Byung-Ji Robot (Air-Hockey penalty kick)

Instructor : prof. Young-Keun Kim

Team member : Dong-min Kim, Eun-chan Kim, Seung-eun Hwang

Date : Spring semester, 2023

Part : Automation part in Industrial AI & Automation class

Project : #2

---

1. Introduction

This Repository includes a **Tutorial** for the final project of the Industrial AI & Automation course conducted in the first semester of 2023 at Handong Global University, namely, the **Robot Byeong-Gi (Airhockey Robot)**.

The Automation project is designed to gain experience in controlling a robotic arm by configuring a scenario where the Indy-10(Neuromeka)'s 6-axis collaborative robotic arm performs a specific task, and aims to promote the Department of Mechanical Control Engineering by providing a positive experience through robot experiences for lower-grade students.

We have selected the project theme with a focus on **"participation"** and **"feasibility"**. We judged that an **Air Hockey Robot** would allow participants to easily play the game and could be implemented with a collaborative robot.

The primary objective was set to predict the moving path of an object using image processing techniques, and move the robotic arm to the predicted location to block the object.

2. Requirements

2.1. Software

- OS: ubuntu 20.04 version
- ROS noetic
- python 3.9
- opencv 4.7.0

2.2. Hardware

- Camera sensor : Logitech Brio 4K (EA : 1)

Figure 1. Logitech Brio 4K

- Robot arm : indy 10 (EA : 1)

Figure 2. Indy 10 Robot Arm

- Specifications of Indy 10 robot
  - 6 degree of freedom
  - 10 [kg] payload
  - 175 deg
  - joint 1,2 : 60 deg/s, joint 3,4,5,6 : 90 deg/s
  - Maximum reach : 1000mm
- Air hockey peddle

3D modeling was conducted using Fusion 360. The size of the robot arm's bracket was measured in real world, and modeling was done accordingly to create the handle.

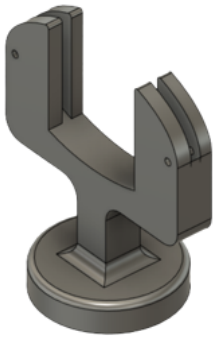


Figure 3. Air Hockey Peddle 3D Model

3D Model Download link

- AirHockey Table

We purchased and used an air hockey table in the 300,000 won range. A table appropriate for the height of Indy 10 was selected. Considering the speed of the robot, we purchased a product with a vertical length of more than 150cm.

Figure 4. Air Hockey Table

Purchase link

- Vacuum gripper

Figure 5. Vacuum Gripper

3. Process

3.1. Process Structure

The structure of the program consists of two parts: image processing and robot operation. In the camera, segmentation was carried out using HSV, and the expected path was calculated based on the position change according to the frame change of the detected object. The average slope of the object's position change was used to predict the angle of incidence and reflection of the Puck. The robot area is divided into three parts: left, center, and right. Depending on which area the end point of the predicted path is located, a movement Flag is passed to the robot. If an object crosses the goal line and is not detected for a certain number of frames, it is judged that a goal has been scored, and a Flag for the robot to pick up the Puck is passed.

Through the Flag received from the camera part, the robot moves to the expected position where the Puck will arrive. The positions of the three areas were pre-set through the robot's absolute coordinates, and the action of picking up the Puck was also set using absolute coordinates. By repeating this process, the code was constructed to sustain the game of sudden death.

3.2. System Flow Chart

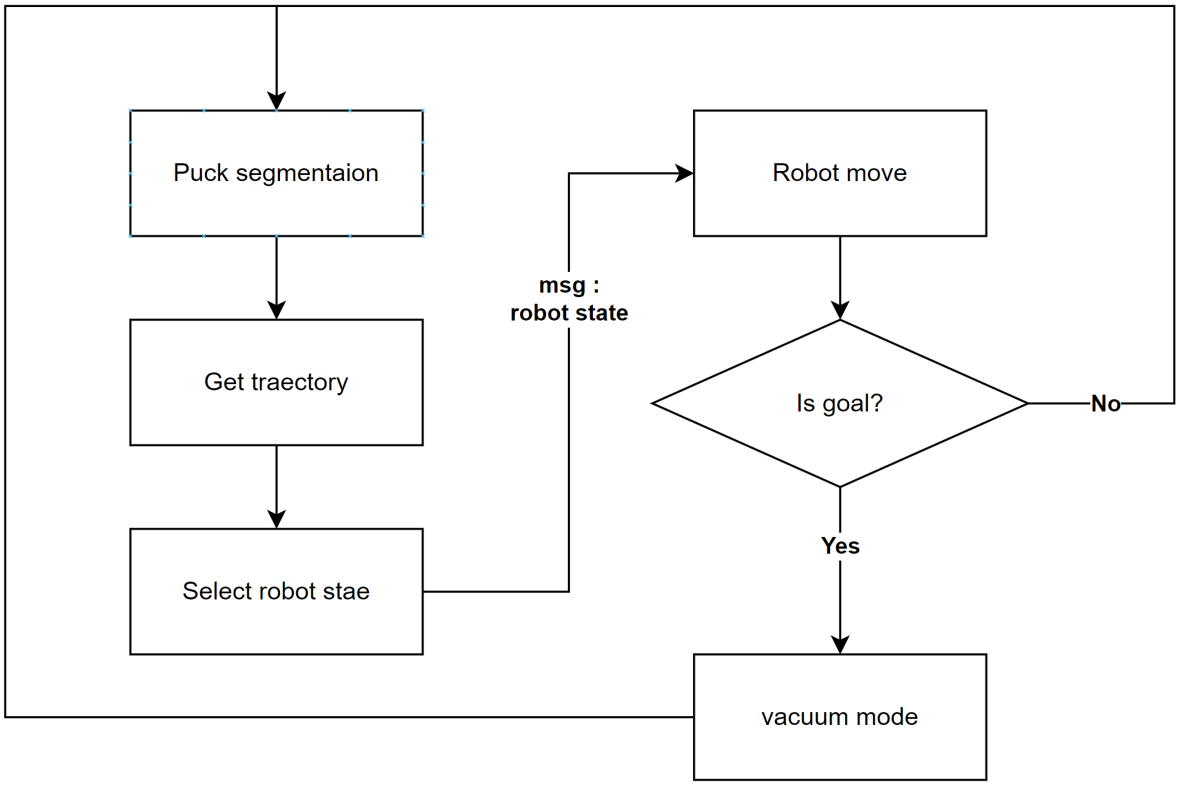
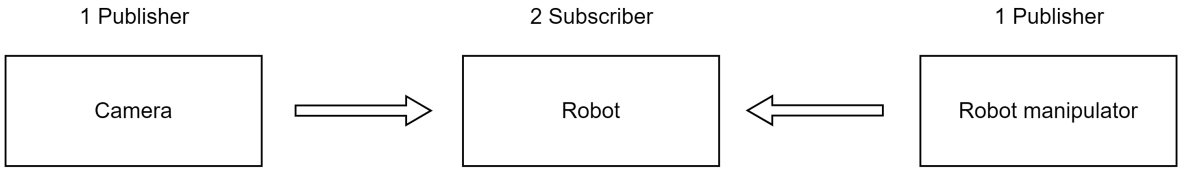


