Assest>RosMessages>Moveit>msg>RobotTrajectoryMsg

Imports and Namespace

The lines here represent a set of tools. They bring in the libraries and tools required for the code to run correctly. The fact that Unity.Robotics.ROSTCPConnector is specifically referred to indicates that ROS, a common robotics programming framework, is being used by this code.

```
//Do not edit! This file was generated by Unity-ROS MessageGeneration.
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
```

Class Definition

RobotTrajectoryMsg, the primary class, is built up in this section. Consider it a template for messages describing the trajectory of a robot's movement. This class is easier to handle because it is grouped with relevant classes in the namespace.

```
namespace RosMessageTypes.Moveit
{
    [Serializable]
    public class RobotTrajectoryMsg : Message
    {
        public const string k_RosMessageName = "moveit_msgs/RobotTrajectory";
        public override string RosMessageName => k_RosMessageName;
```

Fields and Constructors

Fields:

- `joint_trajectory`: Details the movement of individual joints in the robot.
- `multi_dof_joint_trajectory`: Describes more complex movements involving multiple degrees of freedom (like a robotic arm that can rotate and extend in multiple directions).

Constructors:

- The first one sets up a new, empty message.
- The second one allows you to start with specific movement details, if you already have them.

```
        public Trajectory.JointTrajectoryMsg joint_trajectory;
        public Trajectory.MultiDOFJointTrajectoryMsg multi_dof_joint_trajectory;

        public RobotTrajectoryMsg()
        {
            this.joint_trajectory = new Trajectory.JointTrajectoryMsg();
            this.multi_dof_joint_trajectory = new
Trajectory.MultiDOFJointTrajectoryMsg();
        }

        public RobotTrajectoryMsg(Trajectory.JointTrajectoryMsg joint_trajectory,
Trajectory.MultiDOFJointTrajectoryMsg multi_dof_joint_trajectory)
        {
            this.joint_trajectory = joint_trajectory;
            this.multi_dof_joint_trajectory = multi_dof_joint_trajectory;
        }
```

Deserialization and Serialization

- **Deserialization**: This means turning a message from data format back into a `RobotTrajectoryMsg` object. It's like unpacking a box.

- **Serialization**: This means turning a `RobotTrajectoryMsg` object into a data format for sending out. It's like packing a box for shipping.

```
        public static RobotTrajectoryMsg Deserialize(MessageDeserializer
deserializer) => new RobotTrajectoryMsg(deserializer);

        private RobotTrajectoryMsg(MessageDeserializer deserializer)
        {
            this.joint_trajectory =
Trajectory.JointTrajectoryMsg.Deserialize(deserializer);
            this.multi_dof_joint_trajectory =
Trajectory.MultiDOFJointTrajectoryMsg.Deserialize(deserializer);
        }

        public override void SerializeTo(MessageSerializer serializer)
        {
            serializer.Write(this.joint_trajectory);
            serializer.Write(this.multi_dof_joint_trajectory);
        }
```

ToString Method

This method provides a simple, human-readable summary of what's inside the RobotTrajectoryMsg. It's useful for debugging or logging what the robot is supposed to do.

```
        public override string ToString()
        {
            return "RobotTrajectoryMsg: " +
            "\njoint_trajectory: " + joint_trajectory.ToString() +
            "\nmulti_dof_joint_trajectory: " +
multi_dof_joint_trajectory.ToString();
        }
```

Registration

This part makes sure that the `RobotTrajectoryMsg` class is properly registered with the system. It's like adding a new employee to the company directory so everyone knows their role and responsibilities.

```
#if UNITY_EDITOR
        [UnityEditor.InitializeOnLoadMethod]
#else
        [UnityEngine.RuntimeInitializeOnLoadMethod]
#endif
        public static void Register()
        {
            MessageRegistry.Register(k_RosMessageName, Deserialize);
        }
    }
}
```

Assest>RosMessages>NiryoMoveit>msg>NiryoMoveitJointsMsg

Imports and Namespace

The required libraries are imported in these lines. They give the code the means to function, managing collections, transforming data into strings, and interacting with ROS via Unity, among other things.

```
//Do not edit! This file was generated by Unity-ROS MessageGeneration.
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
```

Class Definition

- **Namespace**: Groups this class with others related to Niryo robots and Moveit, which is a ROS library for robot motion planning.

- **Class Definition**: Defines a message type `NiryoMoveitJointsMsg` for handling joint movements and poses (positions and orientations).

```
namespace RosMessageTypes.NiryoMoveit
{
    [Serializable]
    public class NiryoMoveitJointsMsg : Message
    {
        public const string k_RosMessageName = "niryo_moveit/NiryoMoveitJoints";
        public override string RosMessageName => k_RosMessageName;
```

Fields and Constructors

- **Fields**:

  - `joints`: An array representing the positions of the robot's six joints.

  - `pick_pose`: The position and orientation where the robot should pick up an object.

  - `place_pose`: The position and orientation where the robot should place an object.

- **Constructors**:

  - The default constructor initializes the fields with default values.

  - The parameterized constructor allows initializing the fields with specific values, which is useful when creating a message with known joint positions and poses.

```
        public double[] joints;
        public Geometry.PoseMsg pick_pose;
        public Geometry.PoseMsg place_pose;

        public NiryoMoveitJointsMsg()
        {
            this.joints = new double[6];
            this.pick_pose = new Geometry.PoseMsg();
            this.place_pose = new Geometry.PoseMsg();
        }
```

```
        public NiryoMoveitJointsMsg(double[] joints, Geometry.PoseMsg pick_pose,
Geometry.PoseMsg place_pose)
        {
            this.joints = joints;
            this.pick_pose = pick_pose;
            this.place_pose = place_pose;
        }
```

Deserialization and Serialization

Deserialization: Converts data from a message back into a `NiryoMoveitJointsMsg` object. It's like unpacking information from a package.

