과 두뇌를 이어 주는 고유의 처리 과정을 통해 끊임없이 SLAM을 수행합니다. 하지만 자율 로봇의 경우 대부분의 실제 환경에서 SLAM을 정확히 수행하는 일이 여전히 어려운 과제입니다. 자율 주행 로봇이 이전 위치에 근거하여 현 위치를 항상 파악할 수 있도록 하기 위해 정확히 무엇이 필요한지 알아내기 위한 연구가 여전히 활발히 진행되고 있습니다. 해답을 찾는 유일한 방법은 각 사용 사례에서 센서, 알고리즘 등 다양한 방안을 시도하고 그중 어떤 방안이 효과적인지 살펴보는 것입니다.

유니티 예제 프로젝트에는 간단한 창고 환경과 완전히 구현된 Turtlebot3 주행 로봇, 시뮬레이션 LIDAR 및 모터 컨트롤러, 그리고 Nav2 및 slam_toolbox 스택을 시뮬레이션에서 연습시키는 데 필요한 ROS2 종속성이 모두 포함된 이미지를 구축하는 Docker 파일이 포함되어 있습니다. Nav2의 단계적 튜토리얼은 ROS2 또는 SLAM 알고리즘으로 작업해 본 적이 없는 경우에 유용한 컨텍스트를 제공합니다. Unity에서 예제 프로젝트를 실행해 보고 싶다면, 프로젝트를 시작하고 실행하는 데 필요한 모든 지침이 저장소에 있습니다.



왼쪽 : Unity 에서 생성하여 전송한 ROS2 메시지가 표시된 RViz 창 오른쪽 : Unity 에서 SLAM 및 자율 내비게이션을 수행하는 TurtleBot3

Navigation 2 SLAM Example

Setup Instructions

1. Configuring Your Development Environment
2. Setting Up the Unity Project
3. Running the Example
4. Visualizing with Unity
5. Making a Custom Visualizer

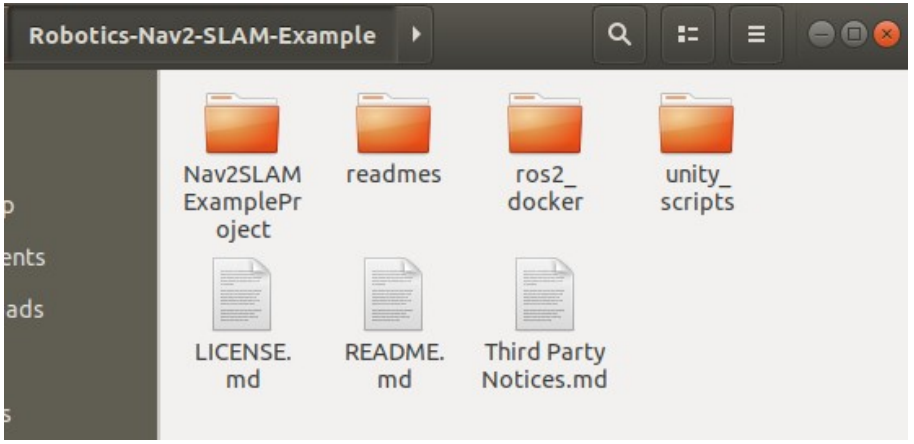
Understanding the Project Components

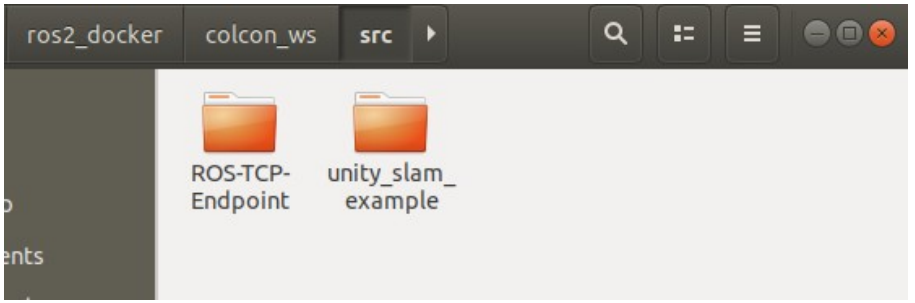
- Breakdown of this Example
- Robotics Warehouse
- ROS TCP Connector
- Visualizations
- URDF Importer

