Robotics Project: Vision-based target Pursuit and Engagement

Alessio Tommasi
alessio.tommasi@usi.ch

Jonas Knupp
jonas.knupp@usi.ch

***Abstract*—In this project, we design a controller for the RoboMaster S1, acting as the chaser, to pursue and engage a RoboMaster EP, serving as the evader, using a simulated laser within a 20 by 20 meter room. The chaser relies solely on visual input from its onboard camera to track the evader, which is programmed to avoid turning toward the chaser. We visualize the trajectories of both robots in a 2D plot and analyze the accuracy of the chaser's and evader's odometries compared to ground truth positions. We evaluate our approach in two CoppeliaSim scenarios: one with an empty room and another with a room that includes a wall as an obstacle. In both scenarios the chaser could successfully engage the evader. Our results also show that the RoboMasters' odometries provide reasonably accurate pose estimates in both scenarios.**

I. Introduction

In this work, we develop a controller for the RoboMaster S1 that autonomously tracks, pursues, and engages a RoboMaster EP in CoppeliaSim using input from a single camera. The RoboMaster S1, referred to as the chaser, features a turret mounted on a gimbal that can be controlled independently of the chassis. The RoboMaster EP, referred to as the evader, is equipped with its own controller that attempts to move it away from the chaser when possible.

To simplify the perception task — since computer vision is not the main focus of this work — we attach a massless blue sphere to the top of the evader, allowing the chaser to easily detect and track it. Because the robomaster-ros package does not provide built-in functionality for target engagement, we implemented a custom system in CoppeliaSim using scripts to simulate and visualize engagement and to detect whether a target has been hit. To simplify the simulation and avoid dealing with ballistic trajectories, we model the weapon as a laser rather than using projectiles.

II. Methods

The project is structured into two repositories: the robotics-project package[1] contains all relevant code except for a custom ROS 2 message which is in the robotics-project-messages package[2]. Given the number of nodes and topics involved, we include Figure 6 in the appendix to illustrate the communication patterns.

We begin by describing the engagement simulation. Next, we detail how computer vision is used for circle detection and distance estimation. Then, we describe the workings of the evader and finally we outline the behavior of the chaser. Note that when describing the pose of a robot, we use the format (x, y, θ) where x is the x-position of the robot in the world frame, y is the y-position in the world frame and θ is the angle between the robot's forward axis x and the world's x-axis (yaw). We perform all our experiments in a room of size 20×20 m centered around the origin of the world coordinate system.

A. Localization

Since localization is not the focus of this work and to avoid errors caused by odometry inaccuracies, we extended the Lua scripts for the evader and chaser robots in CoppeliaSim to publish their true position and orientation on the */rm[0,1]/real_position* topic. The published messages use the *Pose* message type. This is important as the chaser relies on knowing it's current position when it navigates to the estimated last known position of the evader. We denote the true pose of the chaser with L_c and the true pose of the evader with L_e.

B. Engagement Simulation

We assume that the chaser is equipped with a laser weapon with an effective range of 3 m. The engagement simulation is implemented using two Lua scripts: EngageVisScript and HitDetectionScript. The EngageVisScript is responsible for visualizing the laser. It draws a red line originating from the origin of the barrel with the same orientation and a length of 100, representing the inactive laser. The value 100 is chosen to ensure the line spans the entire simulation space, which measures only 20×20 meters.

Laser activation is controlled via CoppeliaSim's signal mechanism. Specifically, the EngageVisScript listens for the *laserShootSignal* emitted by the HitDetectionScript. Upon receiving this signal, the red line is replaced by a green line of length 3 m for 0.5 seconds to indicate that the laser has been fired. After this duration, the signal is reset by setting *laserShootSignal* to 0. The 0.5-second display is intended solely for human visibility.

The HitDetectionScript subscribes to the */rm0/engage* topic, which contains *Empty* messages to indicate an engagement attempt. Upon receiving such a message, the script checks whether the Euclidean distance between the barrel's origin and the center of the sphere is less than or equal to 3 m. If this condition is satisfied, it further verifies whether the angular distance between the barrel-to-sphere direction vector

[1]https://github.com/jonupp/robotics-project
[2]https://github.com/jonupp/project_messages

and the barrel's orientation is less than 1 degree. If both checks succeed, an *Empty* message is published on the */rm0/hit* topic, the *laserShootSignal* is set to 1, and the sphere's color is changed to green to indicate a successful hit.

