GamesmanROS: A Generalized Game-Playing Robotic System

Nakul Srikanth
Dan Garcia, Ed.
Ken Goldberg, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-128
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-128.html

May 22, 2025

Acknowledgement

To my teachers, friends and family.

GamesmanROS: A Generalized Game-Playing Robotic System

by

Nakul Srikanth

A report submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering & Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dan Garcia, Chair
Professor Ken Goldberg

Spring 2025

GamesmanROS: A Generalized Game-Playing Robotic System

Abstract

GamesmanROS: A Generalized Game-Playing Robotic System

by

Nakul Srikanth

Masters of Science in Electrical Engineering & Computer Science

University of California, Berkeley

Professor Dan Garcia, Chair

GamesCrafters is a research and development group at UC Berkeley focused on combinatorial game theory and puzzles. Over the past 25 years, it has engaged more than 750 students in the strong solving of abstract strategy games and puzzles through exhaustive search. The group also provides tools for analysis, visualization, and perfect play. This report details the design and implementation of GamesmanROS—an autonomous, game-playing, 6-degree-of-freedom robot that leverages these strongly solved solutions to play over 22 board games and puzzles.

Acknowledgements

My deepest gratitude to Prof. Dan Garcia for being a friend, companion, and the best mentor that I could have ever asked for during my time at Berkeley. From meeting him before my first class at UC Berkeley in the VLSB courtyard, to spending five semesters as a Gamescrafter under his mentorship, I truly cherish his heartfelt jokes, passionate conversations, and his infamous rap songs about computer science. His radiating enthusiasm and love for games is forever an inspiration that I will carry in my heart. GamesCrafters, woohoo!

My sincerest gratitude to Prof. Claire Tomlin and her PhD student, Kaylene Stocking, whose support led me to pursue graduate studies with the EECS 5th Year MS program at Berkeley.

Thanks to Prof. Ken Goldberg for being a gracious second reader and warm supporter of my endeavors in robotics and game theory.

I want to acknowledge my high school robotics teacher, Joseph Alvarico, who first sparked my interest in robotics and motivated me to lead a sub-team of programmers in our rookie year of FIRST Robotics. Winning the rookie-all-star award in the SF regionals and leading the way to Houston, TX, for the World Championships opened my eyes to the world of robotics. This core memory is forever a motivation in my pursuit of advancing robotic applications in academia and industry.

Special thanks to Miller James Hollinger for his assistance in designing and printing the physical game boards and pieces, as described in Section 3.5 of this report.

Most important of all, I want to thank my parents, Srikanth & Nirmala, for their love, support, and sacrifice. Deepest gratitude to my grandmother, whose unconditional love has been a constant source of motivation. I also cannot forget my little sister, Dhruvi, for all the board games we played growing up, which eventually inspired me to try GamesCrafters during my time at Berkeley.

Table of Contents

List of Figures

Chapter 1

Background

1.1 GamesCrafters

GamesCrafters is a combinatorial game theory and puzzle research and development group founded by Teaching Professor Dan Garcia in 2001, as a continuation of his "GAMESMAN" Master's project, a finite two-player perfect information game generator. Over 25+ years and 750+ undergraduate and graduate students later, the group has "strongly" solved 70+ games and 15+ puzzles. In addition, the group has created advanced analysis and visualization techniques, all of which are made available for public use. Over recent years, they have been motivated to strongly solve larger games like Connect 4, Quarto, and Othello. This approach has led to many optimized solving algorithms that are parallelized and utilize cloud computing. This project brings these games and puzzles to life, enabling a physical experience while leveraging our existing analysis and visualization techniques.

1.2 A Short Introduction to Game Theory

Combinatorial Game Theory (CGT) is a branch of mathematics that analyzes two-player, turn-based games with perfect information and no chance elements. In CGT, every game position (a state of the board configuration along with whose turn it is) is categorized based on its outcome under optimal play called Position Value, as visualized in *Figure 1.1*. Let's break down each possible position value:

- **Win**: The current player can always force a victory regardless of the opponent's moves.
- **Lose**: The current player will lose if the opponent plays optimally.
- **Tie**: Neither player can force a win; the game ends in a stalemate.
- **Draw**: Neither player can force a win; the game in perfect play will go on forever.

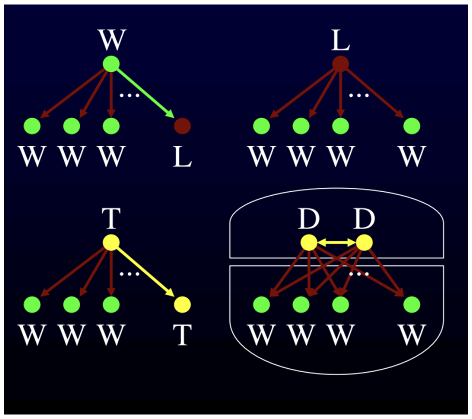


Figure 1.1: Visualization for Position and Move Values (courtesy Dan Garcia)

An essential concept in CGT is **remoteness**, which measures the distance (in moves) from a given position to the game's end position under optimal play. A lower remoteness indicates a quicker path to the game's conclusion.

1.3 GAMESMAN System Architecture

GAMESMAN is a comprehensive software framework designed to solve and analyze two-player, perfect-information combinatorial games. It supports the computation of strongly solved solutions, wherein the entire game tree is exhaustively explored to determine the optimal strategy for every possible position. The system offers interactive tools for visualizing position and move values during gameplay, as well as post-game analysis through a visual value history. To accommodate the increasing complexity and scale of supported games, GAMESMAN employs a generalized distributed architecture, as shown in *Figure 1.2*, that facilitates efficient scaling across all stages, from game solving to visualization. This architecture comprises three primary components: GamesmanClassic, GamesCraftersUWAPI, and GamesmanUNI.

In this section, we briefly introduce each component and its relation to Gamesman, and hopefully contextualize properties that we leverage for GamesmanROS.

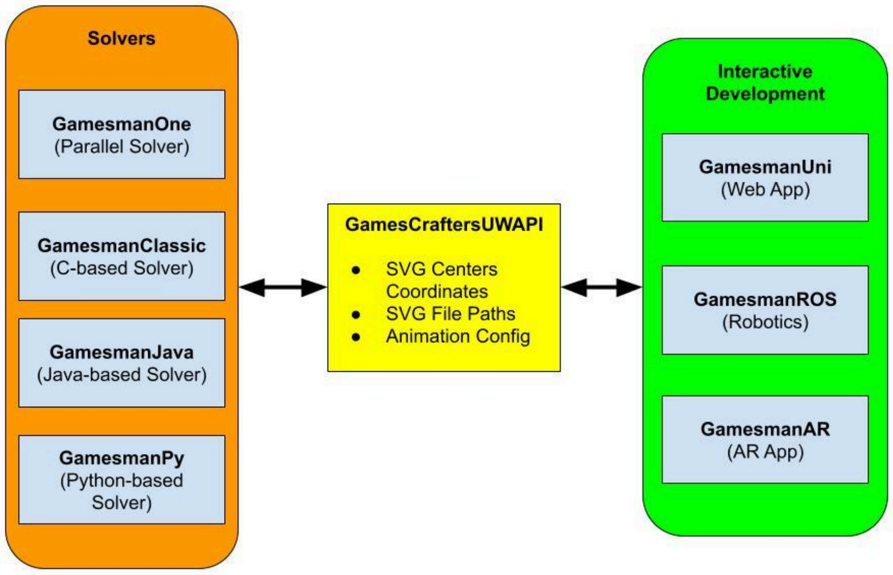


Figure 1.2: System Architecture for GAMESMAN

1.3.1 GamesmanClassic (Game Server)

GamesmanClassic, the original game-solving engine in our system, has been written and actively maintained in the C programming language for over two decades. It provides a robust framework for encoding combinatorial games, where a user implements the game's logic and rules in C. Once defined, the system outputs a *strong solution* by exhaustively traversing the game's state space using a built-in search algorithm. This traversal computes the exact position value (e.g., win, lose, tie) and remoteness (i.e., number of moves to endgame under optimal play) for every possible game state, and stores these results in a solution database.

GamesmanClassic also features a TUI (Text-based User Interface) that allows users to interactively explore these values, analyze strategies, and visualize the solved state space. Additionally, it can be launched in *server mode* via a command-line argument, enabling it to handle web requests from a public API. This API allows external clients to query the value and remoteness of a given position in a particular game, making it a powerful backend for web-based game analysis tools.

1.3.2 GamesCraftersUWAPI (AutoGUI Web Server)

GamesCraftersUWAPI (Gamescrafters Universal Web API), our official Automatic Graphical User Interface (AutoGUI) middleware, hosts all of our metadata used to graphically display our games on the frontend. This Universal software interacts with games written and solved in various languages like C (GamesmanClassic), Java (GamesmanJava), and Python (GamesmanPy) to translate requests and responses for corresponding positions and moves data to interact with the frontend web application.

This software hosts the center coordinates for the Scalable Vector Graphics (SVGs) used by the frontend to pieces on a game board. It also includes file paths for the game board and game pieces, both of which are in SVG format. It also stores the file path for animation and sound effects. This server is modified every time a user wants to add AutoGUI capabilities for a game already solved in GamesmanClassic or any of our other game servers.

1.3.3 GamesmanUNI (Frontend Web Application)

GamesmanUNI, our official web-based graphical user interface (GUI), serves as an interactive platform for engaging with solved games and visualizing analytical results. Developed using Vue.js, TypeScript, and HTML, this application interfaces with the underlying GamesCraftersUWAPI to dynamically retrieve and display game state data, ensuring a seamless and responsive user experience. See *Figure 1.3* that depicts the landing page for games.

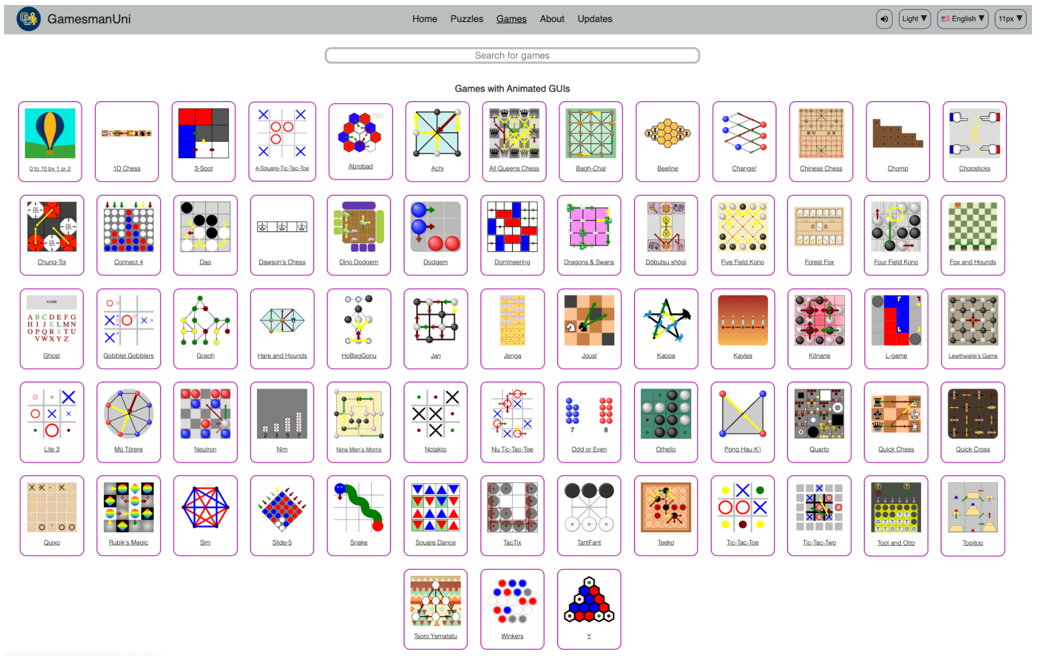


Figure 1.3: GamesmanUni - Frontend landing page for games

Chapter 2

Previous Work

To contextualize our contributions, we begin by reviewing prior work that has addressed similar challenges in robotic board gameplay. Existing approaches include table-top magnetic robots that manipulate game pieces using magnets, as well as nano-drones that physically represent and move as game elements. While several efforts have explored the generalization of board games through robotics, few have leveraged a 6-DOF (6 Degrees Of Freedom) robotic arm to perform these tasks. Among those that do, the use of 6-DOF manipulators is typically coupled with machine learning models trained on specific game configurations, with reinforcement learning (RL) agents often employed to determine the next move. However, such agents typically do not guarantee optimal decision-making, limiting their applicability in games requiring deterministic or perfect play.

One project that is looking to generalize board games is Square Off's Neo and Swap systems that represent a commercial implementation of AI-powered, physical board games. [1] Neo focuses exclusively on chess, featuring a magnetic 2D movement system beneath the board that silently and precisely moves magnetic pieces in response to AI or online opponents. Swap builds on this concept by supporting multiple games—including Draughts, Connect 4, and Chinese Checkers—through interchangeable piece sets and game modes. Both boards are globally connected, allowing remote play against human or AI opponents, and feature adaptive difficulty settings. Notably, the platforms incorporate an interactive virtual coach, "Viktor," which provides personalized training to improve player performance. The systems showcase how physical gameplay can be enhanced through robotics, AI, and user-centric design, bridging the gap between digital convenience and tactile engagement.

One notable example of recent work that explores embodied game-playing agents is SwarmPlay, a system that reimagines board gameplay through a swarm of autonomous drones. [2] The system was demonstrated using the classic game of Tic-tac-toe, in which drones autonomously detect game state changes via an overhead computer vision system and collaboratively execute strategic moves on a physical board. Unlike traditional single-robot approaches, SwarmPlay distributes agency across multiple UAVs, offering richer interactivity and real-time game engagement. A custom decision-making algorithm, based on an improved version of the Basic Algorithm, balances offensive and defensive strategies to create a fair challenge for human players. User studies confirmed the system's perceived responsiveness, minimal cognitive fatigue, and strong engagement—69% of participants rated engagement at the maximum level. Beyond Tic-tac-toe, surveyed participants expressed enthusiasm for drone-based adaptations of more complex games such as Billiards, Battleship, and Tetris, suggesting that SwarmPlay's approach could be generalized to other dynamic, multi-agent game environments.

One project designed a novel pipeline that enabled a 6-DOF general-purpose robotic arm (PAPRAS) to play chess autonomously against a human using a monocular camera attached to the robot's end-effector [3]. The system operates under tight constraints, such as a limited field of view and workspace, and avoids reliance on external sensors or overhead cameras. They introduce a quadrant-based scanning strategy to observe the board, a hand-tracking module using Google's MediaPipe for opponent move detection, and a perception pipeline involving 2D detection, 3D pose estimation, and chess move prediction. Integration with the Stockfish engine facilitates competitive gameplay, and experiments show strong perception (92.16% accuracy) and manipulation (91.94% success rate) capabilities.

Another project introduces a robust and general-purpose robot capable of playing chess against humans using uninstrumented boards and arbitrary pieces. [4] Their system combines a custom 6-DoF manipulator, a depth-sensing Kinect-style camera, and a palm-mounted camera to perceive, track, and manipulate chess pieces. It employs a hierarchical classification system for chess piece recognition and uses visual servoing to adjust grasps in real time, making the system highly adaptable to non-ideal conditions. It also features a natural language interface for human interaction, and performs complete perception-planning-action loops without requiring structured environments. Experimental evaluation shows a high success rate (91.6% manipulation success) and promising grasp quality improvements due to visual servoing (from 17.5% to 77.5% success in worst-case trials).