Setup Environment

Clone the Project

git clone --recurse-submodule git@github.com:Unity-Technologies/Robotics-Nav2-SLAM-Example.git





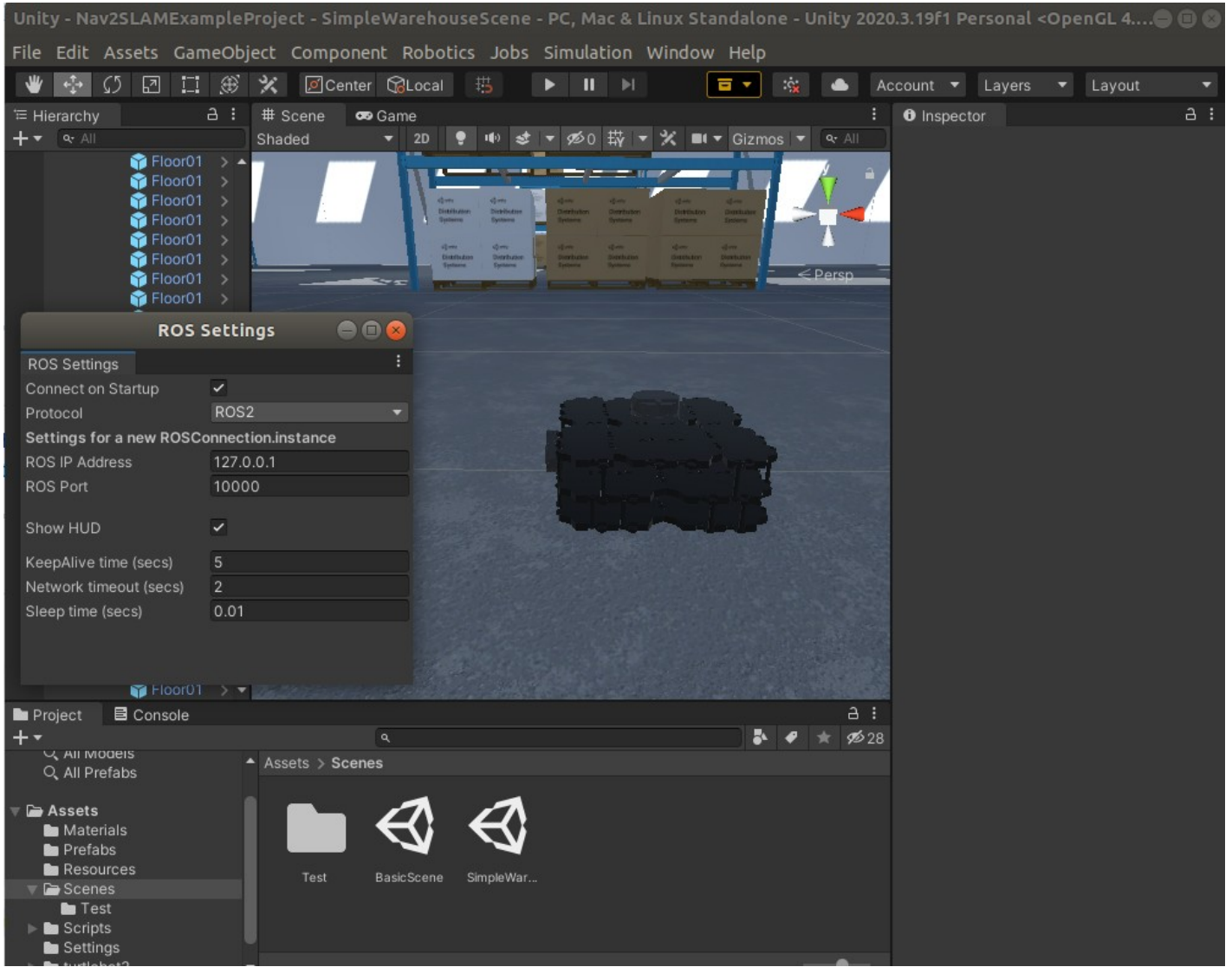
Build the Docker container (ROS2 Environment 설치)

cd ros2_docker
docker build -t unity-robotics:nav2-slam-example ./

```
lch@lch-TB250-BTC:~/Robotics-Nav2-SLAM-Example/ros2_docker$ ll
total 20
drwxrwxr-x 3 lch lch 4096 Oct  8 22:37 ./
drwxrwxr-x 9 lch lch 4096 Oct  8 21:26 ../
drwxrwxr-x 3 lch lch 4096 Oct  8 21:26 colcon_ws/
-rw-rw-r-- 1 lch lch 3186 Oct  8 21:26 Dockerfile
-rwxrwxr-x 1 lch lch  298 Oct  8 21:26 ros2-setup.bash*
```