- The static method `Deserialize` creates a new `NiryoMoveitJointsMsg` using the private constructor.
- The private constructor reads the data for the joints, pick pose, and place pose from the deserializer.

Serialization: Converts a `NiryoMoveitJointsMsg` object into a format suitable for sending as a message. It's like packing information into a package.

- `SerializeTo` method writes the data for the joints, pick pose, and place pose to the serializer.

```
        public static NiryoMoveitJointsMsg Deserialize(MessageDeserializer
deserializer) => new NiryoMoveitJointsMsg(deserializer);

        private NiryoMoveitJointsMsg(MessageDeserializer deserializer)
        {
            deserializer.Read(out this.joints, sizeof(double), 6);
            this.pick_pose = Geometry.PoseMsg.Deserialize(deserializer);
            this.place_pose = Geometry.PoseMsg.Deserialize(deserializer);
        }

        public override void SerializeTo(MessageSerializer serializer)
        {
            serializer.Write(this.joints);
            serializer.Write(this.pick_pose);
            serializer.Write(this.place_pose);
        }
```

ToString Method

This method provides a readable summary of the message contents. It's useful for debugging or logging. It shows the values of the joints and the pick and place poses.

```
        public override string ToString()
        {
            return "NiryoMoveitJointsMsg: " +
            "\njoints: " + System.String.Join(", ", joints.ToList()) +
            "\npick_pose: " + pick_pose.ToString() +
            "\nplace_pose: " + place_pose.ToString();
        }
```

Registration

This part registers the NiryoMoveitJointsMsg class with the message system. It ensures that the system knows how to handle this type of message, similar to adding a new employee to a company directory so everyone knows their role and responsibilities.

```
#if UNITY_EDITOR
        [UnityEditor.InitializeOnLoadMethod]
#else
        [UnityEngine.RuntimeInitializeOnLoadMethod]
#endif
        public static void Register()
        {
            MessageRegistry.Register(k_RosMessageName, Deserialize);
```

```
        }
    }
}
```

Assest>RosMessages>NiryoMoveit>msg>NiryoTrajectoryMsg

Imports and Namespace

These lines import necessary libraries. They provide tools the code needs to function, such as handling collections, converting data to strings, and integrating with ROS through Unity.

```
//Do not edit! This file was generated by Unity-ROS MessageGeneration.
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
```

Class Definition

- **Namespace**: Groups this class with others related to Niryo robots and Moveit, making it easier to manage and organize.

- **Class Definition**: Defines a message type `NiryoTrajectoryMsg` for handling robot movement trajectories.

```
namespace RosMessageTypes.NiryoMoveit
{
    [Serializable]
    public class NiryoTrajectoryMsg : Message
    {
        public const string k_RosMessageName = "niryo_moveit/NiryoTrajectory";
        public override string RosMessageName => k_RosMessageName;
```

Fields and Constructors

Fields:

- `trajectory`: An array that holds multiple `RobotTrajectoryMsg` objects. Each `RobotTrajectoryMsg` represents a specific path or sequence of movements for the robot.

Constructors:

- The default constructor initializes `trajectory` with an empty array.
- The parameterized constructor allows initializing `trajectory` with a specific array of `RobotTrajectoryMsg` objects, useful when creating a message with known trajectories.

```
        public Moveit.RobotTrajectoryMsg[] trajectory;

        public NiryoTrajectoryMsg()
        {
            this.trajectory = new Moveit.RobotTrajectoryMsg[0];
        }

        public NiryoTrajectoryMsg(Moveit.RobotTrajectoryMsg[] trajectory)
        {
            this.trajectory = trajectory;
        }
```

Deserialization and Serialization

- **Deserialization**: Converts data from a message format back into a `NiryoTrajectoryMsg` object. It's like unpacking information from a package.
    - The static method `Deserialize` creates a new `NiryoTrajectoryMsg` using the private constructor.
    - The private constructor reads the array of `RobotTrajectoryMsg` objects from the deserializer, based on the length of the array.

- **Serialization**: Converts a `NiryoTrajectoryMsg` object into a message format for sending out. It's like packing information into a package.
    - `SerializeTo` method writes the length of the `trajectory` array and then the array itself to the serializer.

```
        public static NiryoTrajectoryMsg Deserialize(MessageDeserializer
deserializer) => new NiryoTrajectoryMsg(deserializer);

        private NiryoTrajectoryMsg(MessageDeserializer deserializer)
        {
            deserializer.Read(out this.trajectory,
Moveit.RobotTrajectoryMsg.Deserialize, deserializer.ReadLength());
        }

        public override void SerializeTo(MessageSerializer serializer)
        {
            serializer.WriteLength(this.trajectory);
            serializer.Write(this.trajectory);
        }
```

ToString Method

This method provides a readable summary of the message contents. It's useful for debugging or logging. It shows the contents of the trajectory array as a comma-separated list.

```
        public override string ToString()
        {
            return "NiryoTrajectoryMsg: " +
            "\ntrajectory: " + System.String.Join(", ", trajectory.ToList());
        }
```

Registration

This part registers the NiryoTrajectoryMsg class with the message system. It ensures that the system knows how to handle this type of message, similar to adding a new employee to a company directory so everyone knows their role and responsibilities.

```
#if UNITY_EDITOR
        [UnityEditor.InitializeOnLoadMethod]
#else
        [UnityEngine.RuntimeInitializeOnLoadMethod]
#endif
        public static void Register()
        {
            MessageRegistry.Register(k_RosMessageName, Deserialize);
        }
    }
}
```

Assest>RosMessages>NiryoMoveit>srv>MoverServiceResponce

```
//Do not edit! This file was generated by Unity-ROS MessageGeneration.
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
```

Class Definition

- **Namespace**: Groups this class with others related to Niryo robots and Moveit, making it easier to manage and organize.