Figure 6. Flow Chart

3.3. Entire System Process



- The camera continuously calculates the segmentation information of the puck.
- The destination the robot should go to is transmitted via a publisher flag.
- When the robot receives location information via a subscriber, the robot moves.
- When the robot moves, once it receives a flag indicating the movement is complete, the robot is ready to receive the next location.

3.4. Image Processing

3.4.1 Detection of Puck

The image processing algorithm proceeds as follows: The blue puck is detected by employing HSV color segmentation to obtain its contour. As the distance between the camera, which is fixed in position and profile, and the hockey table is constant, the size of the puck remains uniform. To enhance the detection rate of the puck, the contour area is restricted between 200 and 380 units.

```
for cnt in contours:
    area = cv.contourArea(cnt)
    # filter outlier contours

    if 200 <= area <= 380:
        self.x, self.y, w, h = cv.boundingRect(cnt)
```

3.4.2. Trajectory Prediction

Due to the slow operating speed of the robot and data transmission rate through ROS, it is imperative to predict the trajectory of the puck early on. The algorithm is set to draw the trajectory once the puck has been detected and specific conditions are met: the center of the puck is detected for more than 5 frames, the coordinates of the center move at a certain speed, and the puck moves in the positive x-direction on the image.

```
if self.previous_position is not (0,0) and x_move >= 3 and move_distance >= 4.5:
    if self.x > 120:
        self.puck_move = True
        self.slope_avg((self.current_position[1] - self.previous_position[1]) /
(self.current_position[0] - self.previous_position[0]))
else:
    self.puck_move = False
```

The average movement of the puck is calculated over five frames, and the slope is computed. The x and y coordinates in the fifth frame are designated as the point for trajectory creation and the predicted trajectory is drawn. Furthermore, as there can be positive and negative slopes, the algorithm proceeds separately for both cases.

```
# If puck is moving
if self.puck_move == True and len(self.slope_list) > 0 and self.path_drawn ==
False:

    if self.slope != 0 and self.slope < 0:
        predicted_x = self.previous_position[0] + (roi_y1 -
self.previous_position[1]) / self.slope
    elif self.slope != 0 and self.slope > 0:
        predicted_x = self.previous_position[0] + (roi_y2 -
self.previous_position[1]) / self.slope
    else:
        predicted_x = self.previous_position[0]
```

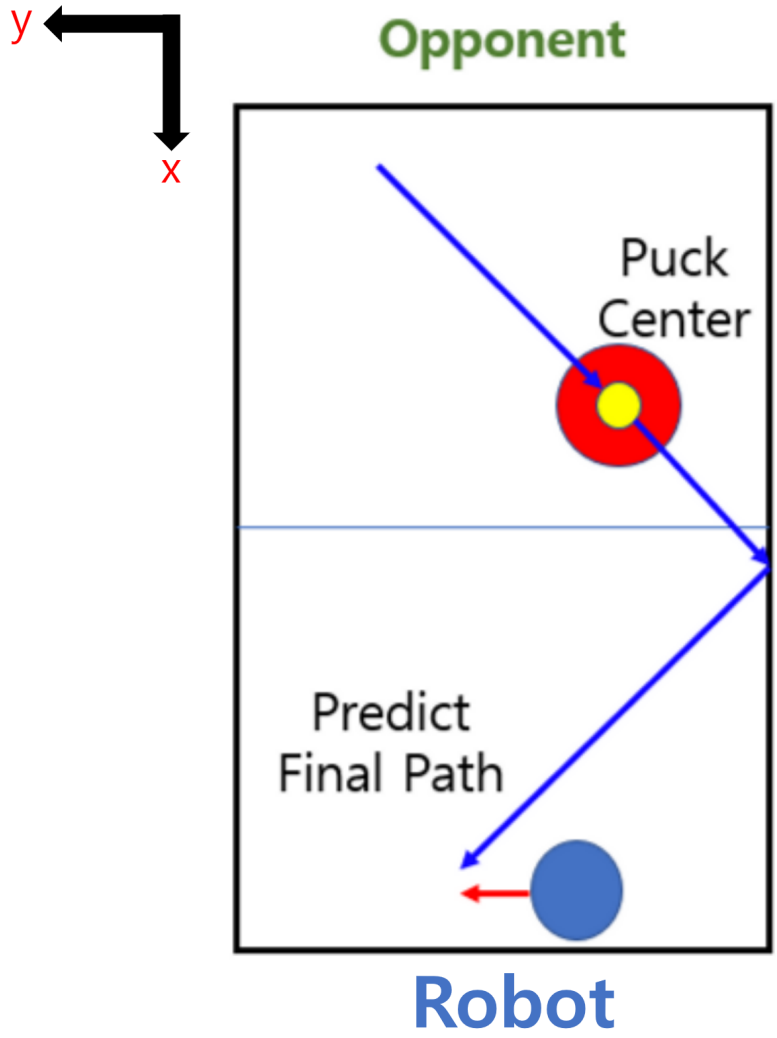


Figure 7. Concept Image

3.4.3 Adjustment of Puck Trajectory Reflection

Since there exists an average slope and a final point, as described in Section 3.4.2, the trajectory can be represented as a straight line akin to a linear equation. However, the puck might not solely travel in a straight path but may bounce off the walls one or more times, and these reflections need to be accounted for.

While assuming that the angle of incidence is equal to the angle of reflection, it is observed that the walls of the hockey table do not have sufficient friction; thus, the angle of reflection is smaller than the angle of incidence.