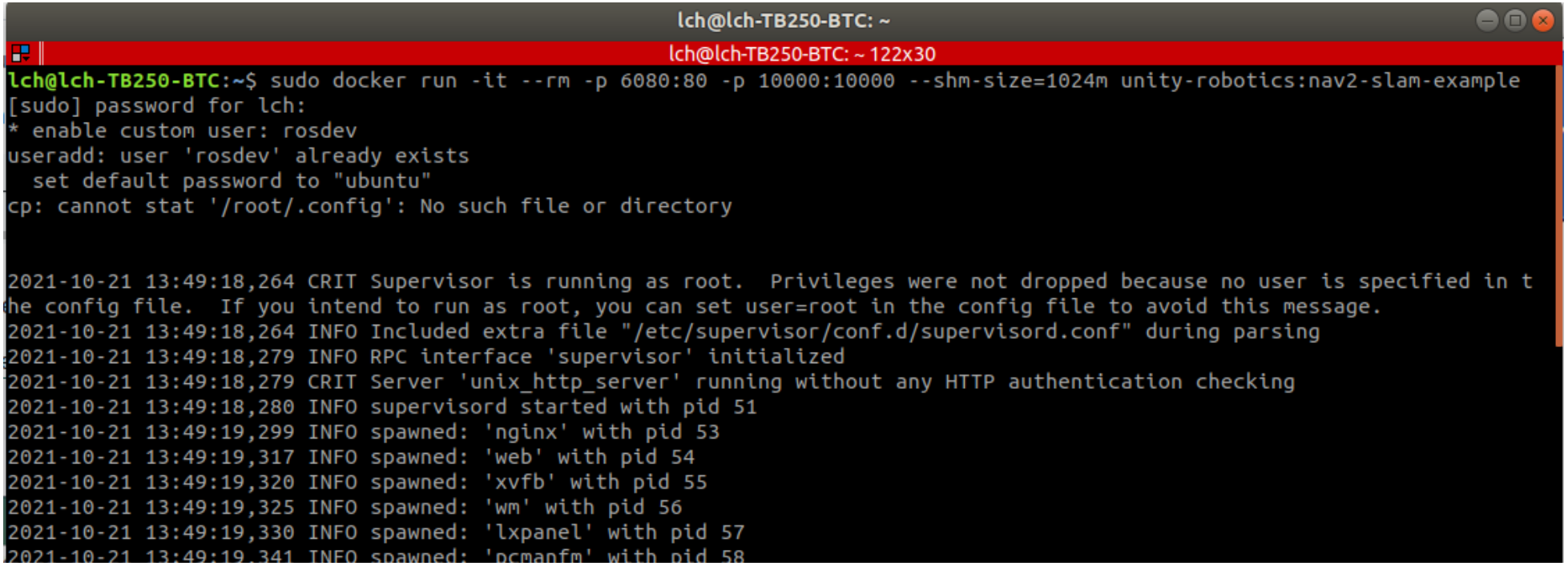
Setup the Unity Project

Open the Project



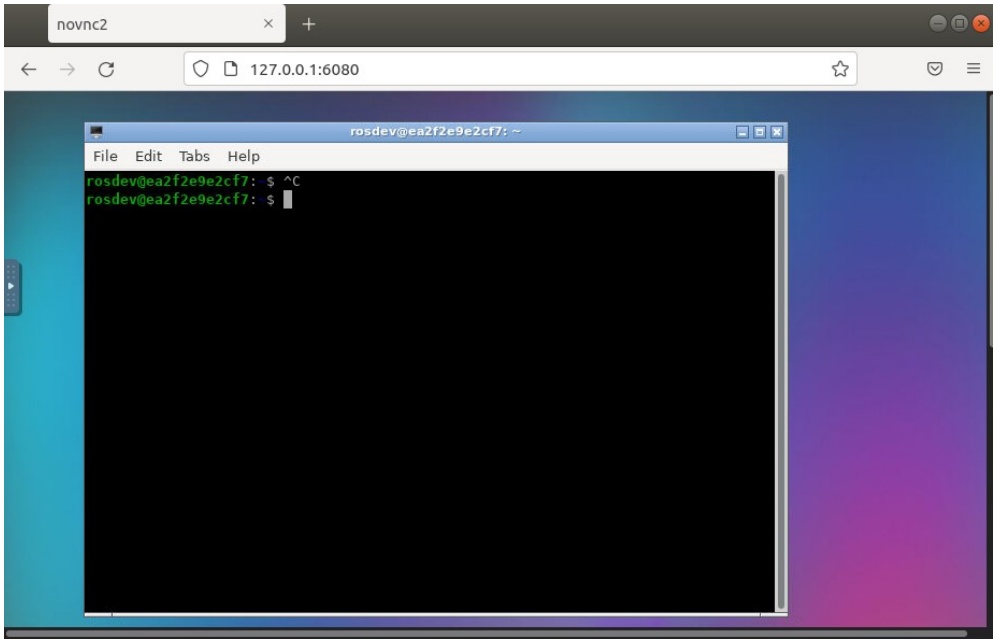
Running the Example