- **Class Definition**: Defines a message type `MoverServiceResponse` for handling responses from the mover service. This service plans the robot's movements.

```
namespace RosMessageTypes.NiryoMoveit
{
    [Serializable]
    public class MoverServiceResponse : Message
    {
        public const string k_RosMessageName = "niryo_moveit/MoverService";
        public override string RosMessageName => k_RosMessageName;
```

Fields and Constructors

```
        public Moveit.RobotTrajectoryMsg[] trajectories;

        public MoverServiceResponse()
        {
            this.trajectories = new Moveit.RobotTrajectoryMsg[0];
        }

        public MoverServiceResponse(Moveit.RobotTrajectoryMsg[] trajectories)
        {
            this.trajectories = trajectories;
        }
```

Deserialization and Serialization

Deserialization: Converts data from a message format back into a `MoverServiceResponse` object. It's like unpacking information from a package.

- The static method `Deserialize` creates a new `MoverServiceResponse` using the private constructor.
- The private constructor reads the array of `RobotTrajectoryMsg` objects from the deserializer, based on the length of the array.

Serialization: Converts a `MoverServiceResponse` object into a message format for sending out. It's like packing information into a package.

- `SerializeTo` method writes the length of the `trajectories` array and then the array itself to the serializer.

```
        public static MoverServiceResponse Deserialize(MessageDeserializer
deserializer) => new MoverServiceResponse(deserializer);

        private MoverServiceResponse(MessageDeserializer deserializer)
        {
            deserializer.Read(out this.trajectories,
Moveit.RobotTrajectoryMsg.Deserialize, deserializer.ReadLength());
        }
```

ToString Method

```
        public override void SerializeTo(MessageSerializer serializer)
        {
            serializer.WriteLength(this.trajectories);
            serializer.Write(this.trajectories);
        }

        public override string ToString()
        {
            return "MoverServiceResponse: " +
            "\ntrajectories: " + System.String.Join(", ", trajectories.ToList());
        }
```

Registration

This part registers the MoverServiceResponse class with the message system. It ensures that the system knows how to handle this type of message, similar to adding a new employee to a company directory so everyone knows their role and responsibilities. The MessageSubtopic.Response indicates that this message is specifically a response type.

```
#if UNITY_EDITOR
        [UnityEditor.InitializeOnLoadMethod]
#else
        [UnityEngine.RuntimeInitializeOnLoadMethod]
#endif
        public static void Register()
        {
            MessageRegistry.Register(k_RosMessageName, Deserialize,
MessageSubtopic.Response);
        }
    }
}
```

Assest>RosMessages>NiryoMoveit>srv>MoverServiceRequest

Imports and Namespace

```
//Do not edit! This file was generated by Unity-ROS MessageGeneration.
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
```

Class Definition

```
namespace RosMessageTypes.NiryoMoveit
{
    [Serializable]
    public class MoverServiceRequest : Message
    {
        public const string k_RosMessageName = "niryo_moveit/MoverService";
        public override string RosMessageName => k_RosMessageName;
```

Fields and Constructors

```
        public NiryoMoveitJointsMsg joints_input;
        public Geometry.PoseMsg pick_pose;
        public Geometry.PoseMsg place_pose;

        public MoverServiceRequest()
        {
            this.joints_input = new NiryoMoveitJointsMsg();
```

```
            this.pick_pose = new Geometry.PoseMsg();
            this.place_pose = new Geometry.PoseMsg();
        }

        public MoverServiceRequest(NiryoMoveitJointsMsg joints_input,
Geometry.PoseMsg pick_pose, Geometry.PoseMsg place_pose)
        {
            this.joints_input = joints_input;
            this.pick_pose = pick_pose;
            this.place_pose = place_pose;
        }
```

Fields and Constructors

Deserialization: Converts data from a message format back into a `MoverServiceRequest` object. It's like unpacking information from a package.

- The static method `Deserialize` creates a new `MoverServiceRequest` using the private constructor.
- The private constructor reads the `joints_input`, `pick_pose`, and `place_pose` objects from the deserializer.

Serialization: Converts a `MoverServiceRequest` object into a message format for sending out. It's like packing information into a package.