Researchers in New Zealand developed a robotic system capable of playing the board game *Trax* autonomously against a human opponent in a natural tabletop setting. [5] Rather than relying on traditional mouse or keyboard inputs, the robot interacts exclusively through real-world manipulation and vision-based sensing. The system comprises four main components: a game engine that includes pattern recognition and threat evaluation, a camera-based vision system for state sensing and move validation, a 4-DOF manipulator mounted on a gantry for piece placement, and a coordination module that handles human-robot interaction phases like negotiation, gameplay, and game conclusion. The authors place strong emphasis on ensuring intuitive and seamless interaction, with the robot passively responding to human actions and using audio cues for feedback.

Chapter 3

GamesmanROS: Hardware Interface

3.1 Overview & Setup

For this project, we are using the MyCobot280pi, a 6-DOF robot arm from Elephant Robotics driven by a Raspberry Pi, to enable simple pick-and-place of game pieces for autonomous robot game play. [6] This robot comes with a standard gripper and a monocular usb camera with pegs for attachment to the robot's end-effector, as shown in *Figure 3.1*.

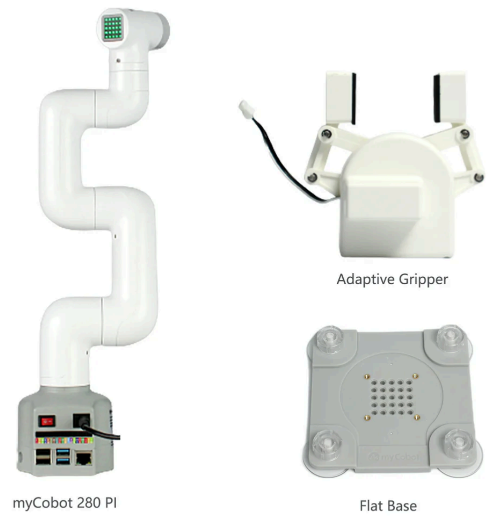


Figure 3.1: MyCobot280pi kit unboxing

To configure this robot, we had to first set up networking configurations and remote access to enable access to the robot’s hardware. We then set up the robot’s sensors, motors, and camera to send a stream of messages. Later, we configured trajectory planners and inverse kinematics to enable precise movement of the robot’s end-effector from point A to point B, and finally, we integrated a computer vision system using the camera to enable detection of the game board and its pieces.

3.1.1 Remote SSH

To facilitate remote development and interaction with the robot from any compatible device, a remote access methodology was implemented, targeting the robot's onboard Raspberry Pi. The initial step involves identifying the robot's IP address, which requires direct access to its graphical user interface (GUI). This is achieved by temporarily connecting a monitor to the Raspberry Pi. A USB-connected keyboard and mouse are then used to navigate the interface and establish a connection to the local Wi-Fi network.

This network configuration is a one-time setup, assuming continued use of the same Wi-Fi network. Once connected, the Raspberry Pi retains the network credentials and will automatically reconnect to the known network on subsequent startups. As a result, future interactions do not require a keyboard, although a monitor and mouse remain necessary until a static IP address configuration is successfully implemented for the Berkeley eduroam Wi-Fi network.

After the Wi-Fi connection is established, the robot's current IP address can be retrieved via the network configuration panel within the desktop environment. This IP address is then used to enable secure remote access to the Raspberry Pi from external development machines. The following figures demonstrate the process.

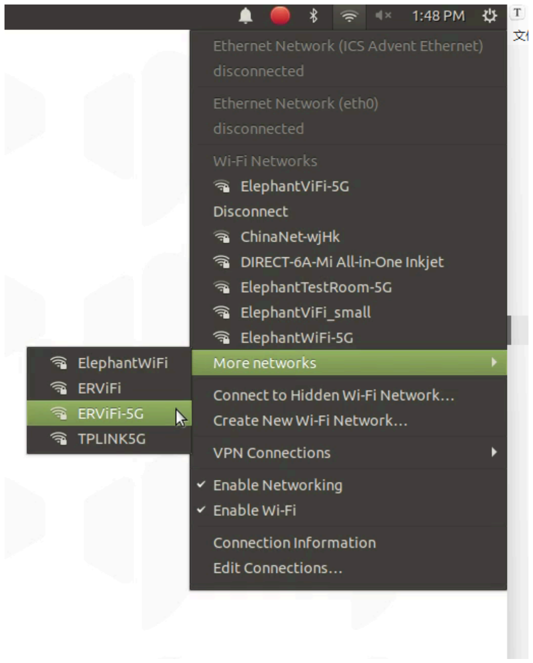
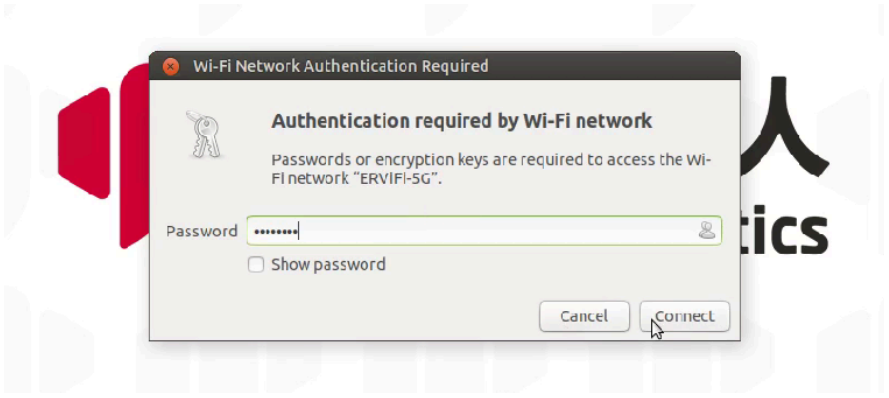


Figure 3.2: Step 1: Navigate to the Networks panel, enable Wi-Fi, and select network


Wi-Fi Network Authentication Required

Authentication required by Wi-Fi network

Passwords or encryption keys are required to access the Wi-Fi network "ERVIFI-5G".

Password ••••••••

☐ Show password

Figure 3.3: Step 2: One-time authentication of your Wi-Fi network

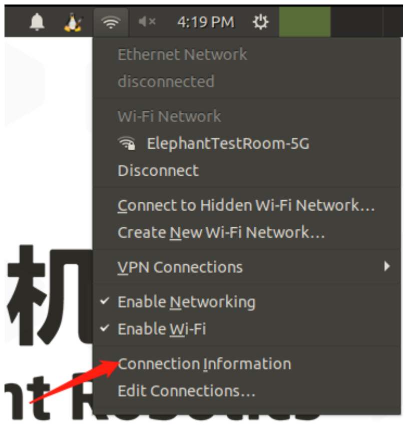
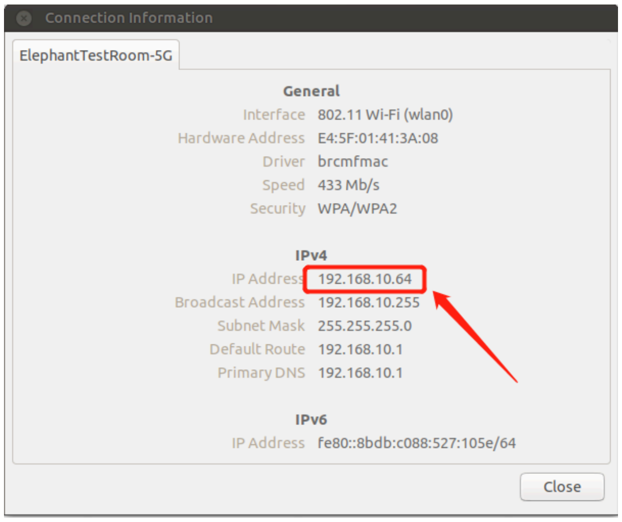


Figure 3.4: Step 3: Navigate back to the Networks panel and select "Connection Information"



Connection Information

ElephantTestRoom-5G

General

Interface	802.11 Wi-Fi (wlan0)
Hardware Address	E4:5F:01:41:3A:08
Driver	brcmfmac
Speed	433 Mb/s
Security	WPA/WPA2

IPv4

IP Address	192.168.10.64
Broadcast Address	192.168.10.255
Subnet Mask	255.255.255.0
Default Route	192.168.10.1
Primary DNS	192.168.10.1

IPv6

IP Address	fe80::8bdb:c088:527:105e/64

Close

Figure 3.5: Step 4: The highlighted field is the robot's IP address

With the robot's IP address, we can now remotely access the robot from any laptop or computer device on the same network. Code development, execution, and real-time data visualization can also be performed from any device connected to the same network as the robot. This approach is preferred due to the limited computational resources of the Raspberry Pi, which may struggle to handle resource-intensive tasks such as web browsing or graphical user interfaces concurrently with trajectory planning and motor control algorithms. Offloading these tasks to more powerful remote machines improves overall system performance and ensures reliable robot operation.

3.1.1 ROS Networking

Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications [7]. A running ROS system can comprise dozens, even hundreds of nodes, spread across multiple machines. Depending on how the system is configured, any node may need to communicate with any other node at any time.

To facilitate data visualization on local development machines from the robot's onboard Raspberry Pi, it is necessary to configure ROS networking protocols consistently across all devices. Proper networking ensures that all systems within the distributed ROS environment can communicate effectively, enabling synchronized access to sensor data, path planning modules, and other critical information. This configuration is especially crucial when multiple remote SSH sessions are required for running concurrent ROS nodes or command-line tools. Without a correctly configured ROS network, data sharing and inter-process communication between nodes may fail, impairing both development and real-time robot operation. Therefore, establishing robust and consistent ROS networking is a foundational requirement for successful robot programming and deployment.

Here are two snippets of bash scripts to be added to the respective bash files for the robot and laptop.

```bash
source /opt/ros/noetic/setup.bash
export ROS_MASTER_URI=http://localhost:11311
export ROS_IP=$(hostname -I | awk '{print $1}')
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/ros/noetic/lib
export PATH=/usr/lib/ccache:$PATH
```

Figure 3.6: ~/.bashrc file from robot

```bash
# export ROS_MASTER_URI=Robot_IP:11311
# export ROS_IP=Laptop_IP
export ROS_MASTER_URI=http://10.40.192.41:11311
export ROS_IP=10.40.222.157
```

Figure 3.7: ~/.bashrc file from laptop or local device

Now we are ready to set up the robot's sensor information from its joints and camera, in addition to enabling trajectory, path planning, and integration of the computer vision system.

3.2 Robot Internal Communication

The robot is equipped with joint servos that include integrated encoders, which provide real-time feedback on the joint angles of the 6-DOF robotic arm. This data is critical for multiple software modules that perform tasks such as state estimation, visualization, and control. Additionally, the robot supports command-based actuation, allowing external software to update joint positions. These functionalities often require integration with various components within the ROS (Robot Operating System) ecosystem. To facilitate this, we implemented a set of ROS nodes, including publishers that broadcast joint states, coordinate frame transformations, and camera data, and subscribers that receive control commands or configuration updates. This section details the setup and configuration of the ROS publishers used to integrate the robot's sensors and actuators into the ROS framework.

3.2.1 TF Trees

`TF` is a ROS package that lets the user keep track of multiple coordinate frames over time. TF maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc, between any two coordinate frames at any desired point in time. [8]

This package is employed to determine the pose of the robot's end-effector relative to its base frame. To achieve this, the Universal Robot Description Format (URDF) file is provided to the TF (transform) package, which generates a TF tree that defines the kinematic relationships between the base link and the end-effector. The tree includes all intermediate joints, depicted in *Figure 3.8,* thereby capturing the physical structure and dependencies within the robot's kinematic chain. By establishing these spatial relationships, the system can perform trajectory planning and inverse kinematics to compute the joint angle trajectories necessary for moving the end-effector from an initial position to a desired target location within the 3D workspace.

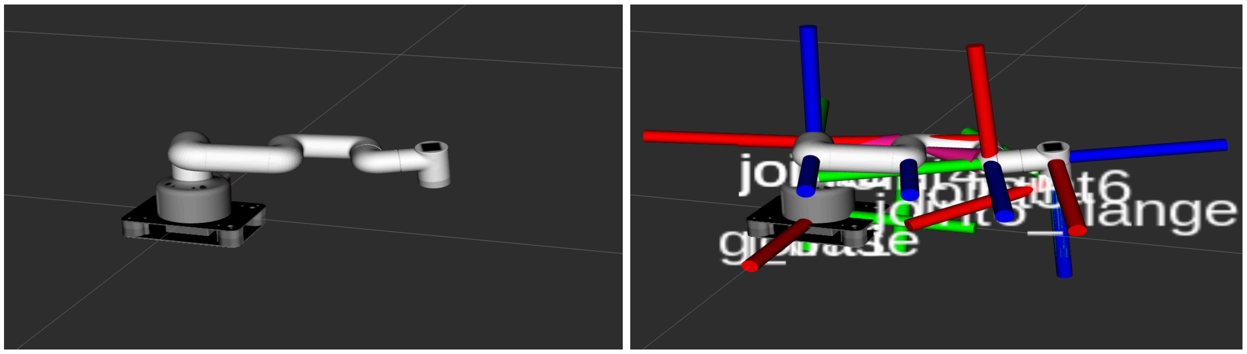


Figure 3.8: Full TF tree of MyCobot280pi in rviz – Before (left); After (right)

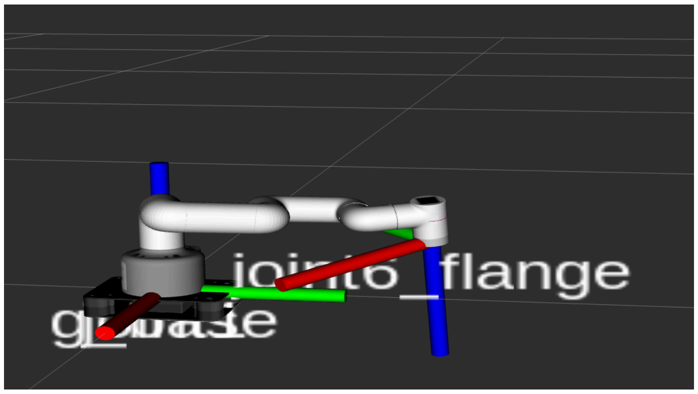


Figure 3.9: Simplified TF tree display highlighting base and end-effector frames

3.2.2 Publishing MyCobot Data to ROS

The robot's motors and encoders can be accessed via the intrinsic Application Programming Interface (API) for the MyCobot280pi provided by Elephant Robotics. To integrate this functionality within the ROS framework, a Python script (`mycobot_topics_pi.py`) and a corresponding ROS launch file (`communication_topic_pi.launch`) are used, as detailed in **Appendix C**. The launch file initializes the Python script as a ROS node, enabling standardized communication through ROS topics and services. This node is responsible for publishing continuous streams of messages containing the robot's current joint angles, end-effector pose, and gripper status. Additionally, it acts as a listener, subscribing to incoming requests that command changes to these same parameters. This bidirectional communication facilitates seamless control and monitoring of the robot within the ROS ecosystem. During the initial stages of this project, we tried to manipulate only the location of the end-effector using these protocols, but the robot's intrinsic inverse kinematics (IK) calculator resulted in very jerky and unpredictable movements from the robot. We later found that manipulating the joint angles directly was a much better strategy using this API. So moving forward, we ignore the communication channels to manipulate the xyz location of the end-effector via this API, and only use the API for manipulation of the joint angles and gripper status.

