CaRoSaC: Cable Robot Simulation and Control Framework

Rohit Dhakate

User Guide

Welcome to the **CaRoSaC: Cable Robot Simulation and Control Framework User Guide**. This document provides detailed instructions on installing, configuring, and utilizing the simulation aspect (CaRoSim) of CaRoSaC framework for suspended cable-driven parallel robots (CDPRs).

Contents

1 CaRoSaC Overview

The **CaRoSaC framework** offers a Unity3D-based simulation environment for suspended Cable-Driven Parallel Robots (CDPR) with flexible cables. It integrates a learning-based control approach, enabling precise manipulation of the CDPR system using cable lengths as control inputs.

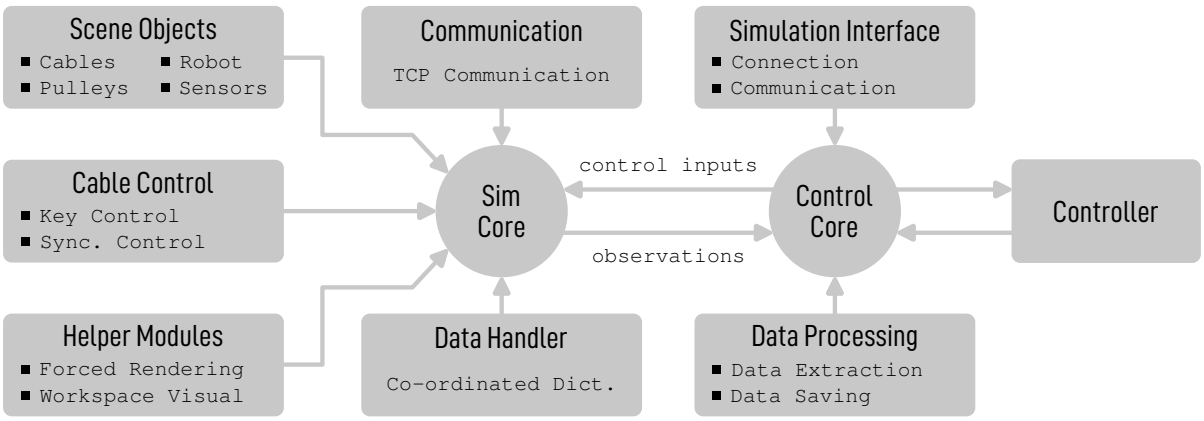

Figure 1: Architecture overview of the CaRoSaC framework, showing the main modules.

1.1 Key Features

- **Realistic Simulation Environment:** Unity3D-based simulation providing high-fidelity modeling of flexible cables and suspended CDPR dynamics.
- **Learning-Based Control:** Integration of reinforcement learning algorithms, such as TD3, for adaptive and intelligent control strategies.
- **Modular Architecture:** Clear separation of modules for simulation, control, and data handling, allowing flexibility in experimentation and scalability.
- **Cross-Platform Integration:** Seamless communication between Unity3D (simulation) and Python (control algorithms) using TCP communication.

1.2 Applications

- **Research and Development:** Ideal for prototyping and testing new state-estimation and control algorithms for CDPR systems.
- **Industrial Applications:** Can simulate real-world scenarios for tasks like crane operations, suspended load handling.