C. Sphere Detection and Distance Estimation

The sphere detection and distance estimation is implemented in the vision_node. It subscribes to the */rm0/camera/image_color* and */rm0/camera/camera_info* topics to get the current image from the chaser's camera and the chaser's camera configuration. We created a custom message type *ImageFeatures* that contains the pixel position of the circle center, the barrel reference pixel, and the distance estimate. Note that ROS 2 assumes that the x-axis in the image space goes to the right and the y-axis in the image space goes dawn. Furthermore, the camera reference frame is right-handed with the camera pointing in the direction of the positive z-axis[3].

We chose a sphere because it looks like a circle from any perspective and circles can easily be detected using the Hough transform. First, the image from the chaser's camera is converted to a cv2 image using the CvBridge package. Then we mask the image such that only blue pixels have non-zero value. To make this easier, we convert the image to the HSV color space. To increase the stability of the Hough transform, we apply a gaussian blurring filter of size 9×9 and standard deviation in x direction of 2. Before we added the gaussian blurring, the Hough transform circle detection was very noisy because it had trouble finding the edge of the circle. The Hough transform returns the circle center (c_x, c_y) and the radius r for each detected circle. We only consider the largest detected circle. Despite extensive refinement, the Hough Transform occasionally detects spheres where none exist. To reduce false positives, we introduce a buffer of length 50 that stores the most recent 50 actual *ImageFeatures* messages—that is, the messages resulting from image processing, not those that were published. If 25 or fewer of the messages in the buffer indicate a detected circle, we classify the current detection as noise. In such cases, instead of publishing the detected message, we publish a message indicating that no circle was found.

Since our goal is to align the barrel, rather than the camera, with the center of the sphere, we need to compute the barrel position in the image space which we denote as barrel reference pixel b from now on. Intuitively, b lies at the center of the image along the x-axis, but slightly below the vertical center along the y-axis, because the camera is mounted a few centimeters above the barrel. Unfortunately, the exact vertical offset between the camera and the barrel is not known. We estimate this offset to be 3.5 cm. We define P as the projection matrix that maps points from the world frame to the image frame, accounting for the vertical offset between the camera and the barrel. The y-offset is positive because the y-axis in image space points downward.

[3]https://docs.ros.org/en/rolling/p/image_pipeline/camera_info.html

$$P = K \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0.035 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The barrel reference pixel is then computed by projecting the point $[0, 0, 1, 1]^T$ using P, and normalizing the result by its last coordinate.

We define the sphere above the evader to have diameter 0.5 m in CoppeliaSim. However, we observed improved results for the distance estimation when assuming a sphere diameter of 0.42 m. Therefore, all calculations are based on a diameter of 0.42 m. By measuring its diameter in the image space, we can estimate the distance from the camera to the center of the sphere in the world frame by using the inverse of the intrinsic camera matrix K. To obtain the diameter in image space, we identify the pixel coordinates of the leftmost point of the sphere as $l = (c_x - r, c_y)$ and the rightmost point as $r = (c_x + r, c_y)$ in the current image.

Next, we project these points into the camera reference frame by augmenting them with a homogeneous coordinate and applying the inverse intrinsic matrix:

$$\tilde{l} = K^{-1} \cdot \begin{bmatrix} l \\ 1 \end{bmatrix}, \quad \tilde{r} = K^{-1} \cdot \begin{bmatrix} r \\ 1 \end{bmatrix}.$$

Finally, knowing the real-world diameter of the sphere and its corresponding diameter in the image space, we estimate the depth z, which approximates the distance from the camera to the sphere center, as follows:

$$\begin{aligned} 0.42 &= \sqrt{(z\tilde{r}_x - z\tilde{l}_x)^2 + (z\tilde{r}_y - z\tilde{l}_y)^2 + (z\tilde{r}_z - z\tilde{l}_z)^2} \\ \iff 0.42 &= z\sqrt{(\tilde{r}_x - \tilde{l}_x)^2 + (\tilde{r}_y - \tilde{l}_y)^2 + (\tilde{r}_z - \tilde{l}_z)^2} \\ \iff z &= \frac{0.42}{\sqrt{(\tilde{r}_x - \tilde{l}_x)^2 + (\tilde{r}_y - \tilde{l}_y)^2 + (\tilde{r}_z - \tilde{l}_z)^2}} \end{aligned}$$

The SphereCameraDistanceScript publishes the true distance between the camera and the center of the sphere in the */sphere_camera_distance* topic. This allows us to calculate the distance estimation error by subtracting the estimated distance from the real one.