3.2.3 Publishing Joint State Data

The MoveIt framework—responsible for path planning and trajectory execution, as detailed in Section 3.3.1—relies on a continuous stream of `joint_state` messages to perform accurate motion planning. These messages need to be a list of 6 joint angles in radians, one for each respective joint in the 6-DOF robot. The MyCobot280pi API, however, provides a list of joint angles in degrees that is fully integrated with ROS. To address this, a python script (`joint_states.py`) was developed to convert joint angles from degrees to radians and republish them on the appropriate ROS topic compatible with the MoveIt framework, as detailed in **Appendix C**.

3.2.4 Camera

The MyCobot280pi robotic platform is equipped with a monocular USB camera, shown in *Figure 3.10*, which is connected directly to one of the USB ports on the robot's onboard Raspberry Pi. Image data from this camera is accessed and integrated into the ROS ecosystem using the **`usb_cam`** package. This package facilitates the acquisition and publication of image streams as ROS topics, thereby enabling their use in vision-based perception and control tasks within the broader robotic system. The package produces a stream of raw images, and a stream of depth images, which tell us the depth of each object in a given pixel of the raw image. While monocular cameras are generally limited in their ability to perceive depth accurately, they can provide sufficiently reliable results for short-range applications, typically within 1–2 feet. For the purposes of this project, such performance is adequate for effective visual perception and object localization within the robot's immediate workspace. Setup for the camera and the **`usb_cam`** package included camera calibration and providing the calibrated camera information to the **`usb_cam`** package. See the Computer Vision chapter for more details..

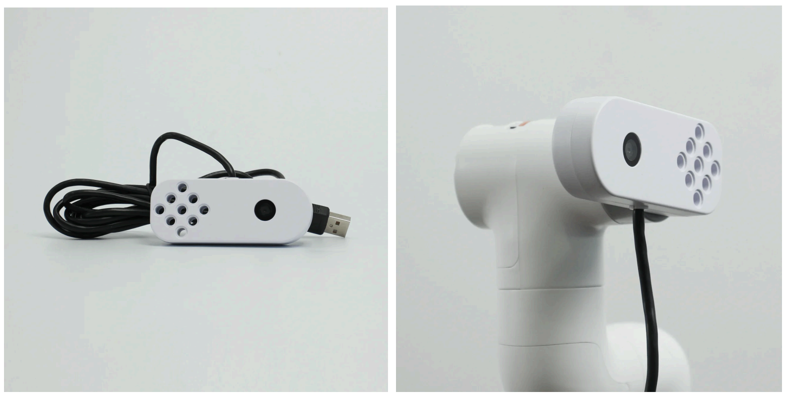


Figure 3.10: Monocular USB camera (left); Camera attached to end-effector (right)

3.2.5 Overview of Robot Communication Graph

Given the setup of the ROS Internal Communication in this section, let's visualize the ROS Nodes and ROS Topics and their relationships with each other via a Robot Operating System Qt-based GUI Framework (R) graph.

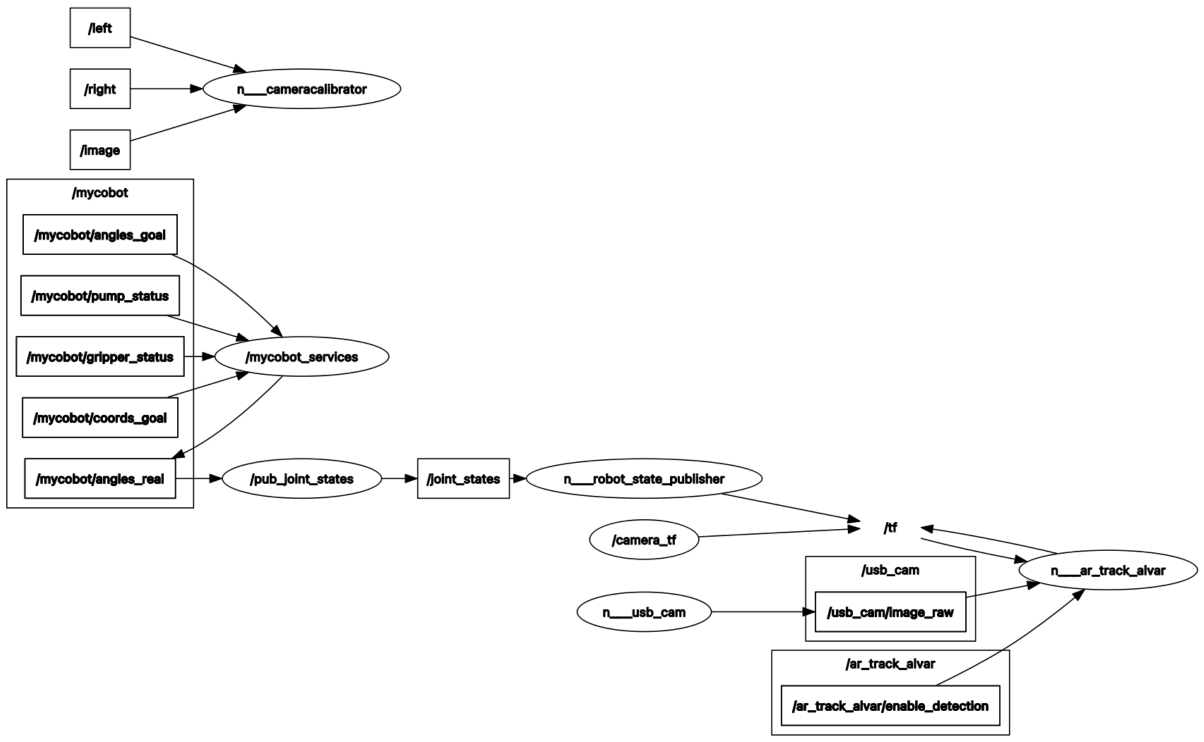


Figure 3.11: Rqt graph of ROS internal communications

3.3 Actuation

As the robot attempts to play board games, we need to establish a software pipeline to tell the robot to move its end-effector to a certain XYZ position to pick and place pieces on the game board. As we had previously discussed, the robot's API best supports actuation by manipulating joint angles. Thus, we need to implement a path planning framework to find Inverse Kinematics solutions and provide a collision-free trajectory from point A to point B in the 3D space. [9] During this project, one key challenge we needed to account for was the robot's dexterity and physical limitations when designing the board space for our games.

The design of the game board is constrained by the physical dimensions and kinematic limits of the MyCobot280Pi robotic arm, shown in *Figure 3.12*. Specifically, the size of the robot's joints imposes restrictions on both the overall dimensions of the board and the spacing between grid cells, as densely packed pieces make it difficult for the gripper to perform reliable pick-and-place operations within acceptable tolerances. According to the manufacturer's specifications, the robot has a maximum reach of 280 mm. However, this does not account for the inner area near the base that the robot cannot reach without risking self-collision, which our empirical testing identified to be approximately 100 mm in radius. As a result, the usable working range extends from 100 mm to

280 mm, meaning the largest game board that ensures full reachability should fit within a 180 mm-wide area. To maintain reliable manipulation at varying heights and ensure consistent graspability across the board, we opted for a more conservative 150 mm x 150 mm square layout. This ensures the robot can access all positions on the board, at any height, without exceeding its mechanical or kinematic constraints.

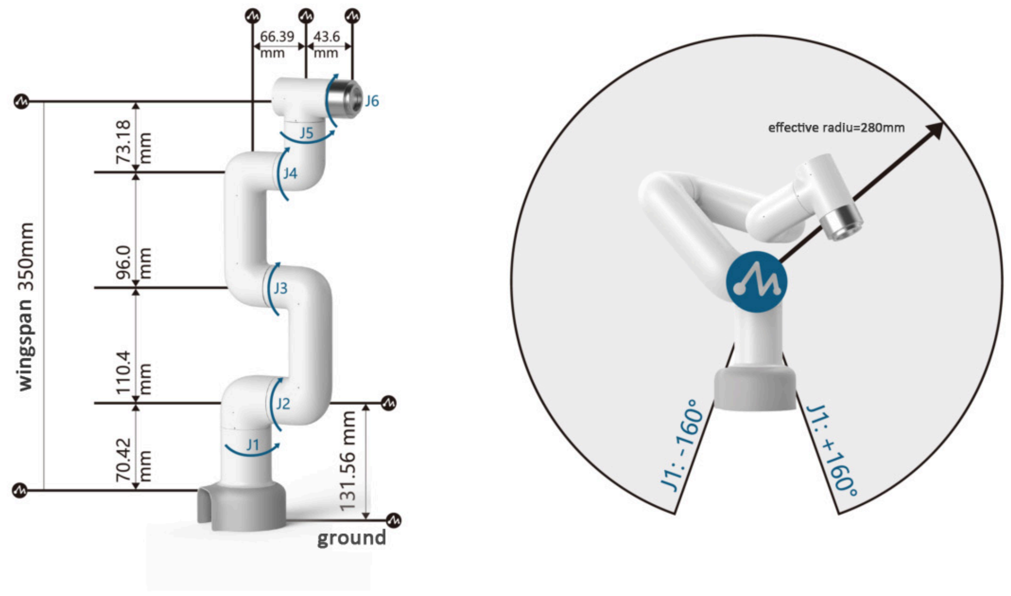


Figure 3.12: MyCobot280pi 6-DOF dimensions spec and reachable workspace [10]

Finally, when planning the physical motion from the initial to the target position in XYZ space, the movement must be decomposed into two discrete steps: pick and place. During the pick phase, an intermediate z-height is introduced to safely lift the piece above surrounding objects, providing clearance and avoiding collisions while transitioning. In the place phase, the piece is lowered from this intermediate height to its precise final position on the board. This separation ensures reliable and collision-free manipulation during both pickup and placement.

3.3.1 MoveIt Framework

MoveIt is a Motion Planning framework that is easy to set up and consists of multiple path planning modules for robot arms. The system is highly modular, offering a plug-and-play interface that accepts a robot's URDF (Unified Robot Description Format) model and produces a collision-free trajectory. This trajectory consists of a sequence of joint configurations which, when executed over time, result in a smooth and continuous motion of the end-effector from an initial position to a target location while maintaining the desired orientation. See MoveIt tutorials [11] for more information on setup and a list of motion planners.

The motion planner that we elected to use for this project was the *PILZ Industrial Motion planner*, which provided more Cartesian path planning algorithms that allows for more predictable movements for pick and place tasks for our board games. Within the PILZ Industrial Motion, we choose the *LIN* motion commander, which provides linear Cartesian trajectories from point A to point B in XYZ coordinates.

This planner generates smooth, coordinated motions for the robot's end-effector by following straight-line paths in space while respecting speed and acceleration limits. It creates a trapezoidal velocity profile, meaning the end-effector gradually speeds up, maintains a steady speed, and then slows down as it approaches the target. To move the end-effector in a straight line, the planner uses simple interpolation between the starting and ending positions. At the same time, it smoothly adjusts the orientation of the end-effector by rotating it between the start and goal angles using a method called spherical linear interpolation (slerp). Both the movement and the rotation are carefully synchronized in time so that the robot's motion looks natural and happens in a coordinated way. The output of the planner is a time-parameterized joint trajectory suitable for execution by the robot. [12]

One of the most accessible interfaces for interacting with MoveIt is the Python-based Move Group Interface. This interface provides high-level wrapper functions that support a wide range of common robotic operations, including setting joint-space or pose-space goals, generating motion plans, executing trajectories, incorporating collision objects into the planning scene, and managing object attachments to and from the robot. These capabilities make it particularly suitable for rapid development and prototyping in robotic applications.[13]

See **Appendix C** our code configuration of a Move Group Interface to interact with the MoveIt framework.

3.3.2 URDFs

The Unified Robot Description Format (URDF) is an XML-based specification used to describe the physical structure and kinematic properties of a robot. It encodes essential information such as link geometries, joint configurations, motor specifications, and inertial properties, effectively serving as a 3D model of the robot. Within the ROS framework, URDF files are parsed to construct the robot's model, which is then utilized for visualization, simulation, and motion planning. This representation allows both the system and the human operator to understand the robot's structure and capabilities prior to execution of any tasks.[14]

The URDF for the MyCobot280pi robot, as shown in **Appendix C**, was available on the Elephant Robotics GitHub page in the **`mycobot_description`** folder in the **`mycobot_ros`** git repository. We use the file path for the URDF as input to the MoveIt framework for path planning and collision avoidance.

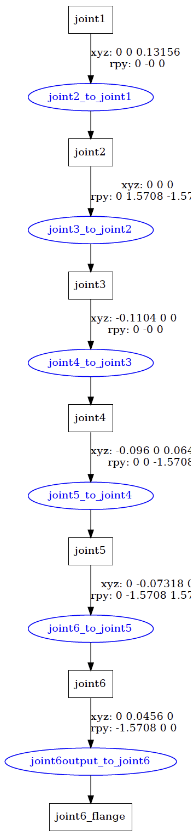


Figure 3.13: URDF visualization of joint link tree

3.3.3 ActionServer

Our MoveGroup, which represents our robot's internal information and source of interaction with the MoveIt framework, requests a plan from a configured path planner in MoveIt; in our case, it is the *LIN* motion commander from *PILZ Industrial motion planner*. When a plan is compiled and a trajectory for the plan is received, this trajectory must then be sent to our robot's API for processing and manipulating the joints. The trajectory in this case is a list of 6 joint angles that must be manipulated over time to change the robot's end-effector's position and orientation while avoiding self-collisions. To support asynchronous, feedback-driven execution of trajectories, we implement a ROS Action Server, which allows the system to receive goals, provide real-time feedback during execution, and return a result upon completion. This is particularly useful for operations such as trajectory execution, where continuous monitoring and the ability to preempt or cancel actions are essential for robust and flexible robot behavior.

3.4 Vision

One key challenge we encountered after setting up the robot's actuation and enabling the robot to play board games is minute physical inaccuracies when placing pieces. The inaccuracies were within 2-3 cm but this often affected the gripper's ability to pick up the piece in future iterations, and if the gripper were able to pick it up, it would do so in awkward positions and the errors would cascade in future movements, later prohibiting the piece to be picked up.

To solve this problem, we chose to integrate a computer vision system that looked for a specific piece on the game board using a unique Augmented Reality Tag (ARTag), identified its real location instead of blindly assuming its ideal location and picked up the piece in a more accurate and error-prone manner. When placing a piece, it still places it in the ideal location on the game board, which is fixed and calibrated before use.

While there are many techniques to identify game pieces on a board, like HSV filtering and Machine Learning image recognition models, we chose to use a much simpler and reliable method of leveraging QR codes called ARTags, from the **`ar_track_alvar`** ROS package. Techniques like HSV aren't very robust when working with multiple pieces that have the same color and require extra image processing, which consumes more of our limited resources with the Raspberry Pi. Image recognition ML models require taking 1000+ pictures of each unique game piece for training and testing, which is time-consuming and hard to scale for numerous games. We also could not find any off-the-shelf ML models for the recognition of game pieces.

3.4.1 Camera Calibration

In order to enable the monocular usb camera to provide accurate data about the xyz location of the game pieces, we first need to calibrate the camera by identifying the camera's intrinsic information like the focal lengths and camera calibration matrix, as well as supplying this information to the **`usb_cam`** package for integration into ROS.

To calibrate the camera, we utilized the **`camera_calibration`** ROS package to snap images of a checkerboard and output the required calibration information to be supplied to the **`usb_cam`** package. [15] Upon calibration, we created our YAML file and then placed it in the [***`~/.ros/camera_info`***] directory for use from the **`usb_cam`** package. See **Appendix C** for calibration and launch files.