Through extensive experimentation, it was noted that the reduction in the angles was not uniform. When the angle of incidence was high, both angles decreased by about 0.7 to 0.8 times, but when the angle of incidence was small, they decreased to around 0.4 times (Note that this factor was applied to the slope, not the angle).

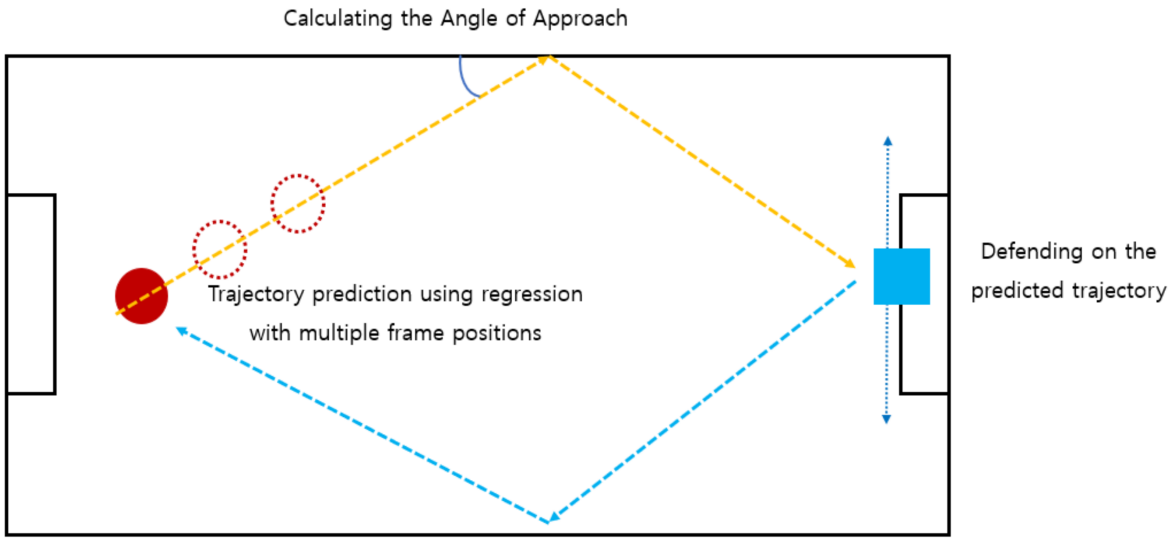


Figure 8. Reflect of Puck

If the predicted trajectory of the puck intersects the wall, this intersection point is used as the new starting point, and the line is redrawn with the reduced slope. The algorithm accounts for two wall bounces. This is because, in most cases, the puck does not reach the region where the robot is positioned if it bounces more than twice.

```
if self.hit_border is not None and predicted_x != roi_x2:
    if self.slope != 0:
        if 0 <= abs(self.slope) <= 1.5:
           self.slope = self.slope * 0.4
        if 1.5 < self.slope <= 5:
           self.slope = self.slope * 0.7
        if -5 < self.slope <= -1.5:
           self.slope = self.slope * 0.8

        self.reflected_y2 = self.hit_border - self.slope  * (roi_x2 -
self.current_position[0])
        self.reflected_x2 = self.current_position[0] + (self.reflected_y2 -
self.hit_border) / -self.slope

        # when hit the borders
        if self.reflected_y2 >= roi_y2 and self.reflected_x2 <= roi_x2:
            self.reflected_y2 = roi_y2
            self.reflected_x2 = self.current_position[0] + (self.reflected_y2 -
self.hit_border) / -self.slope # * 1.2

        if self.reflected_y2 <= roi_y1 and self.reflected_x2 <= roi_x2:
            self.reflected_y2 = roi_y1
            self.reflected_x2 = self.current_position[0] + (self.reflected_y2 -
self.hit_border) / -self.slope

    if self.reflected_x2 is not None and self.reflected_y2 is not None:
        self.reflected_path_line =[[int(predicted_x), int(predicted_y2)],
[int(self.reflected_x2), int(self.reflected_y2)]]
        # print(self.reflected_path_line)

        self.reflected_path_line = list(self.reflected_path_line)
        # print(self.reflected_path_line)

    else:
```

```
        self.reflected_path_line = None

if self.reflected_x2 is not None:
    self.reflected_x2 = round(self.reflected_x2,3)

else:
    self.reflected_path_line = None

if self.reflected_y2 is not None:
    self.hit_border2 = roi_y1 if self.reflected_y2 == roi_y1 else roi_y2

if self.hit_border2 is not None and self.reflected_x2 != roi_x2:
    self.reflected_x3 = roi_x2
    self.reflected_y3_temp = self.hit_border2 + self.slope *  (roi_x2 -
self.reflected_x2)  # reflection

    # self.reflected_y3_temp will be bounded by the ROI borders
    self.reflected_y3 = max(roi_y1, min(roi_y2, self.reflected_y3_temp))

    self.extra_reflected_path_line = [[int(self.reflected_x2),
int(self.reflected_y2)], [int(self.reflected_x3), int(self.reflected_y3)]]
    print(self.path_line)
    print(self.reflected_path_line)
    print(self.extra_reflected_path_line)

else:
    self.extra_reflected_path_line = None
```

3.4.4. Robot Flag

When the end of the predicted trajectory reaches the area depicted in the image below, a corresponding flag is sent to the robot. The robot then moves to the position dictated by the flag, as described in Section 3.5, and the game continues.