2 CaRoSim File Structure

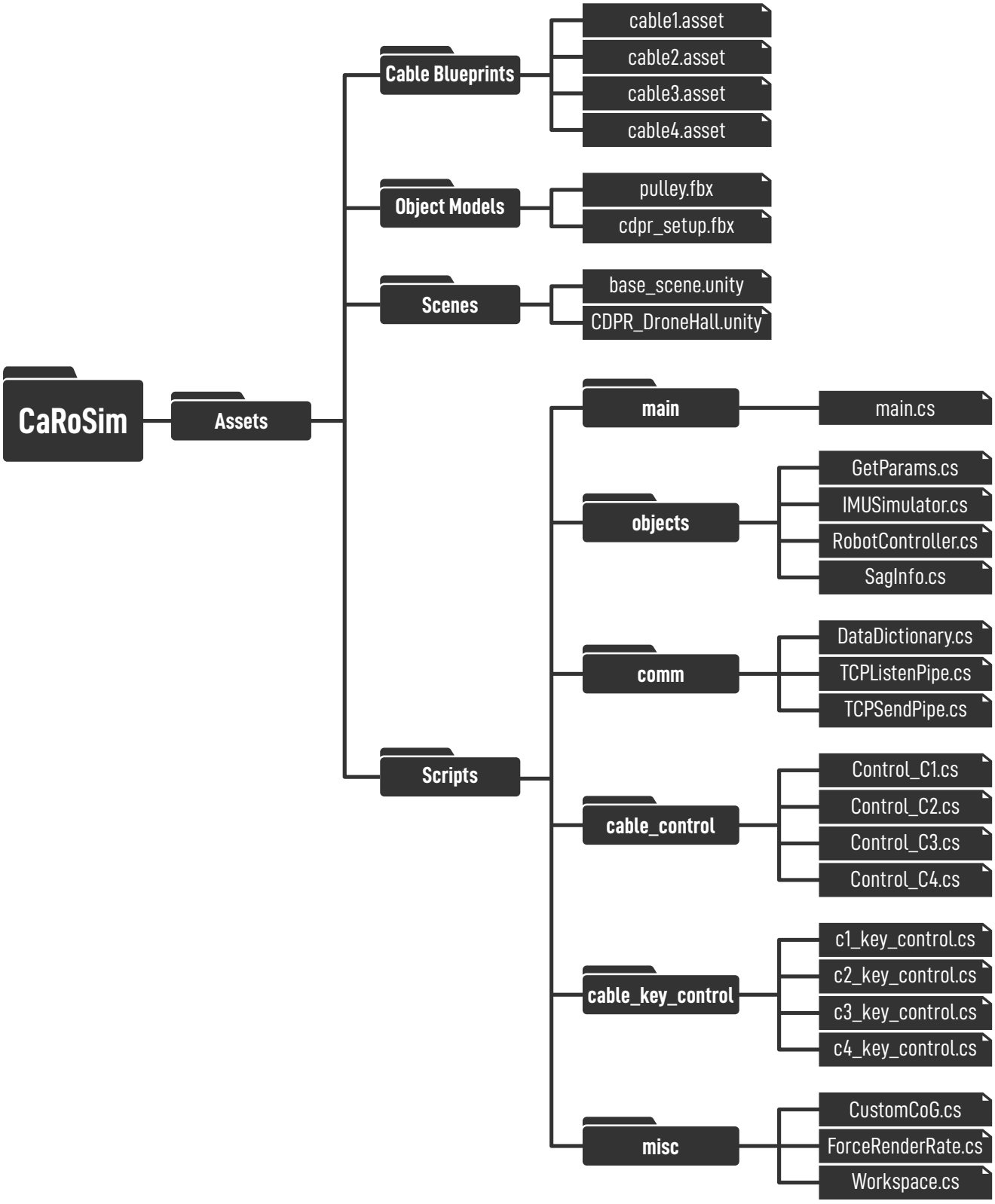

Figure 2: CaRoSim File Structure

3 CaRoSiM: Cable Robot Simulation

We incorporate a third-party package for flexible cable simulation named "Obi Rope" to setup our CDPR system in Unity3D. The current setup is simplified by avoiding wrapping cables over pulleys and just measuring cable lengths from cable exit points on the pulleys, however depending on the physical system configuration, it is possible to setup the simulation scene where cables are wrapped over the pulleys. The simulation's accuracy and reliability are validated by comparing the simulated results with real-world trajectories, demonstrating its practical applicability.