3.4.1 ARTag ROS Package Setup

To develop a generalized game-playing robotic system, ARTags were selected as the preferred method for game piece identification and localization. By affixing an ARTag to each game piece, as shown in *Figure 3.14,* the system can accurately recognize and track their positions in real time, exhibiting high robustness across a wide range of lighting conditions. This approach offers both reliability and computational efficiency, making it a scalable solution applicable to a variety of games and piece types without requiring extensive reconfiguration.

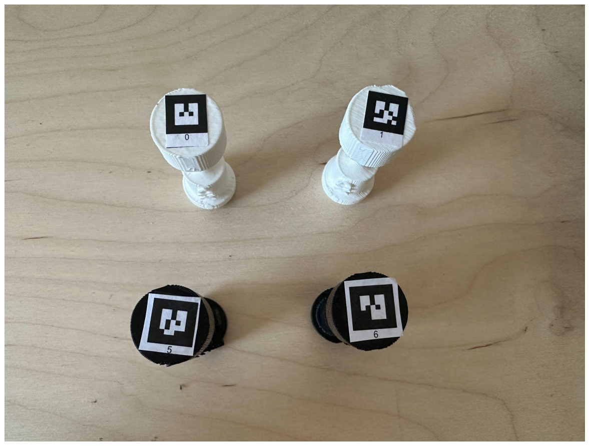


Figure 3.14: ARTag placement on 3D printed game pieces

To enable ARTag detection, the **`ar_track_alvar`** ROS package is utilized. This package requires configuring of its launch files, as shown in **Appendix C**, including parameters such as the physical size of the ARTags, the allowable detection tolerance between neighboring tags, and the detection window for individual tags. It processes input from both the raw RGB camera stream and the depth image feed, typically provided by the **`usb_cam`** package. Upon detecting ARTags, the package publishes the corresponding pose estimates as ROS messages on predefined topics, enabling their integration into the broader robotic perception and control pipeline.

3.4.2 TF Transform

As we launch the **`ar_track_alvar`** package, we begin to see coordinates being published with the XYZ location of the ARTags. This location, however, is with respect to the camera's frame, not the real world. To obtain the location of the piece in the real world, we need to do a transformation from the camera's frame to the robot's base frame. When doing this, we encounter a problem: where is the camera's frame?

To fix this problem, we create a new frame called ***`aligned_usb_cam`*** and specify it to be some offset from the end-effector, where our camera is attached to the robot. We run this new frame as a node (**`transformBoardcast.py`**) via a launch file (**`tf_bringup.launch`**) detailed in **Appendix C**. We then append this frame to our TF tree, thus allowing us to transform the location of the camera with respect to the robot's base, thus always obtaining the real-world coordinates of the camera's location.

3.5 Physical Calibration

Board games come with varying board sizes and piece shapes. This creates two problems, 1) localizing the board to fit the robot's reachability constraints, 2) enabling a generalized methodology to pick-and-place pieces, which is hard to do if every piece has different geometries.

3.5.1 Board Localization

To address the first issue, we designed a custom board enabling a precise fit for PDF-printable game boards, *depicted in Figure 3.15* using the process outlined in *Section 3.5.2*. Additionally, this custom board includes a cutout for the robot's base, allowing accurate localization of the game board relative to the robot.

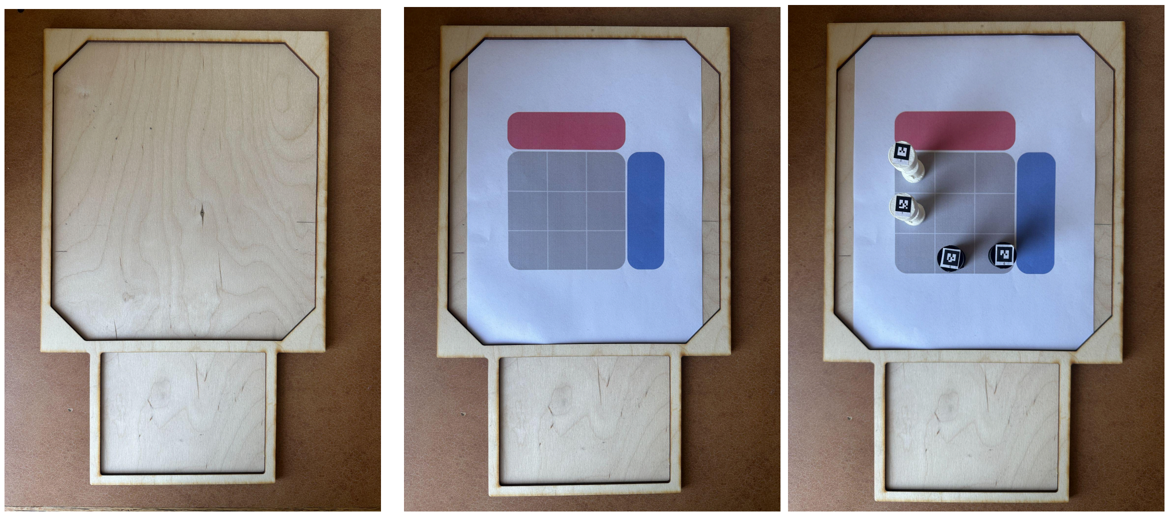
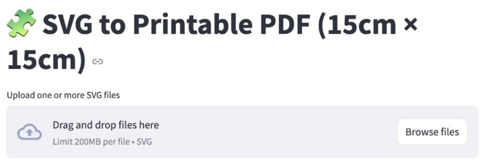


Figure 3.15: Custom wooden board – cutout for robot and paper game boards (left); printable SVG of Dodgem game board (middle); final setup with pieces on game board (right)

3.5.2 Printable Game Boards for Size Calibration

Due to robot reachability constraints described in Section 3.3, all games needed to be standardized to a 150 mm x 150 mm grid. To simplify this process, we developed a Streamlit-based web application to facilitate the conversion of SVG board images from GamesmanUni into a 150 mm x 150 mm format compatible with GamesmanROS. The figures below illustrate the steps required to generate the printable PDF version of the game board. See out GitHub documentation for information on launching the Steamlit application. The following figures depict the process of using the web application to print PDFs.

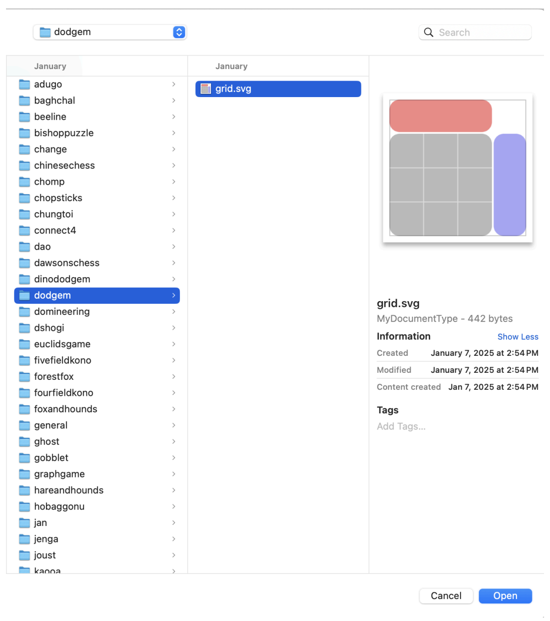


SVG to Printable PDF (15cm × 15cm)

Upload one or more SVG files

Drag and drop files here
Limit 200MB per file • SVG

Figure 3.16: Step 1: Open the Web Application from the GamesmanROS repository



adugo
baghchal
beeline
bishoppuzzle
change
chinesechess
chomp
chopsticks
chungtoi
connect4
dao
dawsonschess
dinododgem
dodgem
domineering
dshogi
euclidsgame
fivefieldkono
forestfox
fourfieldkono
foxandhounds
general
ghost
gobblet
graphgame
hareandhounds
hobaggonu
jan
jenga
joust

grid.svg

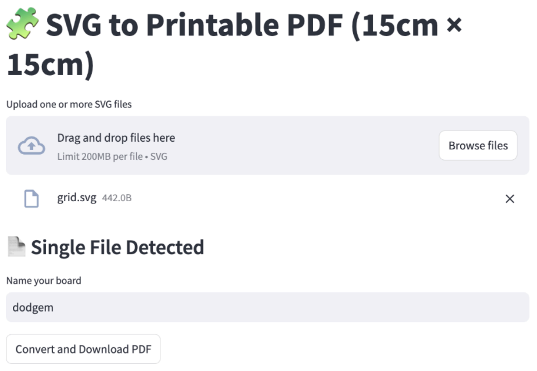
grid.svg
MyDocumentType - 442 bytes

Information

Created	January 7, 2025 at 2:54 PM
Modified	January 7, 2025 at 2:54 PM
Content created	Jan 7, 2025 at 2:54 PM

Tags
Add Tags...

Figure 3.17: Step 2: Choose the game board SVG from the GamesmanUni repository



SVG to Printable PDF (15cm × 15cm)

Upload one or more SVG files

Drag and drop files here
Limit 200MB per file • SVG

grid.svg 442.0B

Single File Detected

Name your board

dodgem

Figure 3.18: Step 3: Rename and download the pdf file

3.5.2 Piece Calibration

To address the second issue, we designed custom 3D-printed game pieces with flat tops to facilitate the attachment of ARTags, shown in *Figure 3.19*. This design allows the pieces to be reused across multiple games, ensuring consistent and reliable pick-and-place functionality for the robot's gripper.

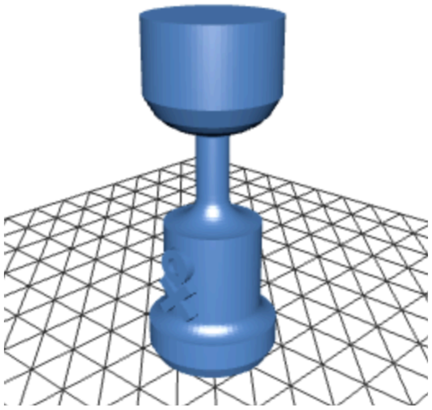


Figure 3.19: Custom 3D printed pieces

3.5.3 Integration

As seen in *Figure 3.20*, we present the solution to the two presented problems with a printed game board of Dodgem. This paper printout is physically localized with the robot, requiring no extra calibration or board detection. The pieces on the board are also reusable for other games and enable smooth pick and place capabilities for the gripper.

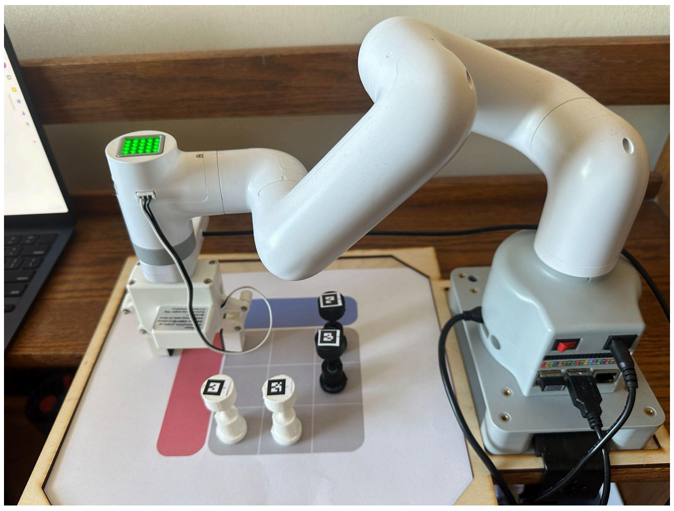


Figure 3.20: Final setup with robot for a game of Dodgem

Chapter 4

GamesmanROS: Software Interface

4.1 System Overview

GamesmanROS is our newly-designed software system for robotics that allows the robot to play multiple games from the GamesmanUni system. This system leverages data provided by GamesCraftersUWAPI, an interface for our frontend GamesmanUni, to support the generalized gameplay with the robot.

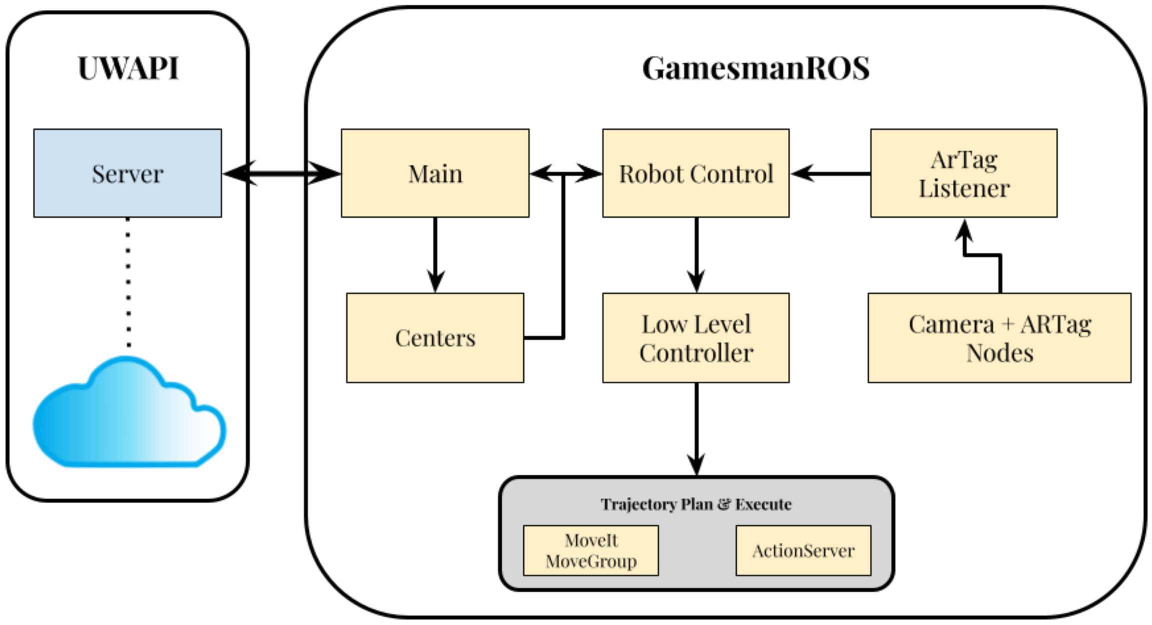


Figure 4.1: GamesmanROS: System architecture diagram

4.2 Robot MoveTypes

When enabling the robot to play various games in our system, it becomes very important to classify different move types at the atomic level to enable smooth game play and generalizability across various types of games.

The four basic atomic moves from the robot that we have created include:

- Place
- Rearrange
- Capture
- Removal

Place is the type of move where pieces are added to the board for gameplay. The robot scans for pieces outside the game board and picks them up to place on the game board. Examples of games that require the placement of pieces include TicTacToe and Dawson's Chess. [16]

Rearrange is the type of move where the number of pieces at the beginning of a game and the end of the game stays constant throughout the gameplay. All the robot has to do is rearrange pieces on the board for valid game moves. Examples of games that require rearrangement include Dao and All Queens Chess. [16]

Capture is the type of move where one piece occupies the spot of another piece, while removing the original piece from the board simultaneously. For the robot, this is a two step process as we have to remove the piece first off the board to a designated removal area to make room for the new piece to occupy the space. Upon this step, we move the new piece to occupy the space of the captured piece. Examples of games that require capture include Chess and Dragons & Swans. [16]

Removal is the type of move where pieces are simply removed from the board, and no other action is taken. For the robot, it simply picks the piece to be removed and places it in a designated removal area off the board. Examples of games that require removal include Chomp and Dodgem. [16]

4.3 Leveraging UWAPI

In order to represent board states and player turn information for the frontend, GamesCrafters developed the Universal Web Interface API (UWAPI) string format, which the API uses to interact with the game server and the frontend.