- `SerializeTo` method writes the `joints_input`, `pick_pose`, and `place_pose` objects to the serializer.

```
        public static MoverServiceRequest Deserialize(MessageDeserializer
deserializer) => new MoverServiceRequest(deserializer);

        private MoverServiceRequest(MessageDeserializer deserializer)
        {
            this.joints_input = NiryoMoveitJointsMsg.Deserialize(deserializer);
            this.pick_pose = Geometry.PoseMsg.Deserialize(deserializer);
            this.place_pose = Geometry.PoseMsg.Deserialize(deserializer);
        }

        public override void SerializeTo(MessageSerializer serializer)
        {
            serializer.Write(this.joints_input);
            serializer.Write(this.pick_pose);
            serializer.Write(this.place_pose);
        }
```

ToString Method

```
        public override string ToString()
        {
            return "MoverServiceRequest: " +
            "\njoints_input: " + joints_input.ToString() +
            "\npick_pose: " + pick_pose.ToString() +
            "\nplace_pose: " + place_pose.ToString();
        }
```

Registration

```
#if UNITY_EDITOR
        [UnityEditor.InitializeOnLoadMethod]
#else
        [UnityEngine.RuntimeInitializeOnLoadMethod]
#endif
        public static void Register()
        {
            MessageRegistry.Register(k_RosMessageName, Deserialize);
        }
    }
}
```

Assest>Scripts>NiryoMoveit> SourceDestinationPublisher

Imports and Declarations

- `System`: Basic functionalities like data types and collections.
- `RosMessageTypes.Geometry` **and** `RosMessageTypes.NiryoMoveit`: For handling specific message types used in ROS communication.
- `Unity.Robotics.ROSTCPConnector`: Provides tools to connect Unity with ROS.
- `Unity.Robotics.ROSTCPConnector.ROSGeometry`: Converts Unity data types to ROS-compatible formats.
- `Unity.Robotics.UrdfImporter`: For working with URDF (Unified Robot Description Format) files to interact with robot joints.
- `UnityEngine`: Core Unity functionalities.

```
using System;
using RosMessageTypes.Geometry;
using RosMessageTypes.NiryoMoveit;
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.ROSGeometry;
using Unity.Robotics.UrdfImporter;
using UnityEngine;
```

Class Definition and Variables

- `k_NumRobotJoints`: Constant that defines the number of robot joints. Here, it's set to 6, which matches the number of joints in the Niryo robot.
- `LinkNames`: Array of strings that names each link of the robot in the URDF file. These names are used to find and interact with the joints.
- `m_TopicName`: The ROS topic where the data will be published. It's like an address where the data is sent.
- `m_NiryoOne`: The GameObject representing the Niryo robot in Unity.
- `m_Target` and `m_TargetPlacement`: GameObjects representing the positions where the robot will pick up and place objects.
- `m_PickOrientation`: The orientation (rotation) used when the robot picks up an object. It's a fixed rotation defined as 90 degrees around the X and Y axes.
- `m_JointArticulationBodies`: Array to hold references to the robot's joints.
- `m_Ros`: The connection to the ROS server.

```
public class SourceDestinationPublisher : MonoBehaviour
{
    const int k_NumRobotJoints = 6;

    public static readonly string[] LinkNames =
        { "world/base_link/shoulder_link", "/arm_link", "/elbow_link",
"/forearm_link", "/wrist_link", "/hand_link" };

    // Variables required for ROS communication
    [SerializeField]
    string m_TopicName = "/niryo_joints";

    [SerializeField]
    GameObject m_NiryoOne;
    [SerializeField]
    GameObject m_Target;
    [SerializeField]
    GameObject m_TargetPlacement;
```

```
    readonly Quaternion m_PickOrientation = Quaternion.Euler(90, 90, 0);

    // Robot Joints
    UrdfJointRevolute[] m_JointArticulationBodies;

    // ROS Connector
    ROSConnection m_Ros;
```

Start Method

Start **Method**: Runs once when the script starts.

- `m_Ros`: Initializes or retrieves the existing ROS connection.
- `m_Ros.RegisterPublisher`: Registers the script as a publisher of `NiryoMoveitJointsMsg` messages on the specified topic. This tells ROS that this Unity script will send data on this topic.
- `m_JointArticulationBodies`: Initializes the array for holding the joint references.
- `for` **Loop**: Loops through each joint, builds the link name, finds the joint in the Unity hierarchy, and stores the joint reference.

```
    void Start()
    {
        // Get ROS connection static instance
        m_Ros = ROSConnection.GetOrCreateInstance();
        m_Ros.RegisterPublisher<NiryoMoveitJointsMsg>(m_TopicName);

        m_JointArticulationBodies = new UrdfJointRevolute[k_NumRobotJoints];

        var linkName = string.Empty;
        for (var i = 0; i < k_NumRobotJoints; i++)
        {
            linkName += LinkNames[i];
            m_JointArticulationBodies[i] =
m_NiryoOne.transform.Find(linkName).GetComponent<UrdfJointRevolute>();
        }
    }
```

Publish Method

Publish **Method**: Sends a message with robot joint positions and poses to the ROS server.

- `sourceDestinationMessage`: Creates a new message of type `NiryoMoveitJointsMsg`.
- `for` **Loop**: Retrieves the position of each joint from the `m_JointArticulationBodies` array and stores it in the message.
- `pick_pose`: Sets the position and orientation where the robot will pick up an object. Converts Unity's position and orientation to ROS-compatible formats.
- `place_pose`: Sets the position and orientation where the robot will place an object, using the predefined `m_PickOrientation`.
- `m_Ros.Publish`: Sends the `sourceDestinationMessage` to the ROS server on the specified topic. This is like sending the package with all the robot's movement instructions.

```
    public void Publish()
    {
        var sourceDestinationMessage = new NiryoMoveitJointsMsg();

        for (var i = 0; i < k_NumRobotJoints; i++)
        {
            sourceDestinationMessage.joints[i] =
m_JointArticulationBodies[i].GetPosition();
        }

        // Pick Pose
        sourceDestinationMessage.pick_pose = new PoseMsg
        {
            position = m_Target.transform.position.To<FLU>(),
            orientation = Quaternion.Euler(90, m_Target.transform.eulerAngles.y,
0).To<FLU>()
```

```
        };

        // Place Pose
        sourceDestinationMessage.place_pose = new PoseMsg
        {
            position = m_TargetPlacement.transform.position.To<FLU>(),
            orientation = m_PickOrientation.To<FLU>()
        };

        // Finally send the message to server_endpoint.py running in ROS
        m_Ros.Publish(m_TopicName, sourceDestinationMessage);
    }
}
```