The chaser-controller node publishes the chaser's current state to the */rm0/chaser_state* topic. The vision-node subscribes to this topic to track the chaser's state in real time and incorporate it into the visualization.

The vision_node produces a video stream from the images received from the chaser, augmented with visual cues. The detected circle's outline is shown in green, while its center, as well as the leftmost and rightmost points, are marked in red. Additionally, the estimated distance to the target and the distance error are displayed on the bottom right of the image. The current state of the chaser is shown on the top right of the image. A yellow cross indicates the barrel reference pixel. An example output from the vision_node is shown in figure 1.

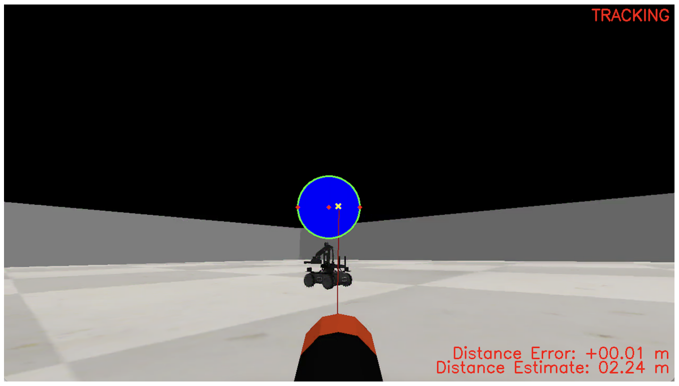

Fig. 1. Example of an image from the vision_node where the chaser is in the *Tracking* state.

D. Evader

The evader is designed to explore the entire room using a random exploration strategy. It includes a collision avoidance mechanism, primarily to prevent crashes into walls. To implement this functionality, we rely on data from four ToF sensors. Additionally, the evader has functionality to avoid moving toward the chaser's location to increase the difficulty of the chaser's task. For simplicity, we assume that the evader always knows the exact location of the chaser. This is implemented by subscribing to the */rm0/real_position* topic.

- *Exploration:* To ensure a smooth exploration of the environment over time, the evader operates with 3 levels of randomness.
  The first is the evader's probability to come back to the moving state after meeting the right condition.
  The second is the direction in which the evader rotates and the last one is the linear velocity that the evader uses to escape.
  Each of these three behaviors is controlled using an independent uniform distribution.
- The *prediction algorithm* is running in a callback independent from the state;
  it estimates the evader's next position.
  The algorithm is based on 2 states: *Rotate* in which the robot spins in place to chose a new movement direction and *Move Forward* state to evaluate the avoidance.
- At each time step, the evader performs the following calculations:
  - calculate the distance between itself and the chaser using information from the */rm0/real_position* and */rm1/real_position* topics.
  - *prediction step:*

    $$\text{next\_position} = \text{current\_position} + v \cdot \vec{o} \cdot \Delta t$$

    where v is the linear velocity of the evader, $\vec{o}$ is the orientation vector of the evader, and Δt is the length of a time step.
    The result is a vector containing the estimated x and y position of the evader in the immediate future.

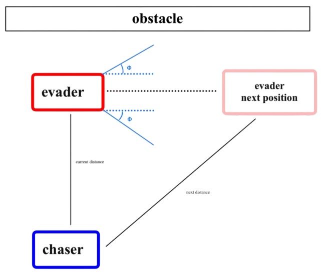

Fig. 2. Progression steps of the evader

Once the next_position and the current_position of the chaser and the evader_position are available, the algorithm calculates both the current Euclidean distance between the chaser and evader and the projected distance if the chaser maintains its current path.
After computing these distances, the algorithm verifies that the evader's projected distance is greater than its current distance.
This ensures that the evader does not unintentionally move closer to the chaser (see Figure 2).
When rotating in place, the evader keeps rotating until it identifies a direction in which moving forward would increase its distance from the chaser.
Once such a direction is found, it switches back to the *Move Forward* state.