The general string format is as follows:

- Position String: "[player turn]_[1D board string]"
- Move String: "[move type]_[token]_[string index for new move]_[sound effect]"

Let's break it down with an example for the game of Dodgem:

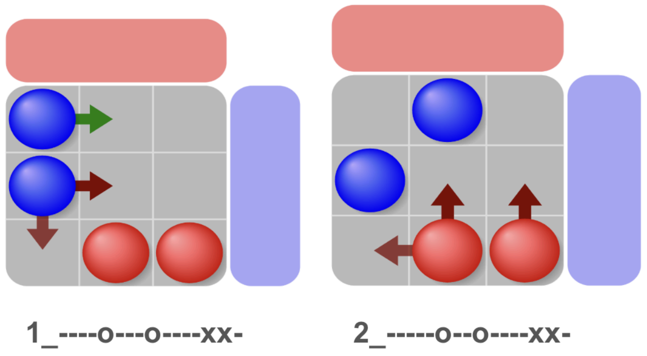


Figure 4.2: Position strings - First player (left); Second player (right)

In this example depicted in *Figure 4.2*, 1 represents the first player and 2 for the second player. The **'-'** character represents an empty space on the board, while 'o' and 'x' represent the blue and red pieces, respectively. The move string to transition from **"M_4_5_x"**, where M represents a "sliding" move of the piece in index 4 to index 5 to represent the new board position string **"2_-----o--o----xx-"** with **'x'** being the configured sliding sound effect. Using the position and move string format for UWAPI, we can now decode and extract robot atomic moves for every board game. Now we are ready to leverage metadata in GamesCraftersUWAPI to support the robot with important information for perfect game play. Such information includes center coordinates for pieces and its mapping to a UWAPI string and choosing the next best optimal move.

To differentiate moves into our 4 atomic move types: **Place**, **Rearrange**, **Capture** and **Removal**, we need to run a UWAPI string comparison between the current and future board states. To do this, we first extract the UWAPI representation of the board that results from making the optimal move by querying and parsing requests from the server. Upon this, we now are keeping track of the UWAPI strings for the current board state as well as the next state. We then analyze the strings to categorize the move types.

When handling **Place**, **Capture, and Removal** move types, we need to specify locations where pieces can be picked up and placed off the board. This is done by encoding pickup and placement zones in the game board to GamesmanROS by providing the corresponding SVG center coordinates for the zones.

Let's consider a simplified game with a 3x1 board and 2 pieces for each player.

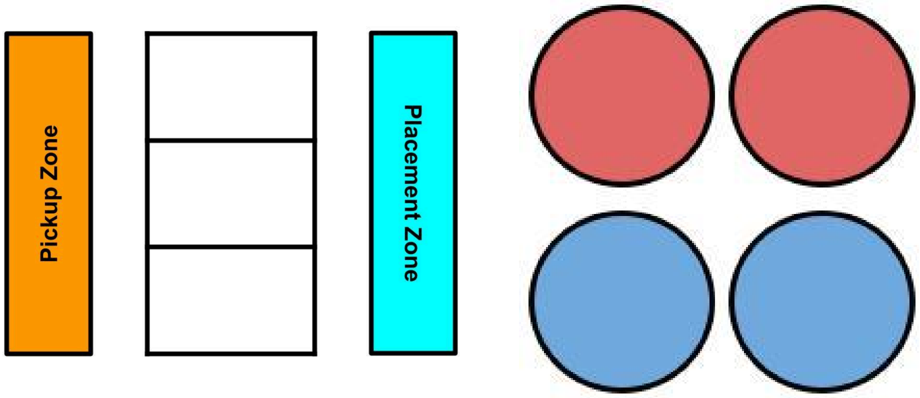


Figure 4.3: 3x1 Grid board game (left); Pieces (right)

For this game we will analyze the parsing of the UWAPI string to determine what type of robot atomic move has been made to provide an overview of UWAPI conversion to robot moves.

General guidelines for classifying UWAPI strings to robot move types:

Place: The UWAPI string representation reflects an increase of one in the total number of pieces between successive game states. This change indicates that a new piece has been placed on the board. With this addition, the robotic system can determine the exact location of the newly placed piece, allowing it to execute a precise pick-and-place action to physically update the game board and reflect the intended move in the ongoing gameplay.

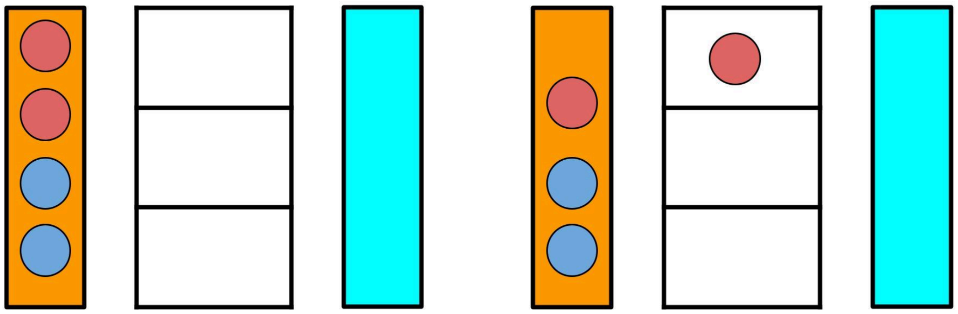


Figure 4.4: Empty board (left); Placing a piece on board (right)

Rearrange: The indices occupied in the initial position string that become vacant in the subsequent position string signify the source locations from which the robot must pick up pieces. Conversely, positions that transition from vacant to occupied indicate the target locations where the robot should place the pieces. By mapping these transitions, we can produce necessary pick-and-place actions for the robotic system. Note: In cases where multiple pieces need to be moved, such action is not supported by GamesCraftersUWAPI, thus it is not compatible with GamesmanROS.

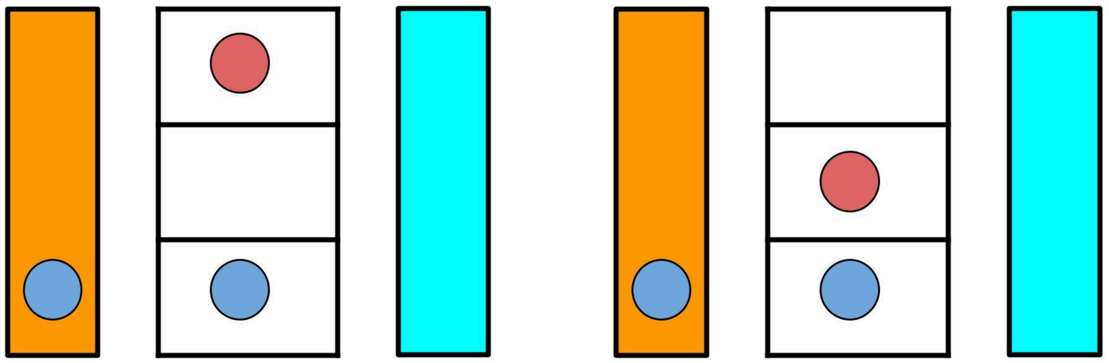


Figure 4.5: Rearranging pieces on a board – Before (left); After (right)

Capture: The UWAPI string representation reflects both a decrease in the total number of game pieces and a change in their positions between successive game states, essentially a Removal plus a Rearranger. We choose to create the category of *Capture* to better align with terminologies in gameplay of board games.

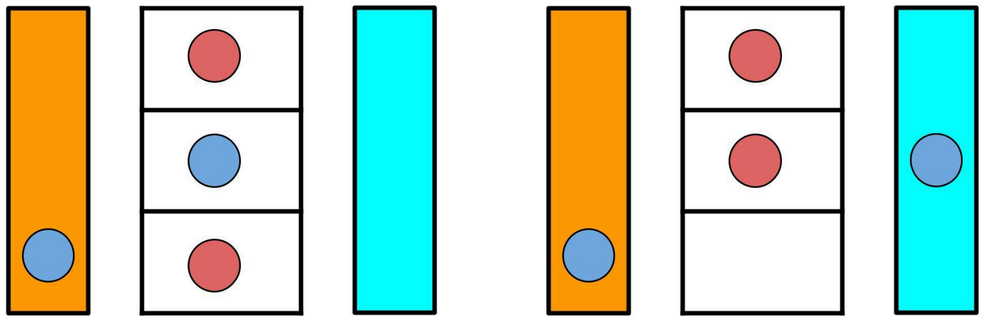


Figure 4.6: Capture of pieces on a board - Before (left); After (right)

Removal: The UWAPI string representation reflects a decrease of one in the total number of pieces between successive game states. This change indicates that a piece has been removed from the board. With this change, the robotic system can identify the specific location from which the piece was removed, enabling it to execute the appropriate pick-and-remove action to accurately update the physical game board and place the piece in a designated removal zone.

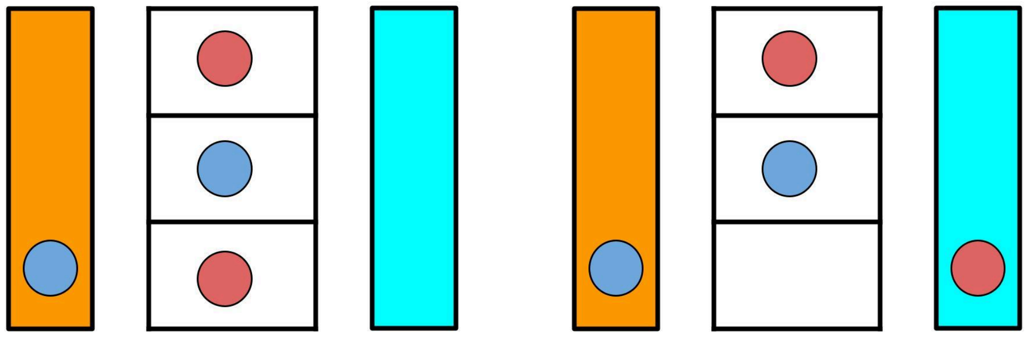


Figure 4.7: Removal of pieces on a board - Before (left); After (right)

Not every game consists of all four atomic move types, some games only have rearrangement, while others have a combination of two, three or all four move types. GamesmanROS requires the user to make a one-time configuration where they specify the *Type* class for a given game that corresponds to the combination of atomic move types. If such a combination is not already encoded, they are welcome to create a new *Type* class to handle the combination of atomic moves, see **Appendix C** for more details.

4.4 Integration

Now that we have classified games and their corresponding physical movements to the robot's atomic move types, we are now ready to find the pieces on the physical board and process the moves by actuating the robot arm.
To do this, we have created two layers of abstraction, the first layer is encoding the move recognition from the section above to output a list of *before* and *after* SVG coordinates that signify each move. For example, the rearranger example from above might return *before* coordinates of **[0.5, 0.5]** and *after* coordinate of **[0.5, 1.5],** resulting in **[[0.5, 0.5], [0.5, 1.5]]**. A rearranger move only consists of 1 set of before and after coordinates, while a capture includes 2 sets of before and after coordinates, one to remove the captured piece off the board and another to rearrange the existing piece to complete the capture.

The second layer of abstraction involves processing game moves by first transforming 2D SVG-based coordinates into corresponding 2D real-world coordinates. The robot is then commanded to execute a pick operation at the initial location and a place operation at the target location, with appropriate adjustments to the Z-axis in the real-world XYZ frame to account for vertical positioning, as seen in `robotControl.py` in **Appendix C**. This structured approach promotes modularity, enabling a plug-and-play design and facilitating the isolation and debugging of potential issues during future system development.

Lastly, we introduce the option to either play blindly, where the robot moves to the assumed piece location derived from the SVG coordinates, or it leverages the vision system that we had setup earlier to find the live error-prone location of the piece on the game board for greater pickup accuracy. For the vision based system, we query the ROS topic from the ARTag package for the location of the specified piece (`artag_listener.py`), which we track by storing the marker number in an internal data structure. With this live location, we can accurately command the robot to move and adjust the gripper for accurate pickup of pieces. Placement of pieces is still completed on the ideal location of the board using SVG coordinate transforms. See **Appendix C** for more information.

Chapter 5

Results

With the integration of hardware and software interfaces, GamesmanROS currently supports 22 games, as detailed in Table 1.1. Additional games can be accommodated through providing software support for custom game boards, specialized game pieces, and accurate localization to facilitate robot movement. GamesmanROS is modular to support configurations for new games. However, some games remain unsupported due to the physical constraints of the robot's reach. Considering a maximum board dimension of 150 mm x 150 mm and the use of custom 3D-printed game pieces, the largest practical board size compatible with reliable robot manipulation is a 5x5 grid; beyond this, spacing becomes insufficient to ensure consistent gripping accuracy.

The robot is capable of autonomously identifying pieces on the game board, planning and executing pick-and-place trajectories, and interacting with the UWAPI servers. It continuously updates the internal UWAPI string to reflect the current game state and requests the optimal next move from the server. Each move is executed within approximately 5 to 10 seconds, accounting for the asynchronous execution of pre-planned trajectories and the delay associated with gripper state transitions. At present, the robot can play autonomously against itself, computing optimal moves for both players. Additionally, we have developed and validated a software interface enabling human-versus-robot gameplay, which has been successfully demonstrated with Dawson's Chess. Future work will focus on robust testing of this capability to all 2D grid-based games currently supported by GamesmanROS, prior to public use.

5.2 Trajectory Execution Results

The path planning and execution modules within the MoveIt framework consistently generate precise linear trajectories for 100% of motion planning queries within the 150 mm × 150 mm game board. This includes accurate trajectories for both the pick/place z-values and elevated z-values used to safely hover over pieces during transitions between locations.

5.3 Vision Results

With the camera pose established relative to the robot's base frame, and the ARTag positions determined relative to the camera—both represented within the unified TF tree—we are able to compute the full transformation from each ARTag to the robot's base frame. This enables the extraction of accurate real-world (x,y,z) coordinates for each ARTag with respect to the robot. As a result, the system achieves precise localization of all game pieces within the robot's operational workspace. The error rate for the perceived location of a piece on the game board with respect to the real location with respect to the robot's base is within 2-3 cm, which is within tolerance for our gripper's pick and place capabilities.