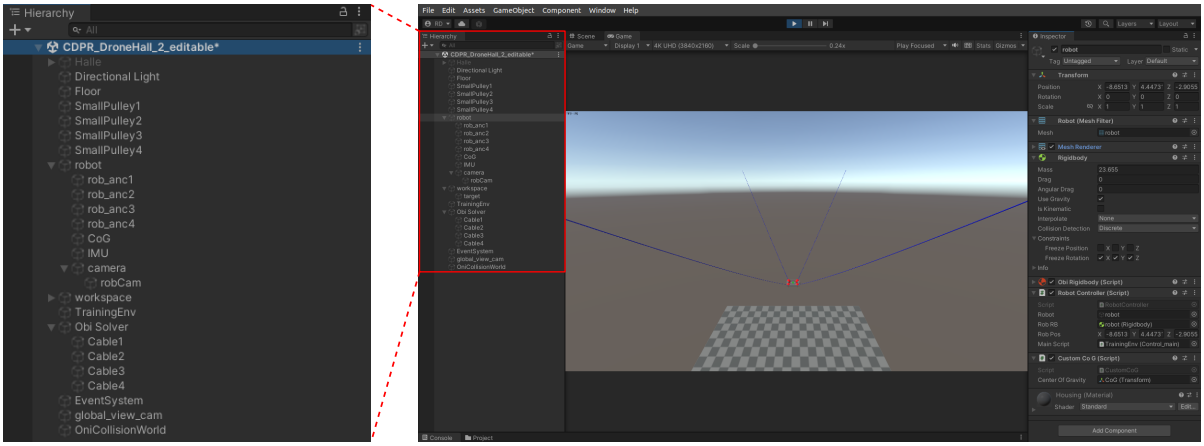

Figure 3: Suspended CDPR scene setup.

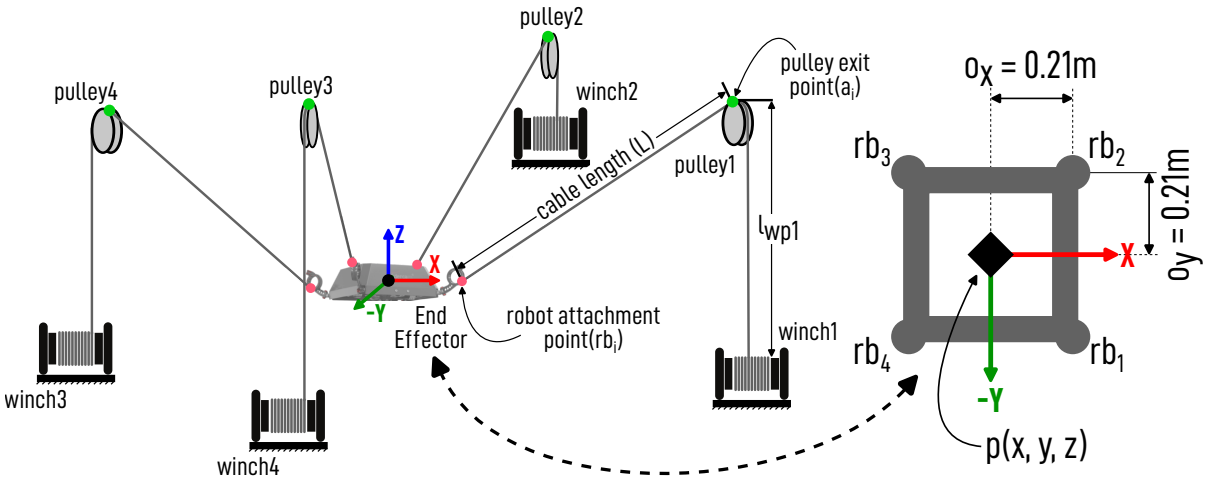

Figure 4: Schematic of our suspended CDPR setup.