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[loop above, "b"] \arrow[r, "a"] & q_1 \arrow[loop above, "c"] \arrow[l, bend left, "d"]
\end{tikzcd}$$

- q_0: *Rotate* — each time it is called, rotate the robot by θ
- q_1 : *Move Forward* — execute a step forward

- a: no obstacle AND valid direction detected
- b: ToF trigger OR next_distance $<$ current_distance
- c: no obstacle AND valid direction
- d: ToF trigger OR next_distance $<$ current_distance

E. Chaser

Since the RoboMaster S1 model in CoppeliaSim does not have ToF sensors available per default, we added four ToF sensors in the same positions and orientations as in the evader. The controller of the chaser is implemented as a single node chaser_controller. It is implemented as a state machine as shown in figure 3. In the subsequent paragraphs, we describe the behavior of the robot in each state.

a) Returning: This is the starting state of the robot. The robot will navigate to position (0,0,0), which is the center of

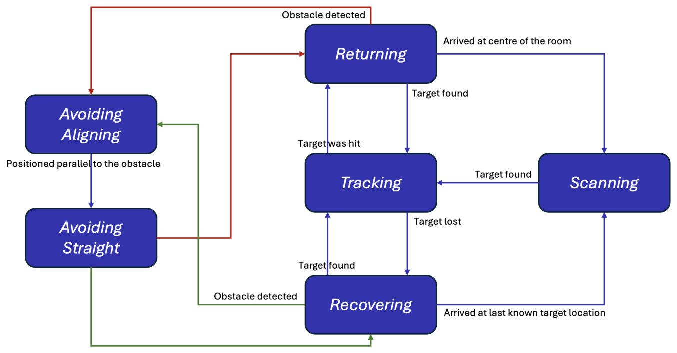

Fig. 3. State machine of the chaser.

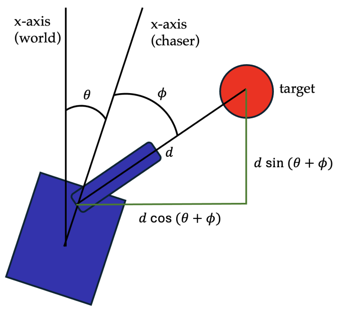

Fig. 4. Calculation of the estimated offset of the evader from the chaser's current position.

the room facing towards the x-axis of the world frame. Once the robot is in the desired pose, it transitions into the *Scanning* state. At any time, if the chaser measures a distance of less than 0.2 m on either the front left or front right ToF sensor it transitions to the *Avoiding Aligning* state.

b) Scanning: The chaser rotates to the left and tries to keep the barrel aligned with the the chassis. This makes the robot scan the entire room (if there are no obstacles) and thus it is likely that at some point the evader will be detected. Once the chaser detects the evader (check whether the vision_node last reported a valid circle center) it transitions to the *Tracking* state.

c) Tracking: The robot aims to keep the barrel aligned with the barrel reference pixel b, provided by the vision_node. To achieve this, a proportional controller adjusts the gimbal on which the barrel is mounted. However, a challenge arises: when the distance d between the barrel and sphere center is large, the same pixel offset results in a smaller required gimbal movement than when the distance is short. To account for this, we adjust the proportional gain dynamically for both the gimbal yaw and pitch by linearly interpolating between K_{near} and K_{far}, based on d:

$$t = \frac{d - d_{\text{near}}}{d_{\text{far}} - d_{\text{near}}}$$
$$K_p = \min\bigl(K_{\text{near}} + t \cdot (K_{\text{far}} - K_{\text{near}}),\ 0.005\bigr)$$

where d_{near} and d_{far} are constants defining what distances are considered "near" and "far". We use $d_{\text{near}} = 2$ and $d_{\text{far}} = 10$. The $\min$ ensures that K_p does not become too large.

While tracking, the robot also tries to align the chassis to the same orientation as the barrel so that it pursuits the target. We did not implement any obstacle avoidance in this state since it would be very hard to balance avoiding an obstacle and keeping track of the target. However, this means that the robot could potentially crash into an obstacle in this state. If the robot loses track of the evader, that is, if the vision_node reports an invalid circle center, the robot transitions into the *Recovering* state.

d) Recovering: In this state, the chaser navigates to the estimated last known position of the evader. The estimated pose of the evader $\hat{L}_e$ is calculated as following:

$$\hat{L}_e = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \circ L_c + \begin{bmatrix} d \cdot \cos(\theta + \phi) \\ d \cdot \sin(\theta + \phi) \\ 0 \end{bmatrix}$$

where d is the estimated distance to the target and ϕ is the yaw of the gimbal with respect to the chaser's chassis. Notice that the last component is 0 because we do not estimate the orientation of the evader. For more clarity, we show the situation in figure 4. The chaser publishes the evader's estimated last known position in the */rm0/estimated_target_position* topic.