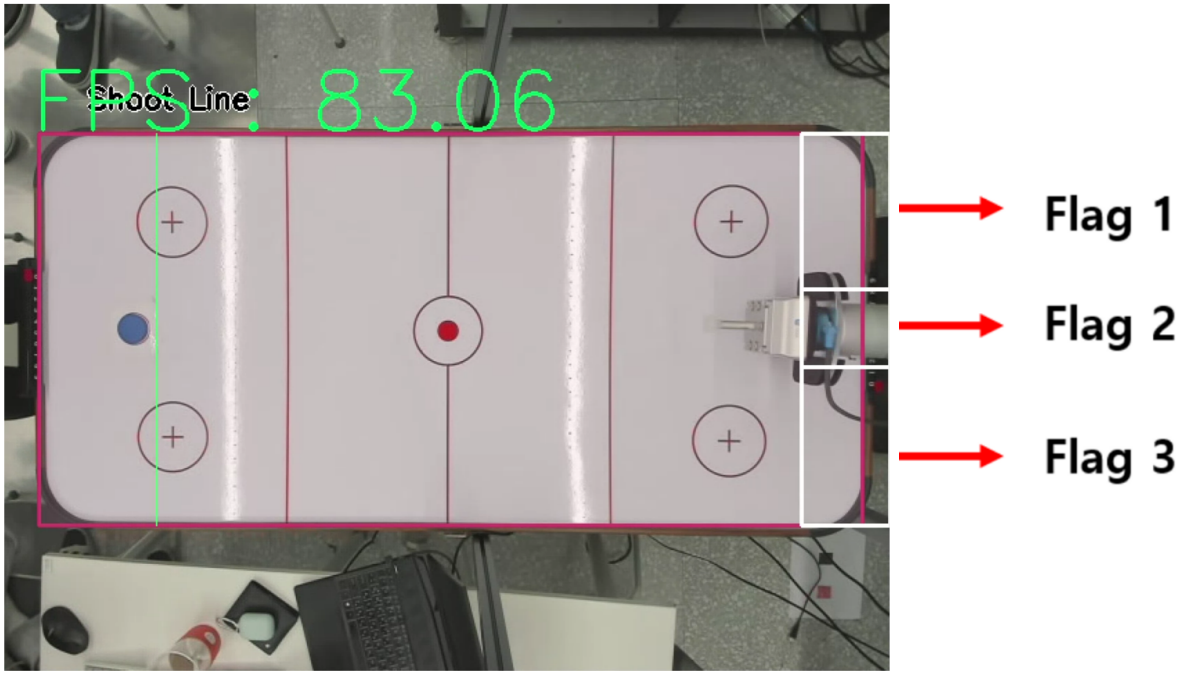


Figure 9. Area of Fleg

3.4.5. Trajectory Removal

Once the trajectory is drawn, the algorithm resets any previously drawn trajectory. The criteria for resetting is when the object moves a specific pixel distance in the negative x-direction with respect to the image coordinates.

```
# Reset slope list and drawn paths if object moves left more than -3
if x_move < -3:
    self.slope_list = []
    self.path_line = None
    self.reflected_path_line = None
    self.path_drawn = False
    self.extra_reflected_path_line = None
```

3.4.6. Goal Flag Activation

If the puck passes the white box shown in Figure 9 and the contour of the puck is not detected for 200 frames, the goal flag is activated and a “--You Win--” message is displayed on the screen. Upon a goal, the robot moves to the position shown in Figure 10, Goal Post Basket, and uses a vacuum gripper to retrieve the puck. After 400 frames, the “--You Win--” message disappears and the game can be restarted from the beginning.

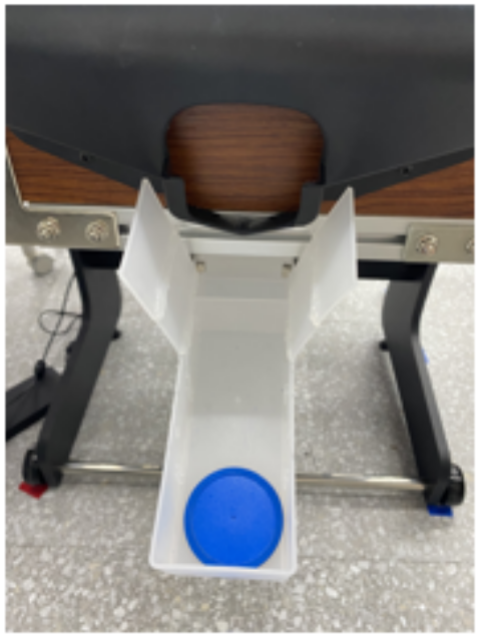


Figure 10. Goal Post Basket

3.5. Robot Manipulation

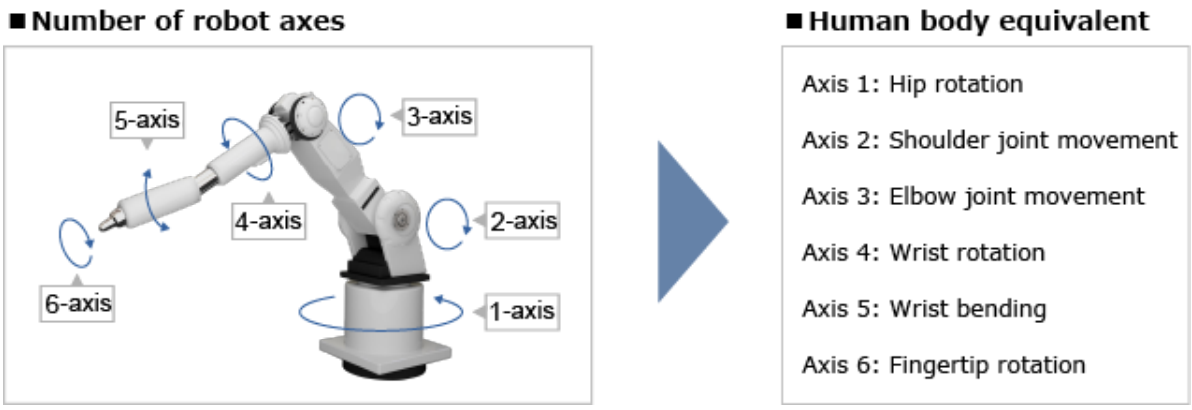


Figure 11. Robot Axis

There are three types of movements for the robot: absolute coordinate commands (x, y, z, roll, pitch, yaw), relative coordinate commands (x, y, z, roll, pitch, yaw), and joint commands (limit radian for each joint) which were used in our experiment.

A problem with using absolute and relative coordinates is that ROS calculates the path when moving along a single path. At this point, instead of taking the optimal path, it tends to slow down by using all joints.

Therefore, in our robot where speed is important, we only utilized the rotation of the z-axis joint on the 1-axis side and used the minimum number of axes. The reason for using the minimum number of axes is because all joint movements do not occur simultaneously, but move in series, so only one axis was used.

Also, for quick movements, we set the stopping part in the go_to_joint_state function to False to move with the minimum delay when the next command comes in.

However, after the goal, which is flag 4, the stop had to be set to True, so different function commands were used.