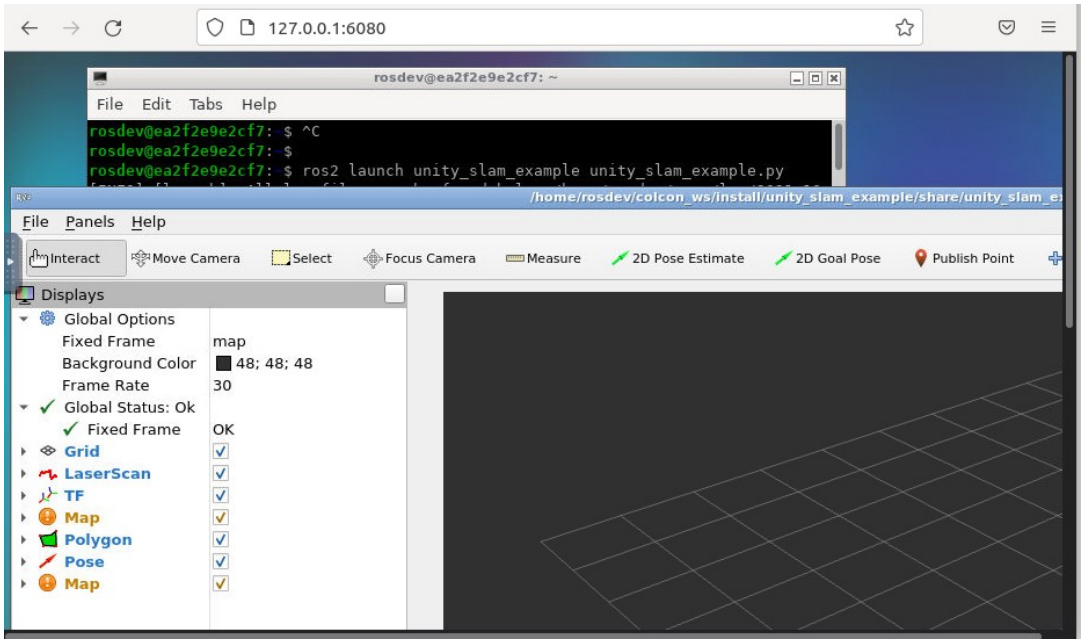
Start Rviz in the Docker Container



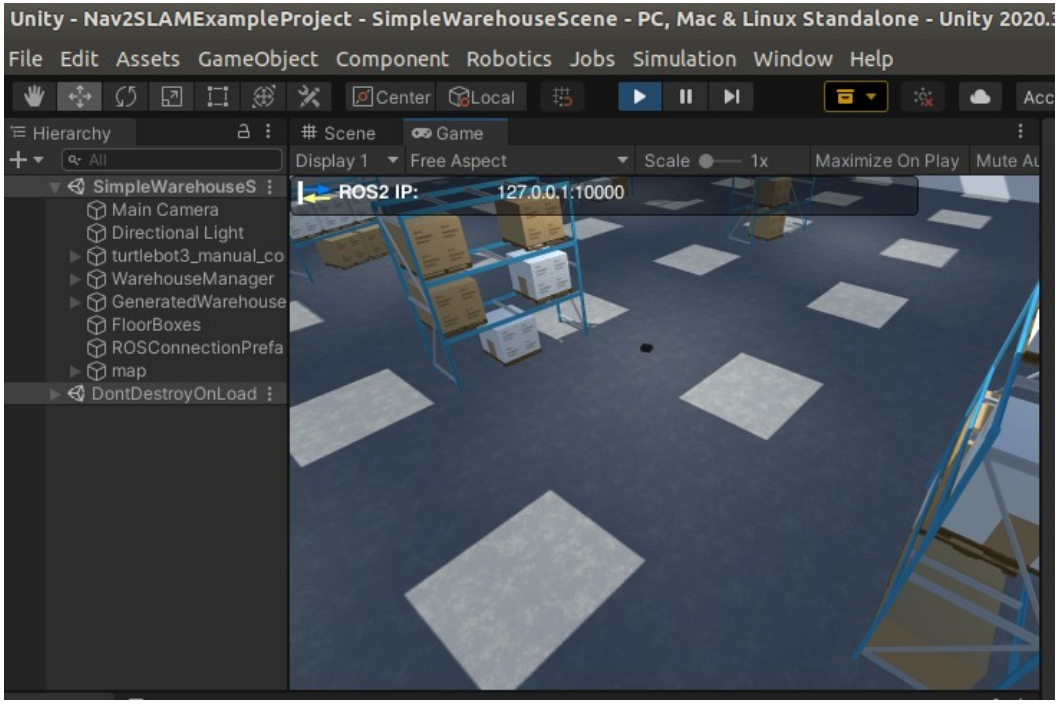