Once the chaser estimates that it is within 3.5 m range of the evader and the absolute difference in both x and y between the circle center and the barrel reference pixel is less than 8 pixels, the chaser starts engaging the evader with its laser. Note that we implemented a cooldown of 1 seconds for firing the laser. Engaging the laser means that the chaser publishes an *Empty* message in the */rm0/engage* topic. If the target was hit, that is, if the chaser receives an *Empty* message in the */rm0/hit* topic, the chaser transitions to the *Returning* state.

At any time, if the chaser measures a distance of less than 0.2 m on either the front left or front right ToF sensor it transitions to the *Avoiding Aligning* state.

e) Avoiding Aligning: We assume the obstacle to be a wall. In this state, the chaser aligns itself parallel to the wall by rotating until the distances reported by the front and rear ToF sensors on one side are equal. Specifically, it turns right until the front left and back left sensors match, or left until the front right and back right sensors do. Since there are two possible orientations that satisfy this condition, the chaser selects the one requiring the smallest rotation. This approach also tends to align the robot more closely with the estimated direction of the evader. Once the chaser is approximately parallel to the wall, it transitions to the *Avoiding Straight* state.

f) Avoiding Straight: If the chaser turned left during wall alignment, it moves forward until the back right ToF sensor reports a distance greater than 0.6 meters. Conversely, if it turned right, it continues moving forward until the back left ToF sensor exceeds the same threshold. Once this condition is met, the chaser transitions back to the state it was in prior to initiating the avoidance maneuver — either *Returning* or *Recovering*.

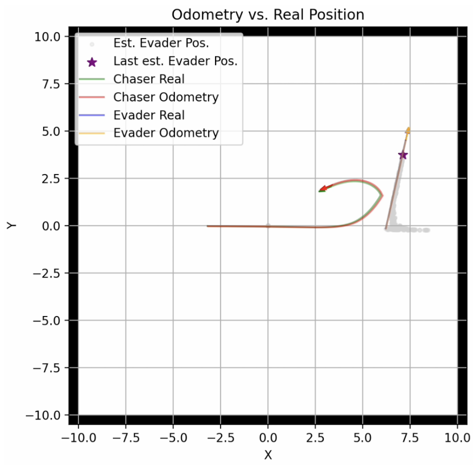

Fig. 5. The poses obtained from the odometry of both the chaser and evader are very similar to the real poses obtained from CoppeliaSim. After a brief period of tracking, the chaser can estimate the evader's position with reasonable accuracy using only visual data from its camera

F. *Position Visualizer*

We develop the position_visualizer node to monitor two types of errors: the difference between the robot's odometry and the ground-truth pose from CoppeliaSim, and the discrepancy in the chaser's estimation of the evader's position in the *Tracking* state. Figure 5 shows how much the trajectories from the odometry and the real position obtained from CoppeliaSim deviate for both the chaser and evader in a simple scenario without obstacle. The figure also shows the chaser's estimate for the evader's position while the chaser is in the *Tracking* state.

III. Results

In the first scenario, the chaser and evader are in an otherwise empty room. We perform several experiments with different starting poses for both the chaser and evader. In all experiments the chaser could eventually hit the evader's sphere and return to the center of the room. We provide a video of one such experiment[4].

In the second scenario, the chaser and evader are placed in a room with a long and wide wall. This setup is intended to evaluate the chaser's ability to avoid obstacles. The evader is positioned near the edge of the wall, just about to disappear behind it, while the chaser starts in the back of the room, positioned so that it can barely glimpse the evader before it vanishes. As anticipated, the chaser briefly detects the evader, then loses visual contact as the evader moves behind the wall. The chaser then navigates toward the last known position of the evader while avoiding crashing into the wall. Before reaching the last known position of the evader, the chaser regains track of the evader and eventually hits it. In the end, the chaser moves to the center of the room. A video of this experiment is available.[5]

[4]https://shorturl.at/9WcI1
[5]https://shorturl.at/aK4kr

A. *Limitations*

As previously mentioned, we use the ground-truth positions of the robots provided by CoppeliaSim. In a real-world setup, a separate localization system would be required. Additionally, we did not implement obstacle avoidance while the chaser is in the *Tracking* state, in order to avoid the complexity of balancing target pursuit with obstacle avoidance. A more robust visual target detection would be desirable since the Hough transform sometimes loses track of the sphere if it is far away. Furthermore, in some situations the chaser keeps aiming slightly behind the sphere center. We suspect this is because of delay. We did not have time to address this issue, but it could likely be resolved through predictive aiming.