3.1 General Information

3.1.1 Simulation Core

The core of the simulation is implemented in `Control_main.cs` script which serves as the heart of the CaRoSim, orchestrating the entire simulation workflow. It integrates key functionalities, including:

- Initialization and Data Management
- Communication Protocol
- Control and Execution
- Simulation State Management

3.1.2 Simulation Scene Objects

The simulation scene, consists of various components that are essential for the functionality of CaRoSim. Each object provides specific data or functionality as described below:

- ***Robot*:** Captures the robot's position, orientation, and velocity. The robot object also includes anchor points (`rob_anc1` to `rob_anc4`) for connecting cables. The `RobotController.cs` script accesses the robot's **position** (`robPos`), **orientation** (`robOriEul`), and computes **velocity parameters** using Unity3D's physics engine. A small script `CustomCoG.cs` is also attached to the object to modify the center of gravity as needed, depending on the attached payload. The data is continuously updated at simulation frame rate, ensuring real-time feedback. Additionally, the script provides hooks for line rendering to visualize the robot's trajectory, though this functionality is optional and can be toggled as needed.

- ***Cables*:** The cable objects (`Cable1 through Cable4`) provide real-time length measurements based on simulation dynamics. These are managed by the *Obi Solver*, which ensures realistic behavior under tension. The script `Control_C1.cs` (similar scripts exist for other cables) is responsible for controlling and updating the properties of Cable 1 during the simulation. The `Control_C1.cs` script integrates the *ObiRope* and *ObiRopeCursor* components to manipulate the cable's length dynamically. The current length (`rope_c1:length under forces`) and rest length (`rope_c1_rest:length under no forces`) of the cable are calculated every frame using the *ObiRope* component. The desired cable length (`c1_control_ip`) is received from external inputs via the `TCPListenPipe` and compared to the current length. The `SetCable1()` function, adjusts the cable length based on a variable velocity (`cable1Velocity`) while maintaining a threshold tolerance (`cable_threshold`). The cable length is adjusted using the `ChangeLength()` method of the *ObiRopeCursor* component. The script uses a time delta (`time_dt`) fetched from the main simulation script (`Control_main.cs`) to ensure synchronization across all cables. This setup enables precise and real-time control of the cable lengths in simulation, making it possible to mimic the physical behavior of CDPRs accurately while maintaining computational efficiency.

- ***Pulleys*:** The pulley objects in the scene serves as the anchor points from which the cable lengths are measured. In the current simulation setup the position and orientation of the pulleys are fixed and known. However, if using a swiveling pulley mechanism a simple script similar to `RobotController.cs` can be added to get current information on pulley positions and orientation during runtime.

- ***Target*:** Represents the position and orientation for a target object, used for learning a trajectory tracking control.

- ***IMU*:** Generates synthetic accelerometer and gyroscope data, simulating onboard sensor measurements. The IMU is placed at the center of the end-effector body. The sampling rate, standard deviation, biases and bias evolution rates are defined in `IMUSimulator.cs` and can be modified as needed.

- ***Workspace*:** The workspace object in the simulation defines and visualizes the operational area for the robot. This area is used for training or validating controllers within specific boundaries. The script `Workspace.cs` manages the workspace visualization and functionality.

3.1.3 Communication Protocol

The simulation employs a robust TCP/IP-based communication protocol for seamless interaction between Unity3D and external controllers (e.g., Python scripts). The communication system is implemented through two core components: `TCPListenPipe.cs` and `TCPSendPipe.cs`.

- ***TCPListenPipe.cs*:** This script acts as a TCP server, listening for incoming data from the external controller. It parses incoming JSON data to extract control commands, such as desired cable lengths (`c1_action`, `c2_action`, etc.), velocities, and target positions (`goal_pos`) and updates the simulation state based on received values, ensuring real-time responsiveness. It also manages the connection lifecycle, initializing the listener on startup and safely closing connections on application exit.

- ***TCPSendPipe.cs*:** This script functions as a TCP client, transmitting simulation data back to the external controller. It sends processed simulation data, such as cable lengths and robot positions, in a byte-encoded format while ensuring a reliable connection by attempting to re-establish communication if the socket is disconnected. It also handles the connection lifecycle, ensuring proper closure of the socket upon application exit.