Running the Example

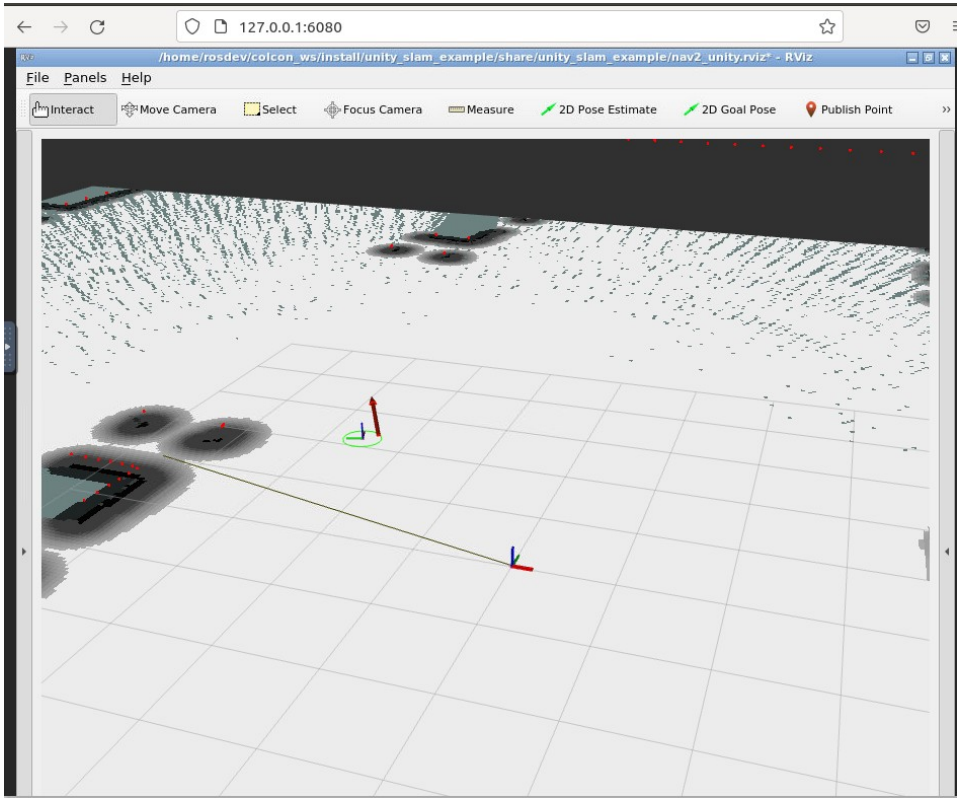
In a web browser connect to http://127.0.0.1:6080 and follow the steps below:





Running the Example



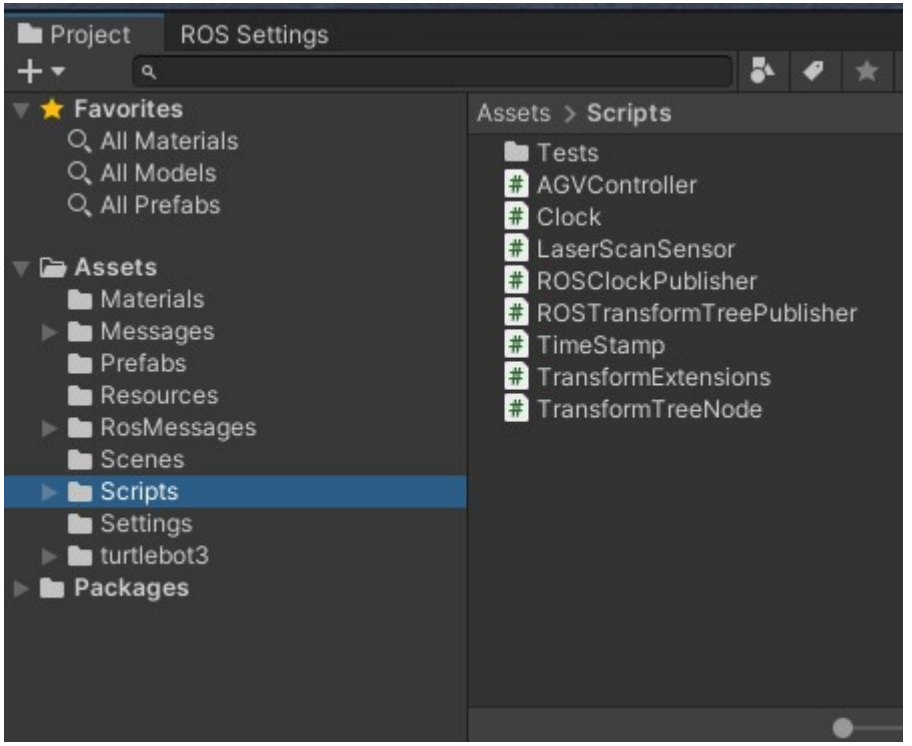


The TurtleBot is now localizing AND mapping, simultaneously! Now, to do navigation, click the 2D Goal Pose button, and left-click, drag, and release a location in RViz to send a commanded pose to the navigation stack.

Assets - Scripts

AGV Controller

The Automated Guided Vehicle Controller.
This Monobehaviour serves as a bridge between externally issued control signals and the ArticulationBody physics classes that we rely on to move our robots in physically accurate ways.




```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.Geometry;
using Unity.Robotics.UrdfImporter.Control;

namespace RosSharp.Control
{
    public enum ControlMode { Keyboard, ROS};

    public class AGVController : MonoBehaviour
    {
        public GameObject wheel1;
        public GameObject wheel2;
        public ControlMode mode = ControlMode.ROS;

        private ArticulationBody wA1;
        private ArticulationBody wA2;
```

Assets - Scripts

```
ROSConnection ros;
private RotationDirection direction;
private float rosLinear = 0f;
private float rosAngular = 0f;

void Start()
{
    wA1 = wheel1.GetComponent<ArticulationBody>();
    wA2 = wheel2.GetComponent<ArticulationBody>();
    SetParameters(wA1);
    SetParameters(wA2);
    ros = ROSConnection.GetOrCreateInstance();
    ros.Subscribe<TwistMsg>("cmd_vel", ReceiveROSCmd);
}

void ReceiveROSCmd(TwistMsg cmdVel)
{
    rosLinear = (float)cmdVel.linear.x;
    rosAngular = (float)cmdVel.angular.z;
    lastCmdReceived = Time.time;
}
```

```
private void RobotInput(float speed, float rotSpeed) // m/s and rad/s
{
    if (speed > maxLinearSpeed)
    {
        speed = maxLinearSpeed;
    }
    if (rotSpeed > maxRotationalSpeed)
    {
        rotSpeed = maxRotationalSpeed;
    }
    float wheel1Rotation = (speed / wheelRadius);
    float wheel2Rotation = wheel1Rotation;
    float wheelSpeedDiff = ((rotSpeed * trackWidth) / wheelRadius);
    if (rotSpeed != 0)
    {
        wheel1Rotation = (wheel1Rotation + (wheelSpeedDiff / 1)) * Mathf.Rad2Deg;
        wheel2Rotation = (wheel2Rotation - (wheelSpeedDiff / 1)) * Mathf.Rad2Deg;
    }
    else
    {
        wheel1Rotation *= Mathf.Rad2Deg;
        wheel2Rotation *= Mathf.Rad2Deg;
    }
    SetSpeed(wA1, wheel1Rotation);
    SetSpeed(wA2, wheel2Rotation);
}
```