Assest>Scripts>NiryoMoveit> TargetPlacement

Imports and Namespace

- `using System;` and others: Import necessary libraries.
- `namespace Unity.Robotics.PickAndPlace`: Organizes this code under the `Unity.Robotics.PickAndPlace` namespace.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace Unity.Robotics.PickAndPlace
{
```

Class Definition and Variables

- **Attributes**: `[RequireComponent(typeof(MeshRenderer))]` and `[RequireComponent(typeof(BoxCollider))]` ensure these components are attached to the GameObject.
- **Constants**: `k_NameExpectedTarget`, `k_ShaderColorId`, and `k_MaximumSpeedForStopped` define fixed values for target name, shader color ID, and speed threshold for considering the target "stopped."
- **Serialized Fields**: `m_Target` is the target GameObject, and `m_ColorAlpha` sets the alpha transparency for color changes.
- **Other Variables**: These hold references to the mesh renderer and box collider, and manage the current and previous states of the target placement.
- **PlacementState Enum**: Defines three states - `Outside`, `InsideFloating`, and `InsidePlaced`.

Autonomous Mode

The TargetPlacement class is used in an autonomous mode. It automatically updates the placement state of the target object based on its position and speed without requiring manual intervention. The system continuously checks the conditions and updates the state and color accordingly.

```
    [RequireComponent(typeof(MeshRenderer))]
    [RequireComponent(typeof(BoxCollider))]
    public class TargetPlacement : MonoBehaviour
    {
        const string k_NameExpectedTarget = "Target";
        static readonly int k_ShaderColorId = Shader.PropertyToID("_Color");
        // The threshold that the Target's speed must be under to be considered
"placed" in the target area
```

```
        const float k_MaximumSpeedForStopped = 0.01f;

        [SerializeField]
        [Tooltip("Target object expected by this placement area. Can be left
blank if only one Target in scene")]
        GameObject m_Target;
        [SerializeField]
        [Range(0, 255)]
        [Tooltip("Alpha value for any color set during state changes.")]
        int m_ColorAlpha = 100;

        MeshRenderer m_TargetMeshRenderer;

        float m_ColorAlpha01 => m_ColorAlpha / 255f;
        MeshRenderer m_MeshRenderer;
        BoxCollider m_BoxCollider;
        PlacementState m_CurrentState;
        PlacementState m_LastColoredState;

        public PlacementState CurrentState
        {
            get => m_CurrentState;
            private set
            {
                m_CurrentState = value;
                UpdateStateColor();
            }
        }

        public enum PlacementState
        {
            Outside,
            InsideFloating,
            InsidePlaced
        }
```

Start Method

Start Method: This method runs once when the script starts.

- **Find Target**: If `m_Target` is not set, it looks for a GameObject named `Target`.
- **Check Target**: If no target is found, it logs a warning and disables the script.
- **Set Component References**: Calls `TrySetComponentReferences()` to get necessary components.
- **Initialize State**: Calls `InitializeState()` to determine the initial state of the target.

Autonomous Mode

The initialization process described in the Start method is characteristic of an autonomous mode. The script automatically checks for necessary components, attempts to find missing references, and sets the initial state without requiring manual intervention. This automatic setup is consistent with autonomous system behavior, where the system handles initial configurations and runtime operations on its own.

```
        // Start is called before the first frame update
        void Start()
        {
            // Check for mis-configurations and disable if something has changed
without this script being updated
            // These are warnings because this script does not contain critical
functionality
            if (m_Target == null)
            {
                m_Target = GameObject.Find(k_NameExpectedTarget);
            }

            if (m_Target == null)
            {
```

```
                Debug.LogWarning($"{nameof(TargetPlacement)} expects to find a
GameObject named " +
                    $"{k_NameExpectedTarget} to track, but did not. Can't track
placement state.");
                enabled = false;
                return;
            }

            if (!TrySetComponentReferences())
            {
                enabled = false;
                return;
            }
            InitializeState();
        }
```

TrySetComponentReferences Method

TrySetComponentReferences Method: Gets references to the necessary components.

- **Check Target Renderer**: Checks if the target has a `MeshRenderer` and logs a warning if not found.
- **Assign Components**: Assigns the mesh renderer and box collider for the target area.

Autonomous Mode

This method supports autonomous behavior because it automatically sets up component references without requiring manual intervention. By checking for required components and logging warnings if they are missing, the script can handle initial setup issues autonomously. This self-sufficiency is characteristic of autonomous systems, where the system manages its configurations and dependencies on its own.

```
        bool TrySetComponentReferences()
        {
            m_TargetMeshRenderer = m_Target.GetComponent<MeshRenderer>();
            if (m_TargetMeshRenderer == null)
            {
                Debug.LogWarning($"{nameof(TargetPlacement)} expects a
{nameof(MeshRenderer)} to be attached " +
                    $"to {k_NameExpectedTarget}. Cannot check bounds without it,
so cannot track placement state.");
                return false;
            }

            // Assume these are here because they are RequiredComponent
components
            m_MeshRenderer = GetComponent<MeshRenderer>();
            m_BoxCollider = GetComponent<BoxCollider>();
            return true;
        }
```

OnValidate Method

OnValidate Method: This runs when values are changed in the Unity Editor.

- **Set References and Initialize**: If the target is set, it tries to set the component references and initialize the state.

Manual Mode

OnValidate facilitates manual configuration and debugging during development, ensuring that your components are correctly set up before you enter Play Mode.