This protocol enables efficient, bidirectional communication between the simulation and external systems, allowing for real-time control and monitoring of the CDPR setup.

3.1.4 Data Handling

The simulation employs an efficient data management system to handle real-time updates and interactions across various components. This functionality is implemented through two key scripts: `DataDictionary.cs` and `GetParams.cs`.

- ***DataDictionary.cs*:** The `DataDictionary.cs` script serves as the foundation for managing simulation data by initializing and maintaining a flexible data dictionary. This dictionary stores key variables such as robot position, cable lengths, target positions, timestamps, and simulation status. It provides dynamic updates through the `UpdateDataDict()` method, which accepts key-value pairs and efficiently modifies the dictionary's contents. This modular approach ensures that the data structure can adapt to the evolving needs of the simulation while maintaining clarity and organization.
- ***GetParams.cs*:** The `GetParams.cs` script is responsible for retrieving and organizing parameters from various simulation components, such as the robot, cables, and target. It defines structured parameter classes, including `RobotParams`, `CableParams`, and `ControlInputs`, to logically categorize and store data. The `GetAllParameters()` method consolidates data into a unified `Parameters` object, enabling consistent access to all relevant variables across the simulation. This structured approach streamlines the management and retrieval of data, ensuring seamless integration and real-time updates within the simulation framework.

This modular data handling framework ensures seamless data flow and integration between simulation components, enabling real-time updates and precise control for the CDPR simulation.

3.1.5 Observable Data

The simulation collects a variety of observable data from its components, which are organized into structured parameter classes within the `GetParams.cs` script. This data is essential for monitoring, controlling, and analyzing the system's performance. Key observable data includes:

- ***Robot Data*:** The robot's current position and orientation are captured as `RobotParams`, which are updated in real time from the `RobotController` script.
- ***Cable Data*:** Each cable's current length and rest length are recorded under the `CableParams` class. This data is updated dynamically during the simulation to reflect real-time cable behavior.
- ***Control Inputs*:** The desired control inputs for each cable, including their lengths (`c1_control_ip`, `c2_control_ip`, etc.), are stored in the `ControlInputs` class. These values are received from the external controller via the `TCPListenPipe` script.
- ***Cable Velocities*:** The current velocity of each cable (`c1_vel`, `c2_vel`, etc.) is captured under the `CableVelocity` class, enabling precise control and validation of cable adjustments.
- ***Target Position*:** The desired goal position of the robot is represented as `TargetPos`, providing a reference for trajectory tracking.

- ***Sag Information*:** Sag-related metrics (`sag_info_1`, `sag_info_2`, etc.) for each cable are recorded in the `Sag` class to evaluate cable sag. The `sag_info_i` is basically the straight-line distance from the pulley exit points to the robot attachment points. This information alongside the actual current length obtained from the simulation can be utilized to quantify the sag behavior.

- ***Unique Data Identifier*:** A unique identifier (`UID`) for each data instance is stored in the `Data_UID` class, allowing for easy tracking of simulation states.

This comprehensive and structured organization of observable data ensures that all relevant variables are readily accessible and consistently updated for efficient monitoring and control of the Cable-Driven Parallel Robot (CDPR) simulation.

3.1.6 Appending Additional Data

The CaRoSaC framework is designed to be flexible and extensible. If you need to append additional observable data to the existing dataset, modifications are required both in the Unity3D and Python scripts to ensure seamless data integration.