- path planning

```
self.target_joints_1 = [-10.0*deg,  -14.18*deg, 121.82*deg,   -1.07*deg,
 -16.58*deg,  1.89*deg ]     # Flag2
self.target_joints_2 = [  0.0*deg,  -14.18*deg, 121.82*deg,   -1.07*deg,
 -16.58*deg,  1.89*deg ]     # Flag3
self.target_joints_3 = [ 10.0*deg,  -14.18*deg, 121.82*deg,   -1.07*deg,
 -16.58*deg,  1.89*deg ]       # Flag4


self.target_joint_goal_safe1 = [0.0*deg,-18.60*deg, 66.45*deg, -1.35*deg,
47.18*deg, 0.10*deg]
self.target_joint_goal_safe2 = [0.0*deg,-28.38*deg, 107.98*deg, -1.30*deg,
101.33*deg, 0.10*deg]# @Flag4 --> Move safe loc
self.target_joint_goal_grip = [-0.01*deg, -26.51*deg, 132.48*deg, -0.06*deg,
76.38*deg, 1.89*deg] # @Flag4 --> Grip loc
self.target_joint_end       = [0, 0, 45*deg, 0, 45*deg, 0] # @Flag4 --> Grip loc
```

- robot manipulation

```
if self.flag == 1   :  ## Flag1에 대한 위치로 이동
    self.indy10_interface.go_to_joint_state_False(self.target_joints_1)
```

```
elif self.flag == 2 :  ## Flag2에 대한 위치로 이동
    self.indy10_interface.go_to_joint_state_False(self.target_joints_2)

elif self.flag == 3 :  ## Flag3에 대한 위치로 이동
    self.indy10_interface.go_to_joint_state_False(self.target_joints_3)

elif self.flag == 4:
    self.indy10_interface.go_to_joint_state_True(self.target_joint_goal_safe1)
    self.indy10_interface.go_to_joint_state_True(self.target_joint_goal_safe2)#
테이블에 닿지 않는 안전한 위치로 로봇 팔 이동(For Grip)
    self.indy10_interface.go_to_joint_state_True(self.target_joint_goal_grip) #
Grip할 위치로 이동
    self.indy10_interface.grip_on()
    cv.waitKey(500)
    self.indy10_interface.go_to_joint_state_True(self.target_joint_goal_safe2)
    self.indy10_interface.go_to_joint_state_True(self.target_joint_goal_safe1)
    self.indy10_interface.go_to_joint_state_True(self.target_joints_2)
    self.indy10_interface.grip_off()
    self.flag == 2
```

3.6. Robot Speed Control

Due to Indy 10 being a cooperative robot, the speed of the robot's actuator is slow. Therefore, speed configuration is required. As the speed function within ROS does not allow for changes in speed, we referred to the corresponding code in the documentation of the robot manufacturer, Neuromeka, on their website.

[neuromeka docs]

The robot's speed can be set from 1 to 9. We set it to the fastest speed, and after setting it to 9, we were able to proceed with a simple game.

indy_set_velocity.py

```
#!/usr/bin/env python3
#-*- coding:utf-8 -*-

import indydcp_client as client
import copy


def main():
    robot_ip = "192.168.0.8"    # 예시 STEP IP 주소
    robot_name = "NRMK-Indy10"   # IndyRP2의 경우 "NRMK-IndyRP2"
    indy = client.IndyDCPClient(robot_ip, robot_name) # indy 객체 생성

    indy.connect() # 로봇 연결
    vel  : int   = 9 # 1 ~ 9

    print(f'setting velocity: {vel:.2f}')
    # Setting
    indy.set_joint_vel_level(vel)     # 1 ~ 9
    indy.set_task_vel_level(vel)     # 1 ~ 9

    indy.disconnect() # 연결 해제
```

```
if __name__ == '__main__':
    try:
        main()

    except Exception as e:
        print("[ERROR]", e)
```

4. Result (Demo video)

Demo Video : Click here

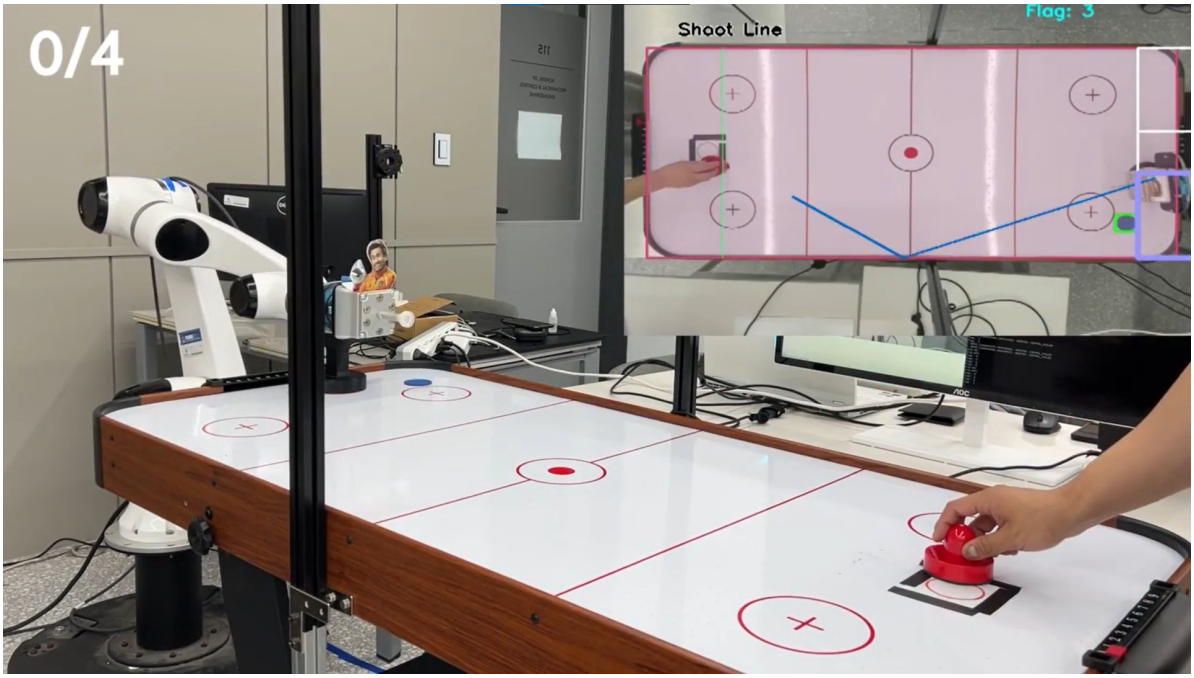


Figure 12. Example of operation

Table 1. Attempt Result

Attempt	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Success	O	O	O	O	O	O	O	O	O	X	O	O	O	O

As can be seen in the video above, it showed a successful defense result in 13 out of 14 attempts. The Flag 4 command, an action to pick up the Puck, was also confirmed to be operating normally.