```
        void OnValidate()
```

```
        {
            // Useful for visualizing state in editor, but doesn't wholly
guarantee accurate coloring in EditMode
            // Enter PlayMode to see color update correctly
            if (m_Target != null)
            {
                if (TrySetComponentReferences())
                {
                    InitializeState();
                }
            }
        }
```

InitializeState Method

InitializeState Method: Determines the initial state of the target.

- **Check Intersection**: Checks if the target's collider intersects with the placement area's collider.
- **Set State**: Sets the state based on whether the target is inside and if it is stopped.

Manual Mode

The InitializeState method is used for manually determining the placement state of an object based on its initial conditions and interactions with other colliders. It sets the initial state of the target object, which is a part of manual setup and control.

```
void InitializeState()
        {
            if
(m_Target.GetComponent<BoxCollider>().bounds.Intersects(m_BoxCollider.bounds))
            {
                CurrentState = IsTargetStoppedInsideBounds() ?
                    PlacementState.InsidePlaced : PlacementState.InsideFloating;
            }
            else
            {
                CurrentState = PlacementState.Outside;
            }
        }
```

Manual Mode

OnTriggerEnter Method

- **OnTriggerExit Method**: Called when another collider exits the placement area.
- **Set State**: If the other collider is the target, sets the state to `Outside`.

Manual Mode

These methods are used to manually track the target's position relative to the trigger area. They react to collision events and update the state based on whether the target is inside or outside the trigger area. This is part of manual management of the target's state in Unity's physics system.

```
        void OnTriggerEnter(Collider other)
        {
            if (other.gameObject.name == m_Target.name)
            {
                CurrentState = PlacementState.InsideFloating;
            }
        }

        void OnTriggerExit(Collider other)
```

```
    {
        if (other.gameObject.name == m_Target.name)
        {
            CurrentState = PlacementState.Outside;
        }
    }
```

IsTargetStoppedInsideBounds Method

IsTargetStoppedInsideBounds Method: Checks if the target is stopped and within bounds.

- **Check Speed**: Ensures the target's velocity is below the threshold.
- **Check Bounds**: Ensures the target's center is within the placement area's bounds.
- **Return Result**: Returns true if both conditions are met.

Manual Mode

This method is used for manual checks within the game logic to ensure that the target object meets the criteria for being considered placed. It doesn't control the robot directly but instead provides information used by other parts of the code to make decisions (e.g., update the state).

```
    bool IsTargetStoppedInsideBounds()
    {
        var targetIsStopped =
m_Target.GetComponent<Rigidbody>().velocity.magnitude < k_MaximumSpeedForStopped;
        var targetIsInBounds =
m_BoxCollider.bounds.Contains(m_TargetMeshRenderer.bounds.center);

        return targetIsStopped && targetIsInBounds;
    }
```

Update Method

Update Method: Runs once per frame.

- **Update State**: If the current state is not `Outside`, checks if the target is stopped and updates the state accordingly.

Autonomous Mode

The Update method reflects the automatic behavior of the script in response to the target's position and movement. It continuously updates the state of the placement area without direct user input, thus falling into the autonomous category.

```
    // Update is called once per frame
    void Update()
    {
        if (CurrentState != PlacementState.Outside)
        {
            CurrentState = IsTargetStoppedInsideBounds() ?
                PlacementState.InsidePlaced : PlacementState.InsideFloating;
        }
    }
```

UpdateStateColor Method

UpdateStateColor Method: Changes the color of the placement area based on the state.

- **Check State**: If the state hasn't changed, does nothing.
- **Set Color**: Creates a new color based on the state (`red` for `Outside`, `yellow` for `InsideFloating`, and `green` for `InsidePlaced`).
- **Apply Color**: Sets the color using a material property block.
- **Update Last State**: Updates the last colored state to the current state.

Manual Mode

The UpdateStateColor method is related to visual feedback and does not directly control or modify the robot's operations or movements. It updates the visual representation based on the object's state, which is a manual process driven by the state changes.

```
        void UpdateStateColor()
        {
            if (m_CurrentState == m_LastColoredState)
            {
                return;
            }

            var mpb = new MaterialPropertyBlock();
            Color stateColor;
            switch (m_CurrentState)
            {
                case PlacementState.Outside:
                    stateColor = Color.red;
                    break;
                case PlacementState.InsideFloating:
                    stateColor = Color.yellow;
                    break;
                case PlacementState.InsidePlaced:
                    stateColor = Color.green;
                    break;
                default:
                    Debug.LogError($"No state handling implemented for
{m_CurrentState}");
                    stateColor = Color.magenta;
                    break;
            }

            stateColor.a = m_ColorAlpha01;
            mpb.SetColor(k_ShaderColorId, stateColor);
            m_MeshRenderer.SetPropertyBlock(mpb);
            m_LastColoredState = m_CurrentState;
        }
    }
}
```

Assest>Scripts>NiryoMoveit> TrajectoryPlanner

Imports and Variables

Imports: Bring in necessary libraries and types for ROS communication, geometry handling, and Unity functionalities.