IV. Discussion and Conclusion

In this work, we presented a controller for the RoboMaster S1 that autonomously pursues and engages a RoboMaster EP in two distinct scenarios within CoppeliaSim, using only visual input from the chaser's camera. We demonstrated that reliable distance estimates can be obtained using information derived from the Hough transform of the detected circle. Finally, we showed that, within the context of our scenarios, both the RoboMaster S1 and RoboMaster EP provide accurate odometry when compared to their true poses.

There are many potential directions for future research. We initially considered using the RoboMaster S1 detector API to identify the evader, as it can be configured to return bounding boxes around RoboMasters in the image. However, we ultimately chose not to use it due to the lack of a reliable method for estimating the distance between the chaser and the evader based solely on the bounding box. An interesting direction for future work would be to simulate firing physical projectiles instead of using a laser to engage the target. By applying elementary physics, one could compute the required barrel orientation to account for the projectile's ballistic trajectory. The difficulty could be adjusted by varying the projectile's properties, such as initial velocity and mass.

Appendix

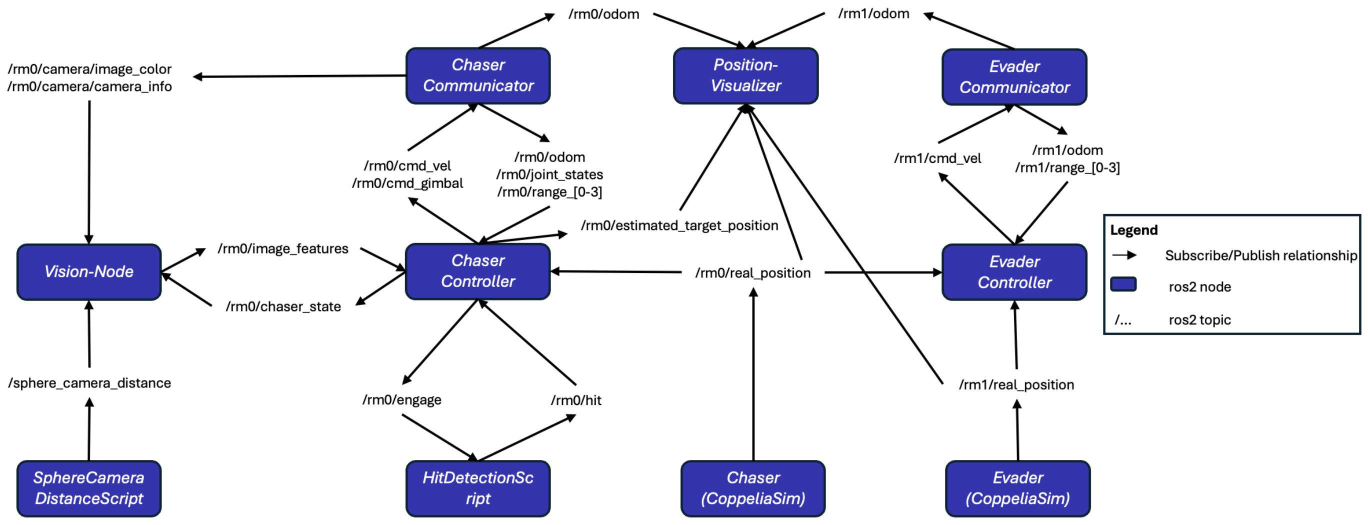

Fig. 6. Message flow between the different ROS 2 nodes. Note that the Chaser Communicator and Evader Communicator nodes are from the robomaster_ros package. The SphereCameraDistanceScript, HitDetectionScript, Chaser (CoppeliaSim), and Evader (CoppeliaSim) nodes are not explicit nodes. These are Lua scripts executed within CoppeliaSim that interface with ROS 2 through the Lua API.