Accuracy : 92.85%

5. Multi Process Shell Script Management

This project involves the inconvenience of running multiple files simultaneously. Even if it is a single program, difficulties arise in the progression of the program if the order is mixed up or if certain files are not executed. Therefore, the aim is to resolve this inconvenience by executing only one shell script file.

- Things to do before running the file:
    - Place the sh file in the **catkin_ws** folder.

    - Type "chmod +x airhockey.sh" in the terminal.
- Shell script execute

```
# Run the Python script

source devel/setup.bash
python3 /home/"user name"/catkin_ws/src/indy_driver/src/indy_set_velocity.py

# Define a function to run a command in a new Terminator window and arrange it
function run_in_terminator {
    local command=$1
    # Use Terminator's --geometry option to specify the window position and size
(format: WIDTHxHEIGHT+X+Y)
    # Adjust values as needed
    terminator --geometry=800x600+0+0 -e "$command" &
    sleep 5
}

# Run the commands
run_in_terminator "bash -c 'source devel/setup.bash; roslaunch
indy10_moveit_config moveit_planning_execution.launch robot_ip:=192.168.0.8; exec
bash'"
run_in_terminator "bash -c 'source devel/setup.bash; rosrun indy_driver
camera.py; exec bash'"
run_in_terminator "bash -c 'source devel/setup.bash; rosrun indy_driver
move_robot.py; exec bash'"

# Wait for user input to close all windows
read -p "Press Enter to close all Terminator windows"

# Close all Terminator windows
killall terminator
```

And execute the following in the terminal window.

```
cd catkin_ws
./airhockey.sh
```

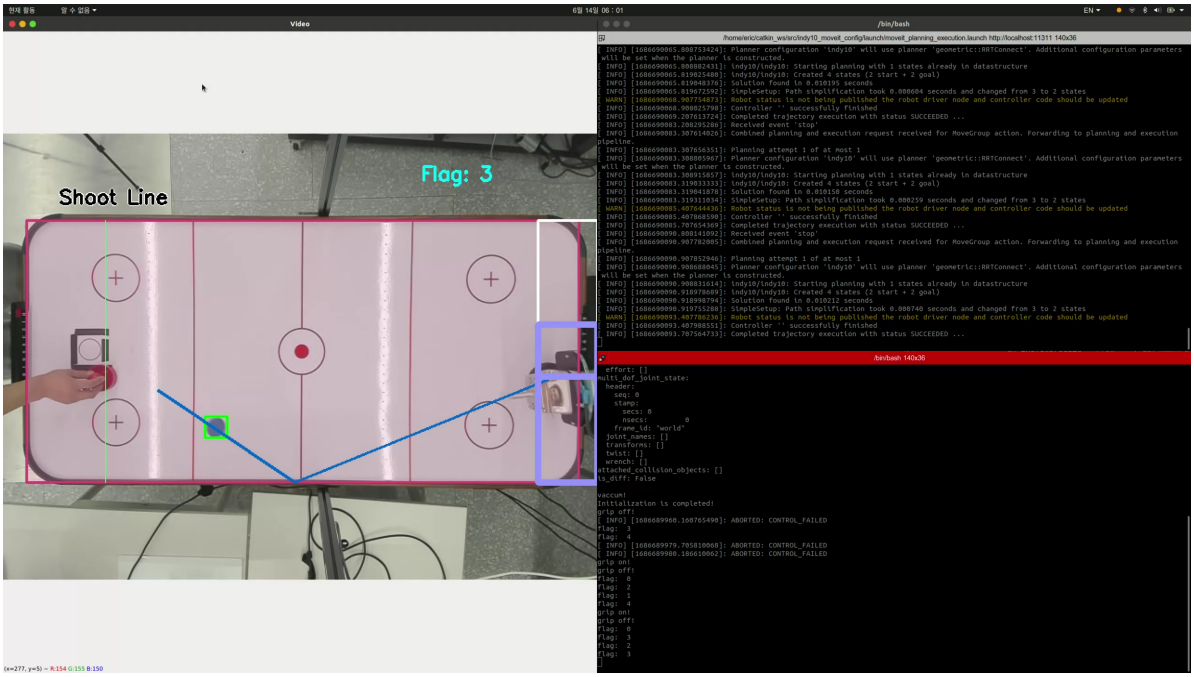


Figure 13. Program Excute

A total of four windows and a camera window will be opened. The provided image shows the two important terminals and the displayed camera.

6. Discussion and Analysis

1. **Issues with Robots and ROS**

There are several issues and limitations in the progression of this project. One of the most significant problems and limitations is that despite the robot's speed being set at its maximum, it showed inadequacy in blocking the Puck. As a result, the initial plan of continuing the rally had to be altered to a game of sudden death, and instead of real-time path prediction, it was necessary to derive the predicted path through the average slope within 5 frames after hitting the Puck. Consequently, discrepancies between the anticipated and actual path of the Puck occurred. In order to resolve this, it is deemed appropriate to use a robot capable of faster action.

Furthermore, when controlling the robot using ROS, an issue arose where the robot's movements slowed down as time elapsed, causing delayed reactions to input commands. Testing showed that the robot takes an additional second to process each multiple of ten commands. This suggests a delay in processing speed over time in ROS, as the same issue occurs in other robots of the same model.

2. **Issues with Camera Perception and Path Prediction**

The predicted path was linearly assumed by setting the angle of incidence and reflection with the slope according to the change in the position of the Puck. However, it was confirmed that the actual movement of the Puck was nonlinear due to the wind coming from the floor of the game court and friction. Potential solutions to the Puck's nonlinear movement include using a faster-acting robot to accommodate real-time path prediction, or using methods such as multiple regression or the Kalman Filter to enable nonlinear path prediction.