```
using System;
using System.Collections;
using System.Linq;
using RosMessageTypes.Geometry;
using RosMessageTypes.NiryoMoveit;
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.ROSGeometry;
using UnityEngine;
```

Hardcoded Variables: Set the number of robot joints and wait times between joint and pose assignments.

Autonomous Mode

The TrajectoryPlanner class is used to automate the robot's movements and trajectory execution. It relies on ROS to plan movements and manage timing, indicating that it is designed for autonomous operation. The constants provided help manage the timing and number of joints involved in the planning and execution process.

```
public class TrajectoryPlanner : MonoBehaviour
{
    // Hardcoded variables
    const int k_NumRobotJoints = 6;
    const float k_JointAssignmentWait = 0.1f;
    const float k_PoseAssignmentWait = 0.5f;
```

Serialized Fields: Allow configuration of ROS service name and GameObject references for the robot, target object, and target placement area within the Unity Editor.

Autonomous Mode

The class uses these fields to set up and manage robot operations, communicate with ROS services, and define target interactions

```
    // Variables required for ROS communication
    [SerializeField]
    string m_RosServiceName = "niryo_moveit";
    public string RosServiceName { get => m_RosServiceName; set =>
m_RosServiceName = value; }

    [SerializeField]
    GameObject m_NiryoOne;
    public GameObject NiryoOne { get => m_NiryoOne; set => m_NiryoOne = value; }
    [SerializeField]
    GameObject m_Target;
    public GameObject Target { get => m_Target; set => m_Target = value; }
    [SerializeField]
    GameObject m_TargetPlacement;
    public GameObject TargetPlacement { get => m_TargetPlacement; set =>
m_TargetPlacement = value; }
```

Pick Orientation and Offset: Define the orientation and offset for picking up the target object.

Autonomous Mode

These variables and their usage are typical of **autonomous systems**, where precise positioning and orientation are required for automated tasks. The use of specific orientations and offsets implies that the system operates without manual intervention, as these settings are pre-defined to ensure consistent and reliable robot actions.

```
    // Assures that the gripper is always positioned above the m_Target cube
before grasping.
    readonly Quaternion m_PickOrientation = Quaternion.Euler(90, 90, 0);
    readonly Vector3 m_PickPoseOffset = Vector3.up * 0.1f;
```

Articulation Bodies: Variables to store the robot's joint components and gripper components.

ROS Connector: Variable to store the ROS connection instance.

Autonomous Mode

This code is part of an autonomous system where the robot's movements and interactions are controlled programmatically rather than manually. The setup ensures that the robot can perform tasks like moving its joints and grippers according to pre-defined commands and interactions with ROS.

```
// Articulation Bodies
    ArticulationBody[] m_JointArticulationBodies;
    ArticulationBody m_LeftGripper;
    ArticulationBody m_RightGripper;

    // ROS Connector
    ROSConnection m_Ros;

    /// <summary>
    ///     Find all robot joints in Awake() and add them to the
jointArticulationBodies array.
    ///     Find left and right finger joints and assign them to their respective
articulation body objects.
    /// </summary>
```

Start Method

- **Initialize ROS**: Get the ROS connection instance and register the ROS service.
- **Get Joint Components**: Find and store references to the robot's joint components.
- **Get Gripper Components**: Find and store references to the robot's gripper components.

Autonomous Mode

the code is intended for **autonomous** operation, where the robot's actions are managed programmatically based on commands from ROS.

```
    void Start()
    {
        // Get ROS connection static instance
        m_Ros = ROSConnection.GetOrCreateInstance();
        m_Ros.RegisterRosService<MoverServiceRequest,
MoverServiceResponse>(m_RosServiceName);

        m_JointArticulationBodies = new ArticulationBody[k_NumRobotJoints];
```

```
        var linkName = string.Empty;
        for (var i = 0; i < k_NumRobotJoints; i++)
        {
            linkName += SourceDestinationPublisher.LinkNames[i];
            m_JointArticulationBodies[i] =
m_NiryoOne.transform.Find(linkName).GetComponent<ArticulationBody>();
        }

        // Find left and right fingers
        var rightGripper = linkName +
"/tool_link/gripper_base/servo_head/control_rod_right/right_gripper";
        var leftGripper = linkName +
"/tool_link/gripper_base/servo_head/control_rod_left/left_gripper";

        m_RightGripper =
m_NiryoOne.transform.Find(rightGripper).GetComponent<ArticulationBody>();
        m_LeftGripper =
m_NiryoOne.transform.Find(leftGripper).GetComponent<ArticulationBody>();
    }
```

Gripper Methods

CloseGripper Method: Closes the robot's gripper by adjusting the target positions of the gripper joints.

Autonomous Mode

The CloseGripper method is used for autonomous operation as it controls the grippers programmatically, aligning with the robot's automated task execution rather than manual manipulation.

```
    /// <summary>
    ///     Close the gripper
    /// </summary>
    void CloseGripper()
    {
        var leftDrive = m_LeftGripper.xDrive;
        var rightDrive = m_RightGripper.xDrive;

        leftDrive.target = -0.01f;
        rightDrive.target = 0.01f;

        m_LeftGripper.xDrive = leftDrive;
        m_RightGripper.xDrive = rightDrive;
    }
```

OpenGripper Method: Opens the robot's gripper by adjusting the target positions of the gripper joints.

Autonomous Mode

The OpenGripper method, along with CloseGripper, is clearly used in an autonomous context where the robot's grippers are controlled programmatically based on the robot's task or state, rather than through manual user input.