ROS2 Workspace

The ROS 2 Workspace

```
unity_slam_example/
├── launch
│   └── unity_slam_example.py
├── package.xml
├── resource
│   └── unity_slam_example
├── rviz
│   └── nav2_unity.rviz
├── setup.cfg
└── setup.py
```

The ROS 2 workspace is relatively simple for this Project, as we are, for the most part, just calling the default Nav2 and slam_toolbox launch files with small modifications to account for Unity being used as the simulator instead of Gazebo.

Launch file (unity_slam_example.py)

Simply includes the appropriate LaunchDescriptions from the nav2 example but also ensure `use_sim_time` is set to `True` across all Nodes

RViz config (nav2_unity.rviz)

Defines an RViz layout to support visualizing the topics that will be published from Unity.

Other project files

`package.xml`, `setup.cfg`, and `setup.py` are simply bog standard ROS 2 package files. We pulled these directly from examples in the ROS 2 tutorials and stripped away anything that was not necessary to support our example.

ROS2 Workspace

unity_slam_example.py

rosdev@780d7c9803e2: ~/colcon_ws/src/unity_slam_example/launch

File Edit Tabs Help

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


def generate_launch_description():
    package_name = 'unity_slam_example'
    package_dir = get_package_share_directory(package_name)

    return LaunchDescription([
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                os.path.join(get_package_share_directory('ros_tcp_endpoint'), 'l
aunch', 'endpoint.py')
            ),
        ),


        Node(
            package='rviz2',
            executable='rviz2',
            output='screen',
            arguments=['-d', os.path.join(package_dir, 'nav2_unity.rviz')],
            parameters=[{'use_sim_time':True}]
        ),