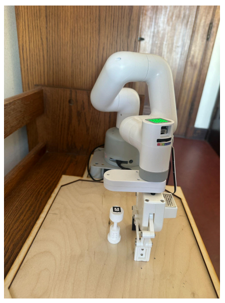
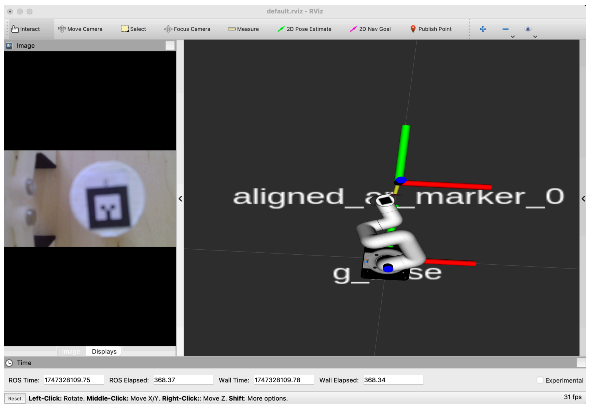


Figure 5.1: Game piece detection by robot



ROS Time: 1747328109.75 ROS Elapsed: 368.37 Wall Time: 1747328109.78 Wall Elapsed: 368.34

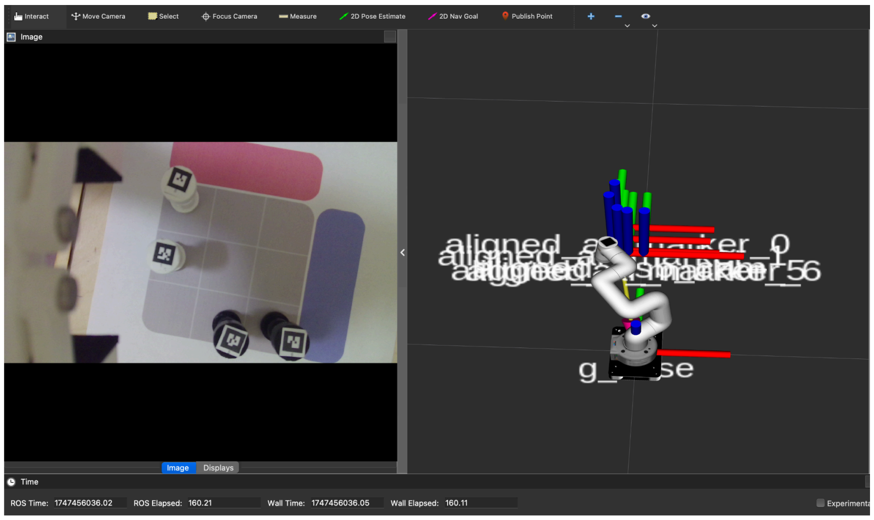
Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift:** More options.

Figure 5.2: ARTag recognition and TF frame visualization in rviz

5.4 Integration Results

Following the successful integration of the hardware and software interfaces described in Chapters 3 and 4, *GamesmanROS* enables the robot to reliably detect and track the last known positions of multiple game pieces on the board, as seen in *Figure 5.3*. The system also tracks the association between ARTag identifiers and their corresponding indices in the UWAPI position string, dynamically updating this mapping as moves are made. This architecture reduces the search space for piece localization and establishes a foundation for enabling human-versus-robot gameplay.

In a series of 100 pick-and-place executions conducted across six perfect-play, robot-versus-robot games of *Dodgem*, the robot achieved a 92% accuracy rate in successfully picking up pieces, avoiding collisions by hovering over other pieces, and precisely placing them within their designated grid cells—mirroring the visualized gameplay in the *GamesmanUni* web application. The remaining 8% error rate was primarily due to minor collisions during gripper actuation, which occasionally displaced neighboring pieces. This marks a significant improvement over the previously observed 27% error rate in a blind (non-vision) baseline, where the robot operated without perception feedback. Integrating a vision system effectively mitigated cascading errors by allowing the robot to confirm the actual position of each piece before executing a pick, thereby compensating for any minor deviations introduced during prior moves. This substantial reduction in error demonstrates the effectiveness of combining perception with motion planning and sets the stage for further enhancements to gameplay accuracy and robustness in real-world conditions.



ROS Time: 1747456036.02 ROS Elapsed: 160.21 Wall Time: 1747456036.05 Wall Elapsed: 160.11

Figure 5.3: ARTag recognition of all pieces in Dodgem

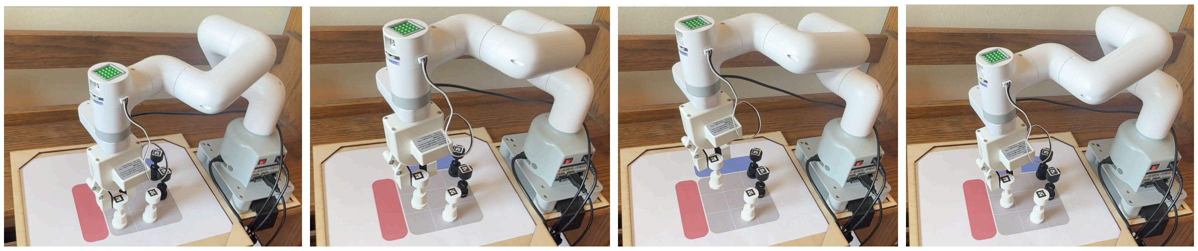


Figure 5.4: Robot playing first move in Dodgem

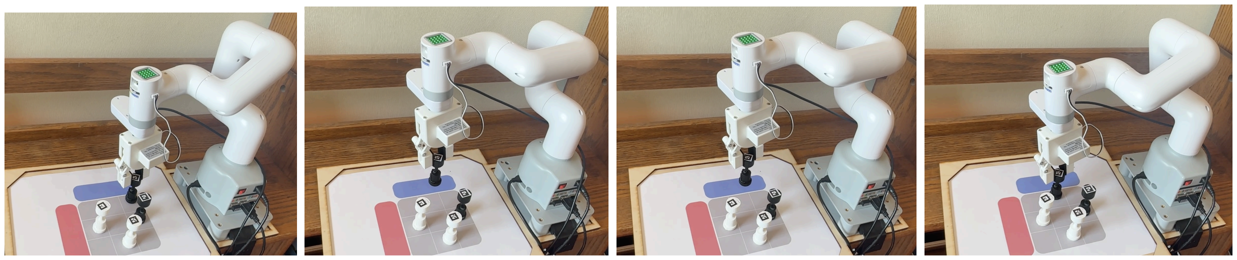


Figure 5.5: Robot playing second move in Dodgem

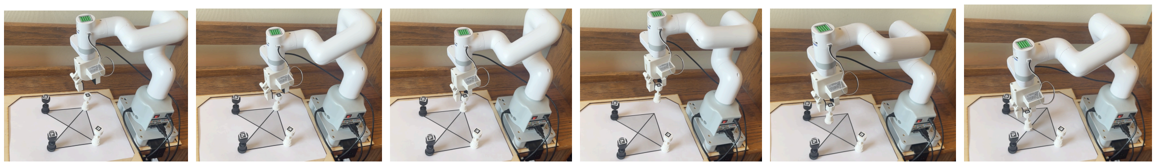


Figure 5.6: Robot playing first move in Pong Hau K'i

See Table 1.1 below for a list of games in GamesmanUni and its status in GamesmanROS.

Game Name	**Status**
Place	
Dawson's Chess	**Supported**
Domineering	
Y	
Capture	
Removal	
Chomp	
Rearranger	
3-Spot	
All Queens Chess	**Supported**
Beeline	
Change!	**Supported**
Dao	**Supported**
Five-Field Kono	**Supported**
Fox and Hounds	
Hares and Hounds	**Supported**
Jan (4x4)	**Supported**
Joust	**Supported**
Lewthwaite’s Game	**Supported**
Mu Torere	
Nu Tic-Tac-Toe	**Supported**
Pong Hau Ki	**Supported**
Adugo	**Supported**
HoBagGonu	
Place + Rearranger	
Abrobad	
Achi	**Supported**

Teeko	Supported
Tsoro Yematatu	
Rearranger + Removal	
Dino Dodgem	Supported
Dodgem	Supported
Rearranger + Capture	
1D Chess	Supported
Four Field Kono	Supported
Quick Chess	Supported
Chess	Supported
Place + Rearranger + Removal	
Nine Men's Morris	Supported
Place + Rearranger + Removal	
Bagh-Chal	Supported
Dragons & Swans	Supported
Rearranger + Capture + Removal	
Kōnan	
Place + Rearranger + Capture + Removal	
Yote	
Not categorized / Custom requirements	
10-to-0-by-1-or-2	
Chinese Chess	
Chopsticks	
Chung-toi	
Connect 4	
Dōbutsu shōgi	
Forest Fox	
Ghost	

Graph	
Jenga	
Kayles	
L-Game	
Lite3	
Nim	
NoTakTo	
Odd or Even	
Othello	
Quarto	
Quick Cross	
Quixo	
Rubic’s Magic	
Sim	
Slide 5	
Snake	
Square Dance	
Tac-Tix	
Tic-Tac-Toe	
Tic-Tac-Two	
Totto and Otto	
TopiTop	
Euclid’s Game	
Shift Tac Toe	
Abalone	
Mancala	

Table 1.1: List of all games in GamesmanUni and its compatibility status with GamesmanROS

Chapter 6

Future Work

6.1 Future Work

When we started this project, the goal was to enable human interaction with Gamesman. With GamesmanROS, we are now able to bring board games to life by having the robot play itself for all compatible games. With a better camera system, we should be able to enable human interaction for most of our supported games, where the computer detects a human move and responds with the corresponding robot move. We can then leverage this software solution to visualize human errors in the GamesmanUni web interface. Future iterations of this project can expand to user-friendly GUI experience as well as visualization features for testing and debugging.

6.1.1 Play against a human

While the robot can support human-readable moves for Dawson's chess, a 1D game board, we are hoping to expand this software to support 2D game boards. A key challenge is snapping the fuzzy real-world XYZ coordinates of a given piece and then converting its locations to its respective UWAPI string. This pipeline has been tried in this project, and we found many gaps that were introduced. Future iterations of this project should re-examine the pipeline and find robust solutions for every step from piece recognition to UWAPI string conversion. In addition, we are requiring a human to provide keyboard input when they have completed their move, so the robot can begin its analysis. This was mainly to conserve resources and reduce errors in mistaken scans when a user is transitioning pieces across various locations in the game board. A nice optimization would be to solve this issue such that no human input is required to begin with, to enable playing against the robot.

6.1.2 Adding more games to GamesmanROS

GamesmanROS is proud to support 22 games! However, this excludes many grid puzzles and other games with unconventional boards like hexagonal and diagonal boards. Most of the games on our system also have pieces with varying shapes and often overlap multiple spots on the grid board, like domineering. These custom requirements require adding a customized solution to the GamesmanROS. Luckily, GamesmanROS is a modular software platform where custom functionality can be added using object-oriented programming and plugged into the current software stack.

Customized solutions may include utilizing suction grippers, dexterous hand-like grippers, or 3D printing custom grippers to pick and place unique types of pieces. Games that may require customized grippers include Connect 4 and Jenga.

6.1.3 GUI for GamesmanROS

The greatest tool that we have built to advance the mission of GamesCrafters is GamesmanUni, a web interface that allows for interaction and visualization of our solved games. When embarking on the journey of building GamesmanROS, the core motivation was not only to play our system via a robot, but also to check the answers against our strongly solved solutions in real-time. This can be possible with future integrations of GamesmanROS with GamesmanUni. Further motivations include the challenge that operating the robot via GamesmanROS requires someone with a technical background, limiting the availability and use of the robot to the general public. We hope to build a personalized GUI for GamesmanROS that allows for user-friendly startup and debugging of the robot.

Chapter 7

Conclusion

With a strong foundational robotic system in place, *GamesmanROS* is well-positioned to support a broader range of games—and even puzzles—on physical hardware. By investing in enhancements such as a more positionally accurate robot with a larger reachable workspace, advanced end-effectors (e.g., suction or human-like grippers), and a static stereo camera setup for improved depth perception, the system could achieve greater precision in piece manipulation and scale to accommodate larger game boards.

GamesmanROS lays the groundwork for a flexible and intelligent game-playing platform—and while there’s plenty of room to grow, we've taken a significant step forward in bridging the gap between AI, robotics, and real-world gameplay.

Bibliography

[1] Yanko Design, “Swappable, AI-powered board games exist and we’re finally in the future!,” *Yanko Design - Modern Industrial Design News*, Nov. 19, 2019. https://www.yankodesign.com/2019/11/19/swappable-ai-powered-board-games-exist-and-were-finally-in-the-future/ (accessed May 19, 2025).

[2] E. Karmanova, V. Serpiva, S. Perminov, R. Ibrahimov, A. Fedoseev, and D. Tsetserukou, “SwarmPlay: A Swarm of Nano-Quadcopters Playing Tic-tac-toe Board Game against a Human,” *Special Interest Group on Computer Graphics and Interactive Techniques Conference Emerging Technologies*, pp. 1–4, Aug. 2021, doi: https://doi.org/10.1145/3450550.3465346.

[3] K. Shin et al., "Exploring the Capabilities of a General-Purpose Robotic Arm in Chess Gameplay," 2023 IEEE-RAS 22nd International Conference on Humanoid Robots (Humanoids), Austin, TX, USA, 2023, pp. 1-8, doi: 10.1109/Humanoids57100.2023.10375209. keywords: {Service robots;Pipelines;Robot vision systems;Humanoid robots;Games;Grasping;Manipulators},

[4] C. Matuszek et al., "Gambit: An autonomous chess-playing robotic system," 2011 IEEE International Conference on Robotics and Automation, Shanghai, China, 2011, pp. 4291-4297, doi: 10.1109/ICRA.2011.5980528. keywords: {Games;Cameras;Robot sensing systems;Image color analysis;Detectors;Humans;Mechanism Design of Manipulators;Physical Human Robot Interaction},

[5] Bailey, Donald & Mercer, Ken & Plaw, Colin. (2004). Autonomous game playing robot.

[6] “Desktop Raspberry Pi Robot Arm | Elephant Robotics,” *www.elephantrobotics.com*, May 17, 2021. https://www.elephantrobotics.com/en/mycobot-pi/

[7] “ROS/NetworkSetup - ROS Wiki,” *wiki.ros.org*. https://wiki.ros.org/ROS/NetworkSetup

[8] “tf - ROS Wiki,” *Ros.org*, 2024. https://wiki.ros.org/tf (accessed May 13, 2025).

[9] MathWorks, “What Is Inverse Kinematics?,” *www.mathworks.com*. https://www.mathworks.com/discovery/inverse-kinematics.html

[10] “1 Introduction Of Product Parameters · GitBook,” *Elephantrobotics.com*, 2023. https://docs.elephantrobotics.com/docs/gitbook-en/2-serialproduct/2.1-280/2.1.5.1-IntroductionOfProductParameters.html

[11] “MoveIt Tutorials — moveit_tutorials Kinetic documentation,” *docs.ros.org*. https://docs.ros.org/en/kinetic/api/moveit_tutorials/html/index.html

[12] “Pilz Industrial Motion Planner — moveit_tutorials Melodic documentation,” *Ros.org*, 2025. https://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/pilz_industrial_motion_planner/pilz_industrial_motion_planner.html#the-lin-motion-command (accessed May 13, 2025).

[13] “Move Group Python Interface — moveit_tutorials Kinetic documentation,” *docs.ros.org*. https://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/move_group_python_interface/move_group_python_interface_tutorial.html

[14] “Glossary | URDF,” *Formant*, Jun. 27, 2024. https://formant.io/resources/glossary/urdf/ (accessed May 13, 2025).

[15] “camera_calibration/Tutorials/MonocularCalibration - ROS Wiki,” *wiki.ros.org*. https://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration

[16] “GamesmanUni,” *Berkeley.edu*, 2025. https://nyc.cs.berkeley.edu/uni/games (accessed May 13, 2025).