The original plan was to use a deep learning model to segment the Puck and predict its path, but despite using a camera supporting 60 FPS, the segmentation was interrupted due to afterimages when the object moved. Therefore, instead of using a deep learning model, the segmentation of the blue Puck was carried out by setting the HSV channel value area. This resulted in a

susceptibility to changes in illumination. There were no significant issues when playing the game under constant lighting, but problems occurred when shadows were formed or the lighting changed. It is expected that by using a higher FPS camera based on a deep learning model, issues of segmentation interruption can be prevented.

7. Tutorial

7.1. Download the Source File from GitHub

1. Download the file from the provided GitHub repository git-hub link.
2. After decompressing the downloaded file, move the **airhockey.sh** file located in the **airhoceky** folder to **catkin_ws**. Move folders with the prefix indy to **/catkin_ws/src**.
3. Proceed with the catkin_make command.
    1. If the cmake build is not successful, read the error message carefully and resolve the issue.
    2. The most common issue we encountered was the need to install packages that do not automatically get installed with ROS. Below are the commands to install the necessary packages for resolving build errors encountered during setup in a new environment.

```
sudo apt install ros-noetic-industrial-robot-client
sudo apt install ros-noetic-moveit-visual-tools
sudo apt install ros-noetic-moveit-commander
```

7.2. Preset

- Hardware Settings

1. **Table Settings:**
    1. Use a tablet that can configure indy10 to manually move the Indy-10 robot to the reference position through direct teaching (e.g., [0, 0, tau/4, 0, tau/4, 0]).
    2. Align the center of the robot with the center of the table.
    3. Use a digital protractor to adjust the table's tilt close to 0 degrees. Proper alignment improves puck trajectory prediction accuracy.
    4. If the table is tilted, place an object under the table legs to level it.
    5. Connect the power to the air hockey table and turn the switch on.

2. **Robot Settings:**

    Robot settings need to be adjusted if the position is misaligned. Use the tablet connected to the robot to check the absolute coordinates.

    1. Specify the right, center, and left positions relative to the robot. If the paddle is too close to the table, the camera may shake along with the table during robot arm movement, so finding the proper distance is crucial.
    2. When choosing the robot's position after a goal is scored, set it to a safe location so that the robot doesn't collide with the table during movement.
    3. Apply the absolute coordinates obtained in the above process to the following positions in **move_robot.py**.

```
self.target_joints_1 = [-10.0*deg,  -14.18*deg, 121.82*deg,   -1.07*deg,
 -16.58*deg,  1.89*deg ] # Flag2
self.target_joints_2 = [  0.0*deg,  -14.18*deg, 121.82*deg,   -1.07*deg,
 -16.58*deg,  1.89*deg ] # Flag3
self.target_joints_3 = [ 10.0*deg,  -14.18*deg, 121.82*deg,   -1.07*deg,
 -16.58*deg,  1.89*deg ] # Flag4


self.target_joint_goal_safe1 = [0.0*deg,-18.60*deg, 66.45*deg,
-1.35*deg, 47.18*deg, 0.10*deg]
self.target_joint_goal_safe2 = [0.0*deg,-28.38*deg, 107.98*deg,
-1.30*deg, 101.33*deg, 0.10*deg]# @Flag4 --> Move safe loc
self.target_joint_goal_grip = [-0.01*deg, -26.51*deg, 132.48*deg,
-0.06*deg, 76.38*deg, 1.89*deg] # @Flag4 --> Grip loc
self.target_joint_end       = [0, 0, 45*deg, 0, 45*deg, 0] # @Flag4 -->
Grip loc
```

4. Modify the path in **move_robot.py** at the location below.

```
sys.path.inser(0, 'indy_driver/src 전체 경로로 설정')
```

3. **Camera Settings:**
    1. Install the camera on the profile fixed to the table. If the camera is not fixed, accurate object recognition cannot be achieved.
    2. In **camera.py**, apply the camera index number between 0 and 2 in the corresponding section to select the camera to use. If a flickering camera image appears, it means that it is an IR camera, not the main camera, and you should change to a different index.

```
def __init__(self, cameraNumber = 0):
```

    3. When running the code, adjust the camera angle so that the lines representing the perimeter of the playing field match the actual game area.

7.3. Running the Program

1. Open the terminal and navigate to the catkin_ws directory.

```
cd catkin_ws
```

2. Modifying the sh file:

If it's your first time downloading the sh file, you need to modify the "user name" in the code within the file to match your username.

airhockey.sh

```
python3 /home/"user name"/catkin_ws/src/indy_driver/src/indy_set_velocity.py
```

3. Execute the sh file by entering the following in the terminal.

```
./airhockey.sh
```

If the sh file is not recognized and does not execute, enter the following to make it executable and then run it.

```
chmod +x airhockey.sh
./airhockey.sh
```

7.4. Playing the Game

1. When the camera window appears, left-click and drag the blue puck area visible in the camera. Slide the puck to the right of the green line to check if it's being recognized properly. If a continuous green box appears around the puck, it is being recognized correctly.