        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
```

Assets - Scripts

Clock

In order to keep ROS 2 nodes and our time-dependent code in Unity synced, we define a `Clock` class that serves as an abstraction layer to ensure we use the same interface to access either Unity time or a ROS 2 time source. For the purposes of this example, we assume `use_sim_time` is true, and that Unity is providing the definitive clock.

LaserScanSensor

A simple implementation of a "perfect" 2-dimensional LIDAR sensor which provides scans instantaneously and without any signal noise. We are working hard to implement accurate, high-fidelity sensor models to replace simple examples like this in the future.

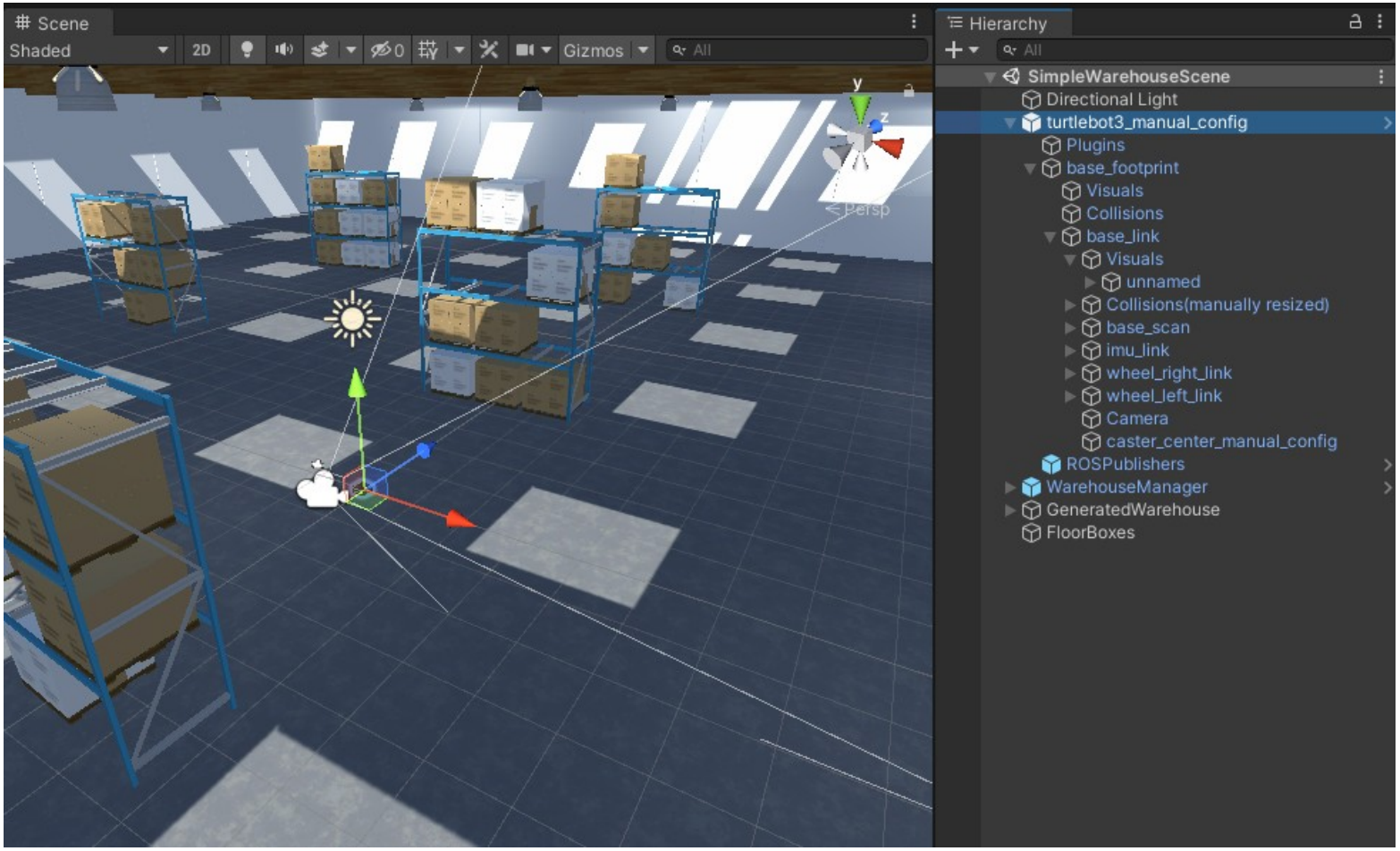
ROSClockPublisher

As the name implies, this publishes the output of our Clock class to the ROS `/clock` topic at fixed intervals. This allows other ROS 2 nodes to subscribe and stay in sync with the currently simulated time.

ROSTransformTreePublisher

This class, along with `TimeStamp`, `TransformExtensions`, and `TransformTreeNode`, allows us to construct and publish `tf2::TFMessage`s to ROS 2, representing the current state of the physical simulation in Unity.

Scenes



This is a prefab scene generated by our Robotics Warehouse Package.

The package is already installed in this project, alongside the Unity Perception package which provides the tooling that enables us to generate more randomized warehouses like this one.

Reference

https://github.com/Unity-Technologies/com.unity.perception

Perception Package (Unity Computer Vision)

The Perception package provides a toolkit for generating large-scale datasets for computer vision training and validation. It is focused on a handful of camera-based use cases for now and will ultimately expand to other forms of sensors and machine learning tasks.

Visit the Unity Computer Vision page for more information on our tools and offerings!

Getting Started

Quick Installation Instructions
Get your local Perception workspace up and running quickly. Recommended for users with prior Unity experience.

Perception Tutorial
Detailed instructions covering all the important steps from installing Unity Editor, to creating your first computer vision data generation project, building a randomized Scene, and generating large-scale synthetic datasets by leveraging the power of Unity Simulation. No prior Unity experience required.

Human Pose Labeling and Randomization Tutorial
Step by step instructions for using the keypoint, pose, and animation randomization tools included in the Perception package. It is recommended that you finish Phase 1 of the Perception Tutorial above before starting this tutorial.