Appendix A: User's Guide

Step 1: Follow the instructions in **3.1.1** to establish a Remote SSH connection with the robot. Password for the SSH connection to the robot is **Elephant**.

```
ssh er@10.40.193.189
Elephant
```

Step 2: Open 6 separate terminals with Remote SSH established in each terminal, run the following command in each terminal:

```
cd Documents/GamesmanROS/catkin_ws && source ~/.bashrc && source
/opt/ros/noetic/setup.bash && source devel/setup.bash
```

Step 3: Running the following commands in each respective terminal, one command per terminal. Allow 3-5 sec delay between running commands in each terminal.

```
roscore
roslaunch mycobot_communication communication_topic_pi.launch
roslaunch gamesmanros tf_bringup.launch
roslaunch gamesmanros spawn_all_pieces.launch
roslaunch mycobot_280_moveit mycobot_moveit.launch
rosrun gamesmanros main.py
```

Step 4: In the terminal running the [*rosrun gamesmanros main.py]* command, follow the TUI (Text-Based User Interface) to choose and play the game of your choice. See **Appendix C** for supported games.

```
[Truncated for visual]
82  :  Tic-Tac-Toe
83  :  Tic-Tac-Two
84  :  Tiltago
85  :  Toads and Frogs Puzzle
86  :  Toot and Otto
87  :  Top Spin
88  :  Topitop
89  :  Towers of Hanoi
```

```
90  :  Tsoro Yematatu
91  :  Winkers
92  :  Y
93  :  Yoté
Pick the index of the game you want to play: 24
Game Chosen:  dodgem
0  :  regular
Pick the index of the variant you want to play: 0
Human or Robot (Enter 'h' or 'r')
Player 1: r
Player 2: r
A :  M_4_5_x
B :  M_14_10_x
[Truncated for visual]
```

Appendix B: Adding a Game to GamesmanROS

Prerequisite: Games must be added to GamesmanClassic (or other game server), GamesCraftersUWAPI and available through GamesmanUni.

Step 1: In **centers.py** file [GamesmanROS/catkin_ws/src/gamesmanros/src] add your game to the following functions and variables, using examples provided:

Make sure that the string key for gameId matches that of GamesCraftersUWAPI

```
def get_centers(game):
   data = {
       "1dchess": [[0.5 + i, 0.5] for i in range(8)],
       "allqueenschess": [[(i % 5 + 0.5) / 5, (i // 5 + 0.5) / 5] for i in range(25)],
       "dawsonschess": [[0.5,1.5],[1.5,1.5],[2.5,1.5],[3.5,1.5],[4.5,2.5],[5.5,2.5]],
       "dao": [[(i % 4 + 0.5) / 4, (i // 4 + 0.5) / 4] for i in range(16)],
       "dodgem": [[(i % 4 + 0.5), (i // 4 + 0.5)] for i in range(16)],
       "dinododgem": [[i // 5 + 0.5, 5.5 - (i % 5)] for i in range(25)] + [
                    [1.5, 0.5], [2.5, 0.5], [3.5, 0.5], [4.5, 0.5],
                    [5.5, 1.5], [5.5, 2.5], [5.5, 3.5], [5.5, 4.5]
                ],
       "dragonsandswans": [[(i % 4 * 10 + 5) / 35, (i // 4 * 10 + 5) / 46] for i in
range(16)] + [[28.7/35, 43/46], [30.2/35, 43/46], [28.7/35, 46/46], [30.2/35, 46/46]],
       "jan": [[((i % 4)), ((i // 4))] for i in range(16)],
       "joust":
[[0.5,0.5],[1.5,0.5],[2.5,0.5],[3.5,0.5],[0.5,1.5],[1.5,1.5],[2.5,1.5],[3.5,1.5],[0.5,2.5],[
1.5,2.5],[2.5,2.5],[3.5,2.5],[0.5,3.5],[1.5,3.5],[2.5,3.5],[3.5,3.5]]
   }
   return data[game] if game in data else None

ar_tracker = {
   "dodgem" : {"ar_marker_16" : 4, "ar_marker_13" : 8, "ar_marker_6" : 13, "ar_marker_7" :
14}
   }

def get_dim(game):
   data = {
       "1dchess": 8,
       "allqueenschess": 5,
       "dawsonschess": 5,
       "dao": 4,
       "dodgem": 3,
       "jan": 3,
       "joust": 4
   }
   return data[game] if game in data else None

def get_pickup(game):
   data = {}
   return data[game] if game in data else None

def get_capture(game):
   data = {
       "1dchess": [5.5, 2.5]
   }

   return data[game] if game in data else None
```

Step 2: Modify **robotControl.py** [GamesmanROS/catkin_ws/src/gamesmanros/src] to specify the Type of game you are adding to GamesmanROS, see **Appendix C** for more details on each corresponding Type of game. If a certain combination of robot atomic moves is not supported by GamesmanROS, add your own custom Type below using previous ones as reference.

Make sure that the string key for gameId matches that of GamesCraftersUWAPI

```
def getType(gameId):
   types_of_games = {"Type1": ["dawsonschess"],
                     "Type2": [],
                     "Type3": [],
                     "Type4": ["allqueenschess", "dao", "jan", "joust", "lewthwaitesgame",
                               "nutictactoe", "ponghauki", "adugo"],
                     "Type5": ["achi", "teeko"],
                     "Type6": ["dinododgem", "dodgem"],
                     "Type7": ["1dchess", "fourfieldkono", "quickchess"],
                     "Type8": [],
                     "Type9": ["baghchal", "dragonsandswans"],
                     "Type10": [],
                     "Type11": []}

   types = {"Type1" : Type1, "Type2" : Type2, "Type3" : Type3, "Type4" : Type4,
            "Type5" : Type5, "Type6" : Type6, "Type7" : Type7, "Type8" : Type8,
            "Type9" : Type9, "Type10" : Type10, "Type11" : Type11}

   for gameType in types_of_games:
       if gameId in types_of_games[gameType]:
           return types[gameType]

   print("Error in RobotControl GameType!")
   return None
```

Appendix C: Code

Visit our official git repository, **GamesmanROS**, part of the GamesCrafters organization!
https://github.com/GamesCrafters/GamesmanROS.git

Below are snippets of code that are referenced in the paper:

```
 <!-- Argument for the URDF file path -->
 <arg name="robot_description"
default="/home/er/catkin_ws/src/mycobot_ros/mycobot_description/urdf/mycobot/mycobot_urdf.ur
df" />

 <!-- Load the URDF file into the parameter server -->
 <param name="robot_description" textfile="$(arg robot_description)" />

 <!-- <param name="use_sim_time" value="false" /> -->
 <node name="pub_joint_states" pkg="gamesmanros" type="joint_states.py" output="screen" />

 <!-- Robot State Publisher to publish the TF tree -->
 <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
respawn="false">
   <param name="robot_description" value="$(arg robot_description)" />
 </node>
```

*Snippet of **`tf_bringup.launch`** file to launch the TF tree of the robot*

```
<launch>
   <!-- Select connecting device and serial port ，选择连接设备及串口-->
   <arg name="port" default="/dev/ttyAMA0" />
   <arg name="baud" default="1000000" />

   <!-- Open communication service --><!-- 开启通讯服务 -->
   <node name="mycobot_services" pkg="mycobot_communication" type="mycobot_topics_pi.py"
output="screen">
       <param name="port" type="string" value="$(arg port)" />
       <param name="baud" type="int" value="$(arg baud)" />
       </node>
</launch>
```

*Launch file **`communication_topic_pi.launch`***

```
from mycobot_communication.msg import (
   MycobotAngles,
   MycobotCoords,
   MycobotSetAngles,
   MycobotSetCoords,
   MycobotGripperStatus,
   MycobotPumpStatus,
)

from pymycobot import MyCobot
# from pymycobot import MyCobotSocket

class Watcher:
   """this class solves two problems with multithreaded
   programs in Python, (1) a signal might be delivered
   to any thread (which is just a malfeature) and (2) if
   the thread that gets the signal is waiting, the signal
   is ignored (which is a bug).

   The watcher is a concurrent process (not thread) that
   waits for a signal and the process that contains the
   threads.  See Appendix A of The Little Book of Semaphores.
   http://greenteapress.com/semaphores/

   I have only tested this on Linux.  I would expect it to
   work on the Macintosh and not work on Windows.
   """

   def __init__(self):
       """Creates a child thread, which returns.  The parent
       thread waits for a KeyboardInterrupt and then kills
       the child thread.创建一个返回的子线程。 父线程等待 KeyboardInterrupt
       然后杀死子线程。
       """
       self.child = os.fork()
       if self.child == 0:
           return
       else:
           self.watch()

   def watch(self):
       try:
           os.wait()
       except KeyboardInterrupt:
           # I put the capital B in KeyBoardInterrupt so I can
           # tell when the Watcher gets the SIGINT
           print("KeyBoardInterrupt")
           self.kill()
       sys.exit()

   def kill(self):
       try:
            os.kill(self.child, signal.SIGKILL)
        except OSError:
            pass


class MycobotTopics(object):
   def __init__(self):
       super(MycobotTopics, self).__init__()

       rospy.init_node("mycobot_topics_pi")
       rospy.loginfo("start ...")
       # problem
       port = rospy.get_param("~port", os.popen("ls /dev/ttyAMA*").readline()[:-1])
       baud = rospy.get_param("~baud", 1000000)
       rospy.loginfo("%s,%s" % (port, baud))
       # self.mc = MyCobotSocket(port, baud) # port
       # self.mc.connect()   #pi
       self.mc = MyCobot(port, baud)
       self.lock = threading.Lock()

   def start(self):
       pa = threading.Thread(target=self.pub_real_angles)
       pb = threading.Thread(target=self.pub_real_coords)
       sa = threading.Thread(target=self.sub_set_angles)
       sb = threading.Thread(target=self.sub_set_coords)
       sg = threading.Thread(target=self.sub_gripper_status)
       sp = threading.Thread(target=self.sub_pump_status)

       pa.setDaemon(True)
       pa.start()
       pb.setDaemon(True)
       pb.start()
       sa.setDaemon(True)
       sa.start()
       sb.setDaemon(True)
       sb.start()
       sg.setDaemon(True)
       sg.start()
       sp.setDaemon(True)
       sp.start()

       pa.join()
       pb.join()
       sa.join()
       sb.join()
       sg.join()
       sp.join()

   def pub_real_angles(self):
       """Publish real angle"""
       """发布真实角度"""
       pub = rospy.Publisher("mycobot/angles_real", MycobotAngles, queue_size=5)
```

```
        ma = MycobotAngles()
        r = rospy.Rate(30)
        while not rospy.is_shutdown():
            self.lock.acquire()
            angles = self.mc.get_angles()
            self.lock.release()
            if angles:
                ma.joint_1 = angles[0]
                ma.joint_2 = angles[1]
                ma.joint_3 = angles[2]
                ma.joint_4 = angles[3]
                ma.joint_5 = angles[4]
                ma.joint_6 = angles[5]
                pub.publish(ma)
            r.sleep()

    def pub_real_coords(self):
        """publish real coordinates"""
        """发布真实坐标"""
        pub = rospy.Publisher("mycobot/coords_real", MycobotCoords, queue_size=5)
        ma = MycobotCoords()
        r = rospy.Rate(30)
        while not rospy.is_shutdown():
            self.lock.acquire()
            coords = self.mc.get_coords()
            self.lock.release()
            if coords:
                ma.x = coords[0]
                ma.y = coords[1]
                ma.z = coords[2]
                ma.rx = coords[3]
                ma.ry = coords[4]
                ma.rz = coords[5]
                pub.publish(ma)
            r.sleep()

    def sub_set_angles(self):
        """subscription angles"""
        """订阅角度"""
        def callback(data):
            angles = [
                data.joint_1,
                data.joint_2,
                data.joint_3,
                data.joint_4,
                data.joint_5,
                data.joint_6,
            ]
            sp = int(data.speed)
            self.mc.send_angles(angles, sp)

        sub = rospy.Subscriber(
            "mycobot/angles_goal", MycobotSetAngles, callback=callback
        )
        rospy.spin()

    def sub_set_coords(self):
        def callback(data):
            angles = [data.x, data.y, data.z, data.rx, data.ry, data.rz]
            sp = int(data.speed)
            model = int(data.model)
            self.mc.send_coords(angles, sp, model)

        sub = rospy.Subscriber(
            "mycobot/coords_goal", MycobotSetCoords, callback=callback
        )
        rospy.spin()

    def sub_gripper_status(self):
        """Subscribe to Gripper Status"""
        """订阅夹爪状态"""
        def callback(data):
            if data.Status:
                self.mc.set_gripper_state(0, 80)
            else:
                self.mc.set_gripper_state(1, 80)

        sub = rospy.Subscriber(
            "mycobot/gripper_status", MycobotGripperStatus, callback=callback
        )
        rospy.spin()

    def sub_pump_status(self):
        def callback(data):
            if data.Status:
                self.mc.set_basic_output(data.Pin1, 0)
                self.mc.set_basic_output(data.Pin2, 0)
            else:
                self.mc.set_basic_output(data.Pin1, 1)
                self.mc.set_basic_output(data.Pin2, 1)

        sub = rospy.Subscriber(
            "mycobot/pump_status", MycobotPumpStatus, callback=callback
        )
        rospy.spin()
```

Python Script ***mycobot_topics_pi.py***

```
class JointStateRepublisher:
   def __init__(self):
       # Create a publisher for JointState
       self.joint_state_pub = rospy.Publisher('/joint_states', JointState, queue_size=10)

       # Subscribe to the topic that publishes joint angles (of type JointAngles)
       rospy.Subscriber('/mycobot/angles_real', MycobotAngles, self.joint_angles_callback)

       # Initialize the JointState message
       self.joint_state_msg = JointState()
        # Actual names of joints (URDF)
                    self.joint_state_msg.name   =   ['joint2_to_joint1',   'joint3_to_joint2',
'joint4_to_joint3', 'joint5_to_joint4', 'joint6_to_joint5', 'joint6output_to_joint6']
       self.joint_state_msg.position = [0.0] * 6  # Initialize with zero angles for 6 joints

   def joint_angles_callback(self, msg):
   # Extract joint angles from the received message and convert them from degrees to radians
           joint_angles = [
           math.radians(msg.joint_1),
           math.radians(msg.joint_2),
           math.radians(msg.joint_3),
           math.radians(msg.joint_4),
           math.radians(msg.joint_5),
           math.radians(msg.joint_6)
       ]

       # Update the JointState message
       self.joint_state_msg.header.stamp = rospy.Time.now()  # Update timestamp
       self.joint_state_msg.position = joint_angles

       # Publish the JointState message
       self.joint_state_pub.publish(self.joint_state_msg)
```

Python Script ***joint_states.py***

```
<!-- move_group settings -->
 <arg name="pipeline" default="pilz_industrial_motion_planner" />
 <arg name="allow_trajectory_execution" default="true"/>
 <arg name="moveit_controller_manager" default="simple" />
 <arg name="fake_execution_type" default="interpolate"/>
 <arg name="max_safe_path_cost" default="1"/>
 <arg name="publish_monitored_planning_scene" default="true"/>

 <arg name="capabilities" default=""/>
 <arg name="disable_capabilities" default=""/>
```

Snippet from ***move_group.launch*** *file with PILZ as the choice of planner*

```
# Initialize MoveIt and ROS nodes
moveit_commander.roscpp_initialize(sys.argv)
# Initialize robot commander and scene interface
robot = moveit_commander.RobotCommander()

# Initialize MoveGroupCommander for your arm (replace 'arm_group' with your MoveGroup name
if different)
group_name = "arm_group"  # Ensure this matches your MoveIt configuration
move_group = moveit_commander.MoveGroupCommander(group_name)
move_group.set_planner_id("LIN")

gripper = rospy.Publisher("/mycobot/gripper_status", MycobotGripperStatus, queue_size=10)
```