- ***Modifications in Unity3D:*** To add new observable data, navigate to the `init_dict()` method in `Control_main.cs` and add the new data keys to the `keys` list.

```
List<string> keys = new List<string>{
    "frame_num", "timestamp", "scene_dt",
    "robPos_x", "robPos_y", "robPos_z",
    "robOri_x", "robOri_y", "robOri_z",
    "goalPos_x", "goalPos_y", "goalPos_z",
    "Cable1", "Cable2", "Cable3", "Cable4",
    "Cable1_rest", "Cable2_rest", "Cable3_rest", "Cable4_rest
       ",
    "sag_info_1", "sag_info_2", "sag_info_3", "sag_info_4",
    "robLinVel_x" // New Data Key Added
    };
```

  To ensures the new data is updated during each simulation frame. In the `update_dict()` method, add:

```
("robLinVel_x", RobotParams.LinearVelocity.x)
```

- ***Modifications in Python:*** The indexing and data extraction are managed automatically based on the order and length defined in the `data_struct.yml` file. Define the new data entry with its length in `data_struct.yml`. Ensure the order matches the Unity3D's data structure to maintain correct indexing.

```
robLinVel:
  len: 3
```

3.2 Setting up custom Unity3D scene

This section guides you through the process of creating a custom suspended CDPR simulation scene using Unity3D integrated with the Obi Rope asset. Although this repository includes pre-configured scenes for suspended CDPRs with four actuators, setting up new configurations with different parameters is straightforward. A CDPR setup typically requires the following components:

- **Robot (End-Effector) Model**
- **Pulley Positions and Orientations**
- **Number of Cables (Actuators)**
- **Cable Properties**

3.2.1 End-Effector Setup

First, we need to configure our end-effector (assuming a 3D model is available) by adding the end-effector object to the scene.

- ***Add the End-Effector Model***: Add your 3D model of the end-effector into the Unity3D scene.
- ***Physics Configuration***: Attach a Rigidbody component to make the object dynamic and add the Obi Rigidbody component for Obi Rope integration.
- ***Attach Associated Scripts***: Add `RobotController.cs` to manage position, orientation, and control logic. Add `CustomCoG.cs` to modify the center of gravity, if needed, based on payload adjustments.

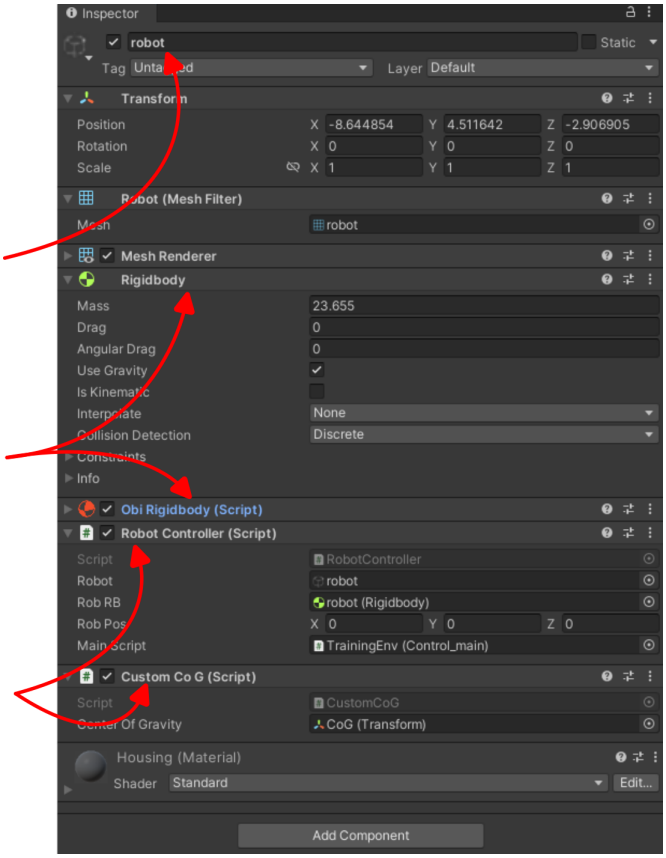

Figure 5: End-effector setup.

3.2.2 Pulley Setup

- ***Adding Pulleys***: Create pulley objects (e.g., simple spheres or custom models) and position them according to your desired CDPR setup.
- ***Fixed or Dynamic Pulleys***: For static setups, pulley positions can be set to fixed. For dynamic (swiveling) pulleys, using a `Hinge Joint` will allow the pulley to rotate freely around a specified axis, typically the vertical axis, to track their orientation in real-time.