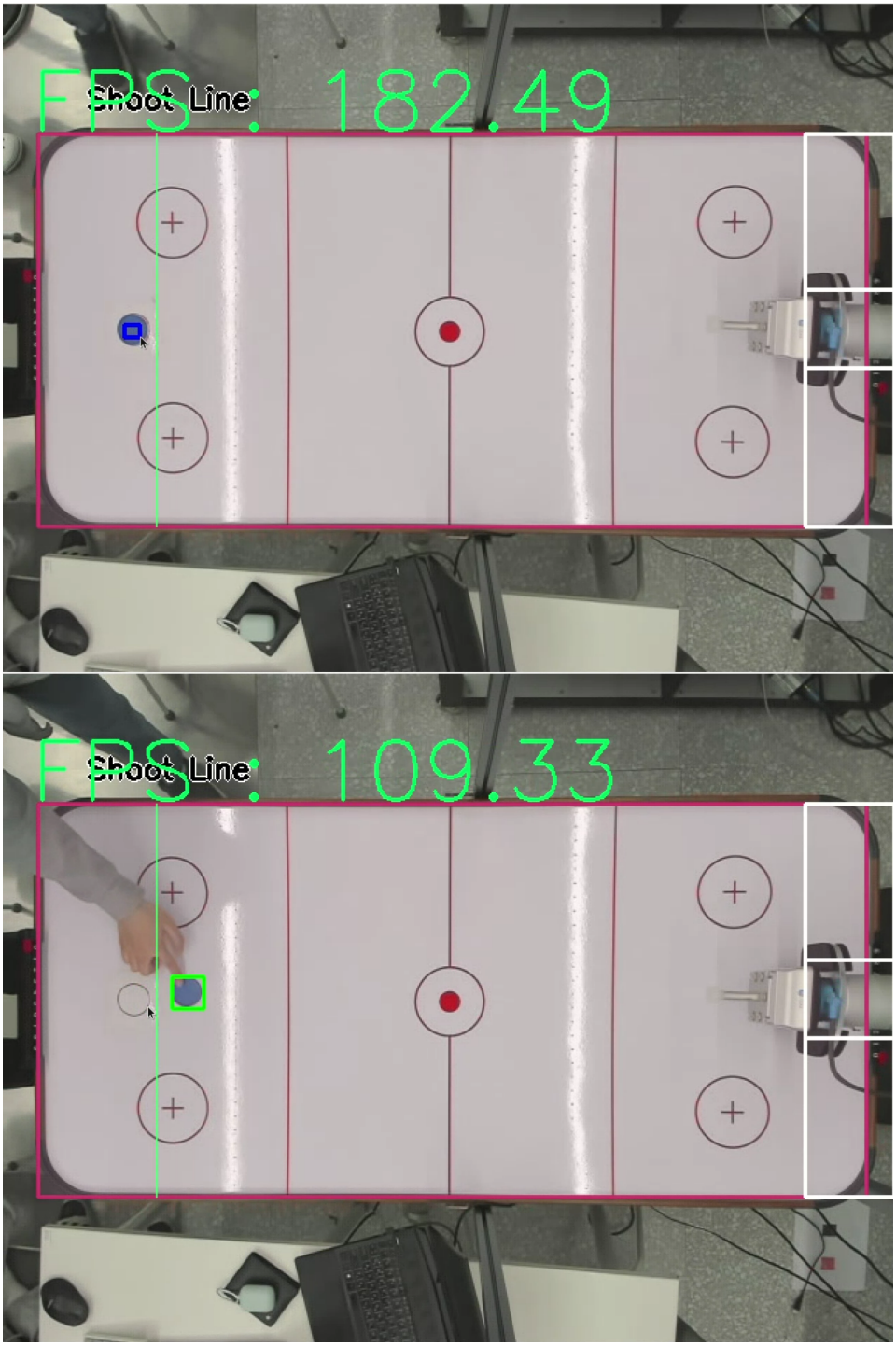


Figure 14. Game Setting 1

2. Place the puck in the circle in the center and enjoy the game by hitting it.

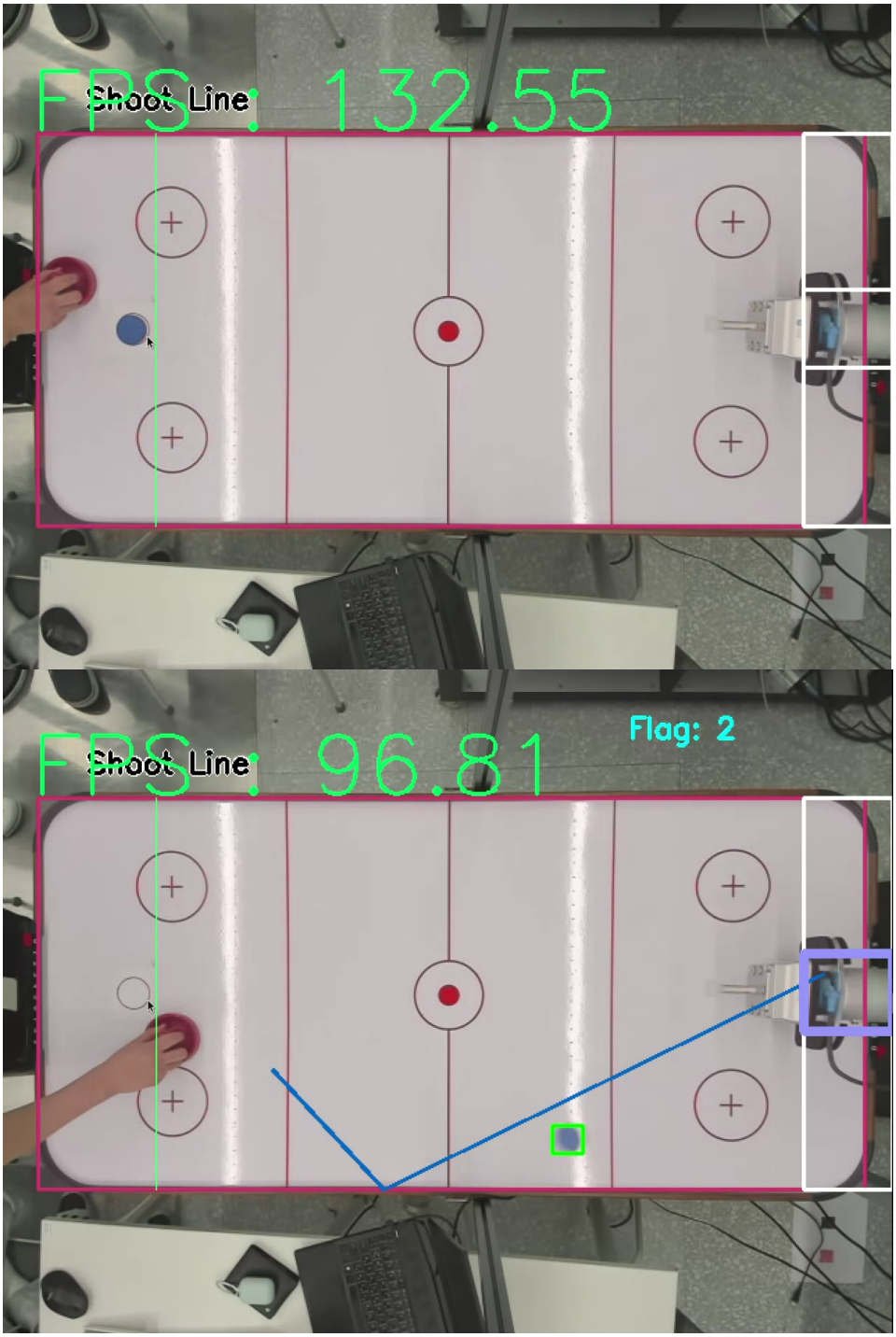


Figure 15. Game Setting 2