```
    /// <summary>
    ///     Open the gripper
    /// </summary>
    void OpenGripper()
    {
        var leftDrive = m_LeftGripper.xDrive;
        var rightDrive = m_RightGripper.xDrive;

        leftDrive.target = 0.01f;
        rightDrive.target = -0.01f;

        m_LeftGripper.xDrive = leftDrive;
        m_RightGripper.xDrive = rightDrive;
    }

    /// <summary>
    ///     Get the current values of the robot's joint angles.
```

```
    /// </summary>
    /// <returns>NiryoMoveitJoints</returns>
```

CurrentJointConfig Method: Gets the current joint angles of the robot and returns them in a NiryoMoveitJointsMsg.

Autonomous Mode

The CurrentJointConfig method is used in an autonomous context. It programmatically retrieves and returns the robot's joint positions, which is a typical task in automated systems where the robot's state is monitored or controlled programmatically.

```
    NiryoMoveitJointsMsg CurrentJointConfig()
    {
        var joints = new NiryoMoveitJointsMsg();

        for (var i = 0; i < k_NumRobotJoints; i++)
        {
            joints.joints[i] = m_JointArticulationBodies[i].jointPosition[0];
        }

        return joints;
    }

    /// <summary>
    ///     Create a new MoverServiceRequest with the current values of the
robot's joint angles,
    ///     the target cube's current position and rotation, and the
targetPlacement position and rotation.
    ///     Call the MoverService using the ROSConnection and if a trajectory is
successfully planned,
    ///     execute the trajectories in a coroutine.
    /// </summary>
```

PublishJoints Method: Sends the current joint configuration and target poses to the ROS service for trajectory planning.

- **Create Request**: Creates a new request with the current joint angles and target positions.
- **Send Request**: Sends the request to the ROS service and specifies the `TrajectoryResponse` method as the callback.

TrajectoryResponse Method: Handles the response from the ROS service.

- **Check Trajectories**: Logs a message and starts executing the trajectories if they are returned.
- **Log Error**: Logs an error message if no trajectories are returned.

Autonomous Mode

The PublishJoints method is designed for autonomous operations. It involves sending commands to a ROS service to plan and execute robotic movements based on the current state of the robot and specified target positions. This is characteristic of autonomous control where actions are performed without direct manual intervention.

```
    void TrajectoryResponse(MoverServiceResponse response)
    {
        if (response.trajectories.Length > 0)
        {
            Debug.Log("Trajectory returned.");
            StartCoroutine(ExecuteTrajectories(response));
        }
```

```
        else
        {
            Debug.LogError("No trajectory returned from MoverService.");
        }
    }

    /// <summary>
    ///     Execute the returned trajectories from the MoverService.
    ///     The expectation is that the MoverService will return four trajectory
plans,
    ///     PreGrasp, Grasp, PickUp, and Place,
    ///     where each plan is an array of robot poses. A robot pose is the joint
angle values
    ///     of the six robot joints.
    ///     Executing a single trajectory will iterate through every robot pose
in the array while updating the
    ///     joint values on the robot.
    /// </summary>
    /// <param name="response"> MoverServiceResponse received from niryo_moveit
mover service running in ROS</param>
    /// <returns></returns>
```

ExecuteTrajectories Method: Executes the planned trajectories.

- **For Each Trajectory**: Loops through each trajectory plan returned from the ROS service.
- **For Each Pose**: Loops through each pose in the trajectory and sets the joint positions.
- **Wait**: Waits for the robot to move to each pose.
- **Grasp**: Closes the gripper if the current trajectory is for grasping the object.
- **Final Wait**: Waits after the final pose is achieved.
- **Open Gripper**: Opens the gripper after all trajectories are executed to release the object.

Autonomous Mode

The ExecuteTrajectories method is designed for autonomous operation. It automates the process of executing a series of pre-defined movements and actions based on the trajectory data received from the ROS service. It does not require manual input for each step, making it a part of an autonomous system where the robot performs tasks based on programmed instructions.

```
    IEnumerator ExecuteTrajectories(MoverServiceResponse response)
    {
        if (response.trajectories != null)
        {
            // For every trajectory plan returned
            for (var poseIndex = 0; poseIndex < response.trajectories.Length;
poseIndex++)
            {
                // For every robot pose in trajectory plan
                foreach (var t in
response.trajectories[poseIndex].joint_trajectory.points)
                {
                    var jointPositions = t.positions;
                    var result = jointPositions.Select(r => (float)r *
Mathf.Rad2Deg).ToArray();

                    // Set the joint values for every joint
                    for (var joint = 0; joint < m_JointArticulationBodies.Length;
joint++)
                    {
                        var joint1XDrive =
m_JointArticulationBodies[joint].xDrive;
                        joint1XDrive.target = result[joint];
                        m_JointArticulationBodies[joint].xDrive = joint1XDrive;
                    }

                    // Wait for robot to achieve pose for all joint assignments
                    yield return new WaitForSeconds(k_JointAssignmentWait);
                }

                // Close the gripper if completed executing the trajectory for
the Grasp pose
```

```
                if (poseIndex == (int)Poses.Grasp)
                {
                    CloseGripper();
                }

                // Wait for the robot to achieve the final pose from joint
assignment
                yield return new WaitForSeconds(k_PoseAssignmentWait);
            }

            // All trajectories have been executed, open the gripper to place the
target cube
            OpenGripper();
        }
    }
```

Enum Pose: Defines the different poses the robot will execute during the pick-and-place operation.

Autonomous Mode

This enum is used to manage different stages of an autonomous robotic task. The robot transitions through these poses to complete a specific sequence of actions autonomously, such as grasping, picking up, and placing an object.

```
   enum Poses
    {
        PreGrasp,
        Grasp,
        PickUp,
        Place
    }
}
```