3.2.3 Cable Setup

Cable blueprints determine the physical and simulation properties of each cable. In order to add cables to the scene, we first need to define individual blueprint for each cable.

- ***Create Cable Blueprint***: Cable blueprints can be defined from the toolbar as, `Assets -> Create -> Obi -> Rope blueprint`
- ***Key Blueprint Parameters***: The cable blueprints in Obi Rope are defined by the following parameters:
  - ***Thickness***: Defines the radius of particles forming the cable (in meters), affecting the overall cable diameter.
  - ***Resolution***: Controls particle density along the cable length.
    `1 = Overlapping particles (denser, robust for collisions).`
    `0.5 = Particles just touching (balanced performance).`
    `< 0.5 = Gaps between particles (less accurate but faster)`
    `0 = One particle per control point.`
  - ***Pooled Particles***: Allocates extra particles for dynamic changes like tearing or resizing. Set to 0 if such dynamics are not needed.
- ***Adding Cables to the Scene***: To simulate cable dynamics accurately, cables need to be properly added and configured in the scene, linking the end-effector to the pulley anchor points. A cable object can be added to the scene as,
  `GameObject -> 3D Object -> Obi -> Obi Rope`
  - ***Attach Blueprints to Cables***: Select the rope object and assign the respective created blueprint in the Inspector panel for each cable.
  - ***Attach Cable Control Scripts***: Attach the control scripts to each cable, `Control_C#` (`# = 1 to 4`).
  - ***Attach Cables to Anchor Points***: Define start and end points for the rope using `Obi Particle Attachment`.