*Snipped of `low_level_controller.py`, where “**LIN**” is the choice of Motion Commander*

```
# Function to move to a specified (x, y, z) position
def plan_to_xyz(x, y, z):
   current_state = robot.get_current_state()
   move_group.set_start_state(current_state)

   # Set up a Pose target at the desired location
   pose_goal = geometry_msgs.msg.Pose()
   pose_goal.position.x = x
   pose_goal.position.y = y
   pose_goal.position.z = z

   # Apply a 180-degree rotation around X-axis (downward)
   q1 = quaternion_from_euler(pi, 0, 0)  # Pi radians (180 degrees) around X
   q2 = quaternion_from_euler(0, 0, pi/4)
   q3 = quaternion_from_euler(-pi/16, 0, 0)      # 45° about Y
   q12 = quaternion_multiply(q1, q2)
   q = quaternion_multiply(q12, q3)


   pose_goal.orientation.x = round(q[0], 6)
   pose_goal.orientation.y = round(q[1], 6)
   pose_goal.orientation.z = round(q[2], 6)
   pose_goal.orientation.w = round(q[3], 6)

   # Set the target pose for the MoveGroup
   move_group.set_pose_target(pose_goal)

   # Plan the trajectory to the target pose
   plan = move_group.plan()

   # Check if planning succeeded
```

```
    if not plan[0]:
        rospy.logwarn("Planning failed for the target pose.")
        return False

    move_group.execute(plan[1], wait=True)
    rospy.loginfo("Trajectory executed successfully.")
    return True
```

Snipped of ***low_level_controller.py****, where we plan and execute trajectories*

```
<?xml version="1.0"?>
<robot  xmlns:xacro="http://www.ros.org/wiki/xacro" name="firefighter" >
 <xacro:property name="width" value=".2" />


 <link name="g_base">
   <visual>
     <geometry>
       <mesh filename="package://mycobot_description/urdf/mycobot_pi/G_base.dae"/>
     </geometry>
     <origin xyz = "0.0 0 -0.03" rpy = "0 0 1.5708"/>
   </visual>
   <collision>
     <geometry>
       <mesh filename="package://mycobot_description/urdf/mycobot_pi/G_base.dae"/>
     </geometry>
     <origin xyz = "0.0 0 -0.03" rpy = "0 0 1.5708"/>
   </collision>
 </link>


 <link name="joint1">
   <visual>
     <geometry>
      <!--- 0.0 0 -0.04  1.5708 3.14159-->
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint1_pi.dae"/>
     </geometry>
   <origin xyz = "0.0 0 0 " rpy = " 0 0 0"/>
   </visual>
   <collision>
     <geometry>
      <!--- 0.0 0 -0.04  1.5708 3.14159-->
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint1_pi.dae"/>
     </geometry>
   <origin xyz = "0.0 0 0 " rpy = " 0 0 0"/>
   </collision>
 </link>

 <link name="joint2">
```

```
   <visual>
     <geometry>
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint2.dae"/>
     </geometry>
   <origin xyz = "0.0 0 -0.06096 " rpy = " 0 0 -1.5708"/>
   </visual>
     <collision>
     <geometry>
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint2.dae"/>
     </geometry>
   <origin xyz = "0.0 0 -0.06096 " rpy = " 0 0 -1.5708"/>
   </collision>
 </link>



 <link name="joint3">
   <visual>
     <geometry>

      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint3.dae"/>
     </geometry>
   <origin xyz = "0.0 0 0.03256 " rpy = " 0 -1.5708 0"/>
   </visual>
     <collision>
       <geometry>
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint3.dae"/>
     </geometry>
   <origin xyz = "0.0 0 0.03256 " rpy = " 0 -1.5708 0"/>
   </collision>
 </link>


 <link name="joint4">
   <visual>
     <geometry>
      <!--- 0.0 0 -0.04 -->
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint4.dae"/>
     </geometry>
   <origin xyz = "0.0 0 0.03056 " rpy = " 0 -1.5708 0"/>
   </visual>
     <collision>
     <geometry>
      <!--- 0.0 0 -0.04 -->
      <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint4.dae"/>
     </geometry>
   <origin xyz = "0.0 0 0.03056 " rpy = " 0 -1.5708 0"/>
   </collision>
 </link>


<link name="joint5">
```

```
  <visual>
    <geometry>
     <!--- 0.0 0 -0.04 -->
     <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint5.dae"/>
    </geometry>
  <origin xyz = "0.0 0 -0.03356 " rpy = " 0 -1.5708 1.5708"/>
  </visual>
    <collision>
    <geometry>
     <!--- 0.0 0 -0.04 -->
     <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint5.dae"/>
    </geometry>
  <origin xyz = "0.0 0 -0.03356 " rpy = " 0 -1.5708 1.5708"/>
  </collision>
</link>


<link name="joint6">
  <visual>
    <geometry>
     <!--- 0.0 0 -0.04 -->
     <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint6.dae"/>
    </geometry>
  <origin xyz = "0 0.00 -0.038 " rpy = " 0 0 0"/>
  </visual>
    <collision>
    <geometry>
     <!--- 0.0 0 -0.04 -->
     <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint6.dae"/>
    </geometry>
  <origin xyz = "0 0.00 -0.038 " rpy = " 0 0 0"/>
  </collision>
</link>


<link name="joint6_flange">
  <visual>
    <geometry>
     <!--- 0.0 0 -0.04 -->
     <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint7.dae"/>
    </geometry>
  <origin xyz = "0.0 0 -0.012 " rpy = " 0 0 0"/>
  </visual>
    <collision>
    <geometry>
     <!--- 0.0 0 -0.04 -->
     <mesh filename="package://mycobot_description/urdf/mycobot_pi/joint7.dae"/>
    </geometry>
  <origin xyz = "0.0 0 -0.012 " rpy = " 0 0 0"/>
  </collision>
</link>
```

```
<joint name="g_base_to_joint1" type="fixed">
  <axis xyz="0 0 0"/>
  <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
  <parent link="g_base"/>
  <child link="joint1"/>
  <origin xyz= "0 0 0" rpy = "0 0 0"/>
</joint>


<joint name="joint2_to_joint1" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
  <parent link="joint1"/>
  <child link="joint2"/>
  <origin xyz= "0 0 0.13956" rpy = "0 0 0"/>
</joint>


<joint name="joint3_to_joint2" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
  <parent link="joint2"/>
  <child link="joint3"/>
  <origin xyz= "0 0 -0.001" rpy = "0 1.5708 -1.5708"/>
</joint>

<joint name="joint4_to_joint3" type="revolute">
  <axis xyz=" 0 0 1"/>
  <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
  <parent link="joint3"/>
  <child link="joint4"/>
  <origin xyz= "  -0.1104 0 0   " rpy = "0 0 0"/>
</joint>


<joint name="joint5_to_joint4" type="revolute">
  <axis xyz=" 0 0 1"/>
  <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
  <parent link="joint4"/>
  <child link="joint5"/>
  <origin xyz= "-0.096 0 0.06462" rpy = "0 0 -1.5708"/>
</joint>


<joint name="joint6_to_joint5" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
  <parent link="joint5"/>
  <child link="joint6"/>
  <origin xyz= "0 -0.07318 0" rpy = "1.5708 -1.5708 0"/>
</joint>
```

```
 <joint name="joint6output_to_joint6" type="revolute">
   <axis xyz="0 0 1"/>
   <limit effort = "1000.0" lower = "-3.14" upper = "3.14159" velocity = "0"/>
   <parent link="joint6"/>
   <child link="joint6_flange"/>
   <origin xyz= "0 0.0456 0" rpy = "-1.5708 0 0"/>
 </joint>
</robot>
```

MyCobot280pi URDF - ***mycobot_urdf.urdf***

```
class JointTrajectoryActionServer:
   def __init__(self):
       rospy.init_node('joint_trajectory_action_server')

       # Action server setup
       self.server = actionlib.SimpleActionServer(
           'arm_group/follow_joint_trajectory',
           FollowJointTrajectoryAction,
           execute_cb=self.execute_callback,
           auto_start=False)

       self.server.start()

       self.pub = rospy.Publisher("/mycobot/angles_goal", MycobotSetAngles, queue_size=10)

       rospy.loginfo("JointTrajectoryActionServer started.")

   def execute_callback(self, goal):
       feedback = FollowJointTrajectoryFeedback()
       result = FollowJointTrajectoryResult()

                     rospy.loginfo("Received   goal   with   %d   points   in   trajectory.",
len(goal.trajectory.points))
       r = rospy.Rate(30)

       # Loop through each trajectory point and execute
       for point in goal.trajectory.points:
           feedback.desired = point
           self.server.publish_feedback(feedback)

           angles = list(point.positions)

           mycobot_sendAngles = MycobotSetAngles()
           mycobot_sendAngles.joint_1 = math.degrees(angles[0])
           mycobot_sendAngles.joint_2 = math.degrees(angles[1])
           mycobot_sendAngles.joint_3 = math.degrees(angles[2])
```

```
            mycobot_sendAngles.joint_4 = math.degrees(angles[3])
            mycobot_sendAngles.joint_5 = math.degrees(angles[4])
            mycobot_sendAngles.joint_6 = math.degrees(angles[5])
            mycobot_sendAngles.speed = 40

            self.pub.publish(mycobot_sendAngles)
            # Wait briefly to avoid overlapping trajectories
            r.sleep()  # Adjust delay as necessary

        rospy.loginfo("Trajectory executed successfully.")
        result.error_code = result.SUCCESSFUL
        self.server.set_succeeded(result)
```

JointTrajectoryActionServer - ***joint_trajectory.py***

```
<!-- Run joint_trajectory.py -->
    <node    pkg="gamesmanros"    type="joint_trajectory.py"    name="joint_trajectory_node"
output="screen">
   <!-- <param name="use_sim_time" value="true" /> -->
 </node>
```

Snippet of action server launch file - ***tf_bringup.launch***

```
image_width: 1280
image_height: 720
camera_name: head_camera
camera_matrix:
 rows: 3
 cols: 3
 data: [1308.70048,    0.     ,  638.84558,
           0.     , 1321.24676,  281.60301,
           0.     ,    0.     ,    1.     ]
distortion_model: plumb_bob
distortion_coefficients:
 rows: 1
 cols: 5
 data: [0.065854, -0.103668, -0.023520, 0.003236, 0.000000]
rectification_matrix:
 rows: 3
 cols: 3
 data: [1., 0., 0.,
        0., 1., 0.,
        0., 0., 1.]
projection_matrix:
 rows: 3
 cols: 4
 data: [1333.49377,    0.     ,  641.41345,    0.     ,
           0.     , 1331.89819,  269.28536,    0.     ,
           0.     ,    0.     ,    1.     ,    0.     ]
```

Camera Calibration File - ***`head_camera.yaml (~/.ros/camera_info)`***

```
<launch>
 <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
   <param name="video_device" value="/dev/video0" />
   <param name="image_width" value="1280" />
   <param name="image_height" value="720" />
   <param name="pixel_format" value="yuyv" />
   <param name="camera_frame_id" value="usb_cam" />
   <param name="io_method" value="mmap"/>
 </node>
</launch>
```

Camera Launch File - ***`usb_cam.launch`***

```
#!/usr/bin/env python3
import rospy
import tf
import math

if __name__ == '__main__':
   rospy.init_node('dynamic_tf_broadcaster')
   br = tf.TransformBroadcaster()
   rate = rospy.Rate(10.0)
   while not rospy.is_shutdown():
       br.sendTransform((0.0, -0.04, -0.0),
                        (0.70710678, -0.70710678, 0, -0),
                        rospy.Time.now(),
                        "usb_cam",
                        "joint6_flange")
       rate.sleep()
```

*Snippet of **`transformBroadcast.py`***

```
<node name="camera_tf" pkg="gamesmanros" type="transformBroadcast.py" output="screen" />
```

*Launching transformBroadcast in **`tf_bringup.launch`***

```
def pick_best_move(moves):
   position_values = {}
   for i in range(len(moves)):
       if moves[i]['moveValue'] not in position_values:
           position_values[moves[i]['moveValue']] = [moves[i]['autoguiMove']]
       else:
           position_values[moves[i]['moveValue']].append(moves[i]['autoguiMove'])

   if 'win' in position_values and len(position_values['win']) > 0:
       return position_values['win'][0]
   elif 'draw' in position_values and len(position_values['draw']) > 0:
       return position_values['draw'][0]
   elif 'lose' in position_values and len(position_values['lose']) > 0:
       return position_values['lose'][0]
   else:
       print('error: in pick_best_move')
       exit()
```

Code Snippet for choosing optimal move from UWAPI server response

```
class RobotControl:
   # XYZ Values in mm
   def __init__(self, board_size=150, dim=3, y_offset=100, pickup_z=135, lift_z=185):
       self.board_size = board_size
       self.dim = dim
       self.scaling = self.board_size/(self.dim)
       self.x_offset = (self.board_size/2)
       self.y_offset = y_offset
       self.pickup_z = pickup_z
       self.lift_z = lift_z / 1000

   def svg_to_real(self, svg_coord):
       T = np.array([[1, 0, 0],
                    [0, -1, self.dim+1],
                    [0, 0, 1]])

       coord = np.array([svg_coord[0], svg_coord[1], 1])

       real_coord = np.dot(T, coord.T)
       real_coord[1] = abs(real_coord[1])
       return [real_coord[0], real_coord[1]]

   #gripper: Open 0, Close 1
   def play(self, before, after):
       before = self.svg_to_real(before)
       x = (before[0] * self.scaling) - self.x_offset
       y = (before[1] * self.scaling) + self.y_offset
       z = self.pickup_z

       # Convert XYZ Values to meters
       x = x / 1000
       y = y / 1000
       z = z / 1000

       after = self.svg_to_real(after)
       after_x = (after[0] * self.scaling) - self.x_offset
       after_y = (after[1] * self.scaling) + self.y_offset
       after_z = self.pickup_z

       # Convert XYZ Values to meters
       after_x = after_x / 1000
       after_y = after_y / 1000
       after_z = after_z / 1000

       print("Before: ", (x, y, z), " | ", "After: ", (after_x, after_y, after_z))

       gripper_status("open")
       time.sleep(0.5)

       plan_to_xyz(x, y, self.lift_z)
       time.sleep(1)
       plan_to_xyz(x, y, z)
```

```
        gripper_status("close")
        time.sleep(0.5)

        plan_to_xyz(x, y, self.lift_z)
        time.sleep(1)
        plan_to_xyz(after_x, after_y, self.lift_z)
        time.sleep(1)
        plan_to_xyz(after_x, after_y, after_z)
        time.sleep(1)

        gripper_status("open")
        time.sleep(0.5)

        plan_to_xyz(after_x, after_y, self.lift_z)
        time.sleep(1)
```

*RobotControl class - **`robotControl.py`***

```
class ARTagListener:
    def __init__(self):
        self.positions = {}
        self.lock = threading.Lock()

        # Subscribe to all AR tag topics
        for i in range(18):
            tag_id = f"ar_marker_{i}"
            rospy.Subscriber(f"/piece_position/{tag_id}", Pose, self.make_callback(tag_id))

    def make_callback(self, tag_id):
        def callback(msg):
            with self.lock:
                self.positions[tag_id] = [msg.position.x, msg.position.y]
        return callback

    def get_pose(self, tag_id):
        with self.lock:
            return self.positions.get(tag_id, None)

    def get_all(self):
        with self.lock:
            return dict(self.positions)  # Shallow copy
```

*ArTagListener Class - **`artag_listener.py`***