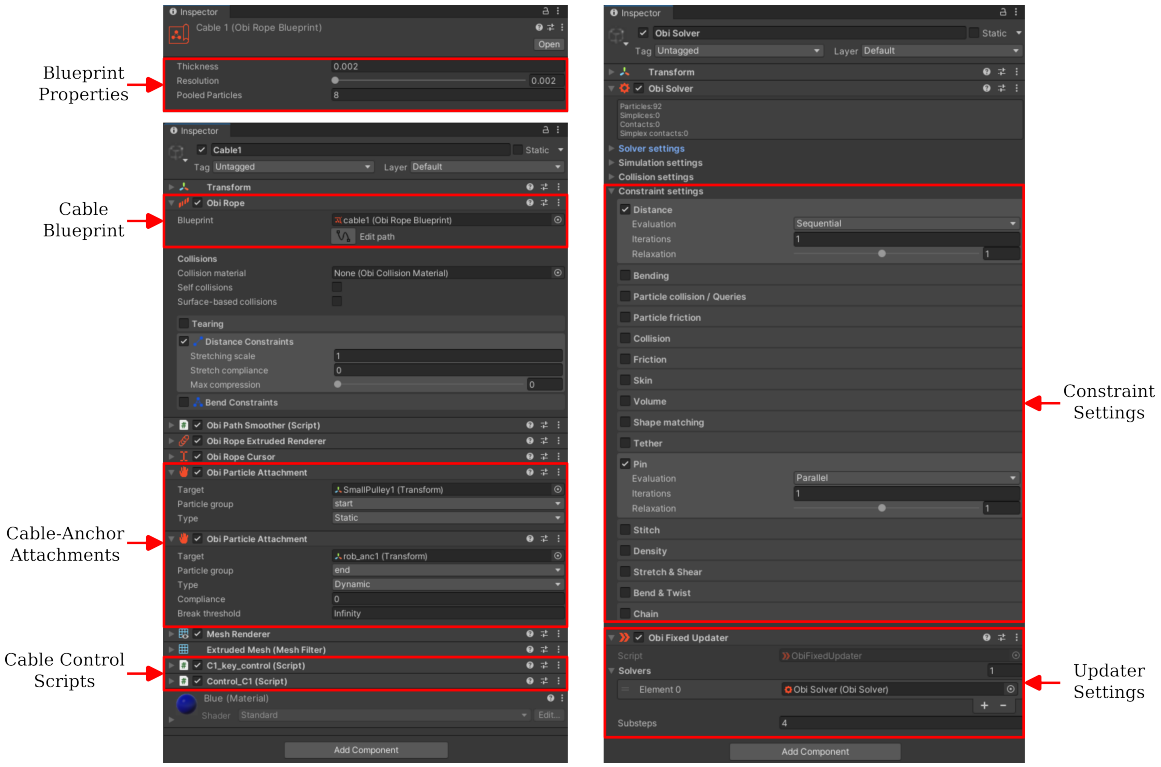

Figure 6: Cable Blueprint and Cable Object settings (`left`), Obi Solver settings (`right`).

3.2.4 Obi Solver Parameters

The Obi Solver manages the physics calculations for cables and their constraints.

- ***Add Obi Solver***: Add an Obi Solver to the simulation scene as,

  `GameObject → 3D Object → Obi → Obi Solver`

- ***Configuring Solver Settings***:
  - *Substeps*: Controls the number of simulation substeps per frame, improving accuracy at the cost of performance.
  - *Iterations*: Determines how many times the solver attempts to resolve constraints within each substep.

- ***Connecting Cables to the Solver***: Select each cable object and assign it to the Obi Solver in the Inspector panel to ensure it is processed during simulation.

- ***Performance Considerations***: The performance difference between using a single solver versus multiple solvers is negligible as long as they are managed by the same ObiUpdater component. All solvers are updated in parallel, maintaining high efficiency.

  However, assigning separate updaters to different solvers results in sequential updates, significantly reducing performance. Hence, it is recommended to use a single updater for all solvers.

To simulate real cable behavior, multiple constraints can be combined to capture both tensile and bending properties:

- ***Distance Constraints***:
  - Maintain particle spacing, controlling stretch and compression.
  - Adjust Stretch Compliance to match tensile stiffness (inverse of Young's modulus).
- ***Bending Constraints***:
  - Control cable curvature under load.
  - Use Max Bending and Bend Compliance to replicate real-world flexibility.
- ***Pin Constraints***:
  - Fix cable ends to anchor points for accurate boundary conditions.
- ***Fine-Tuning Parameters***:
  - Substeps: Increase for stability under high loads.
  - Iterations: Boost for more accurate constraint resolution.

3.3 Running CaRoSiM

Once the simulation scene is set up and configured according to the real system, you can proceed with running CaRoSiM to evaluate the behavior of the CDPR under different conditions.

3.3.1 Starting the Simulation

- ***Save the Scene***: Before running, ensure all changes are saved.
- ***Enter Play Mode***: Press the Play button in Unity3D's toolbar to start the simulation. This initializes the Obi Solver and begins real-time physics processing.

3.3.2 Real-Time Monitoring in Unity3D

- ***Observe Cable Dynamics***: Check for realistic cable tension, sagging, or slack behavior.
- ***Monitor Robot Movement***: Track the end-effector's position, orientation, and responses to control inputs.

3.3.3 Data Transfer to Python Client

- ***Unity3D → Python Data Flow***: Unity3D sends simulation data (e.g., cable lengths, robot position, orientation) via a TCP socket. The `sock_comm.py` and `get_data.py` scripts handle incoming data on the Python side.
- ***Python Data Processing***: The `data_processing.py` script parses the received data based on a predefined structure `data_struct.yml`.
- ***Control Feedback to Unity3D***: Python can send control commands back to Unity3D using the `send_data.py` script. Commands include desired cable lengths.
- ***Data Logging***: All received data can be saved as CSV files for later analysis. Data logging includes simulation timestamps, control inputs, and system states.

3.3.4 Stopping the Simulation

Click the Play button again to exit Play Mode.