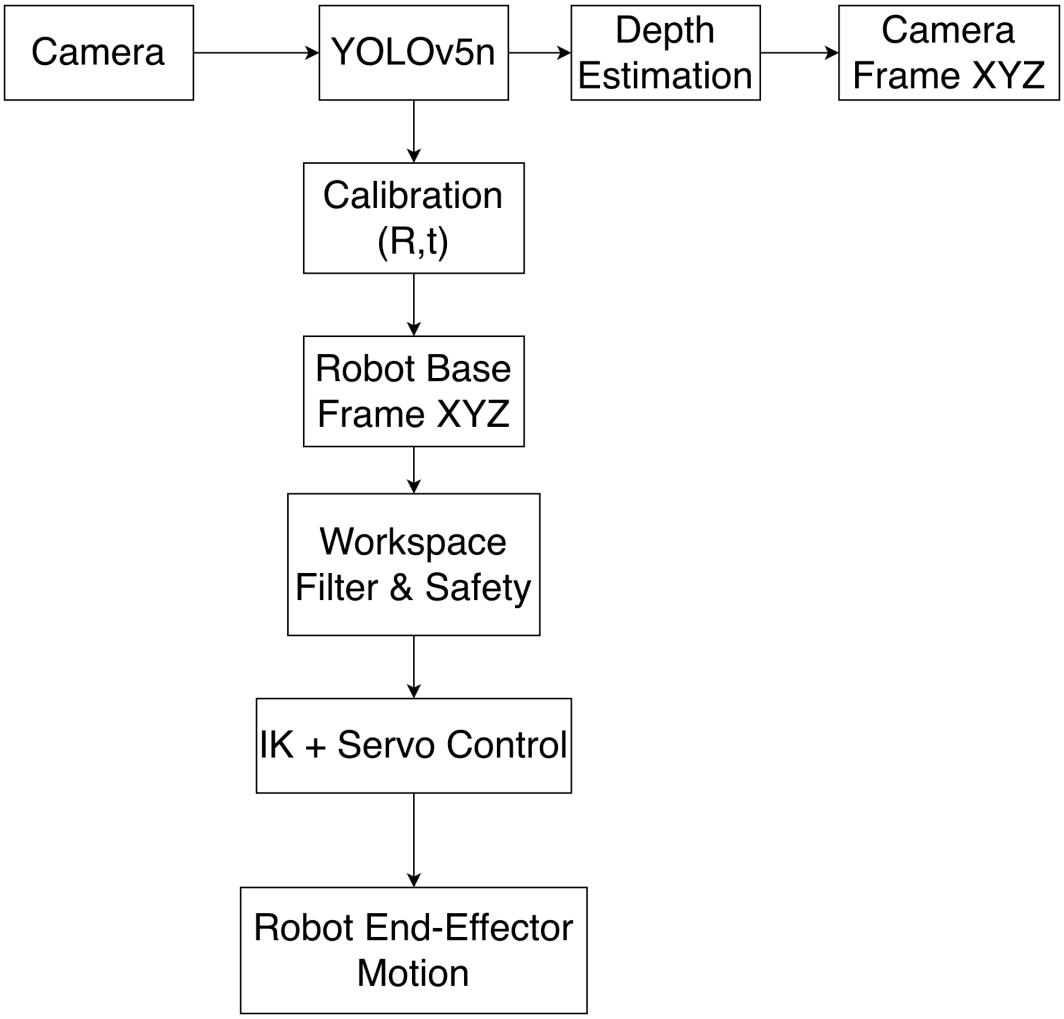
Juan Ali Ruiz Guzman

EE473
03 December 2025
Project 13

Project 13 Visual Perception & Servoing

Project Diagram:



Part 1 — Real-Time Object Detection

The goal of this section was to implement real-time object detection using YOLOv5n (nano) from the DepthAI model zoo on the OAK-D camera to detect common objects. After configuring the model, the OAK-D handled inference using the Myriad X VPU, which allowed the Raspberry Pi to maintain strong performance without heavy CPU load.

On my Raspberry Pi, the object detection ran between 15–20 FPS depending on how busy the frame was. For depth estimation, I enabled the stereo camera pipeline. Instead of taking a single pixel value, I sampled depth around the center of the detected bounding box because a single pixel tends to be unstable. This resulted in far more reliable distance measurements. If the depth was invalid or too noisy, the system ignored the measurement.

I displayed bounding boxes, confidence scores, and the FPS counter on the live image. This allowed the system to look at everyday objects in real time and identify them (e.g., “cup”, “apple”, “phone”) along with a confidence score.

Demo Video:

https://drive.google.com/file/d/1384nqpVkot29LfZluaPXAYn1nEKo9O9r/view?usp=drive_link

---

Part 2 — AprilTag Fixed-Camera Calibration + Auto-Check

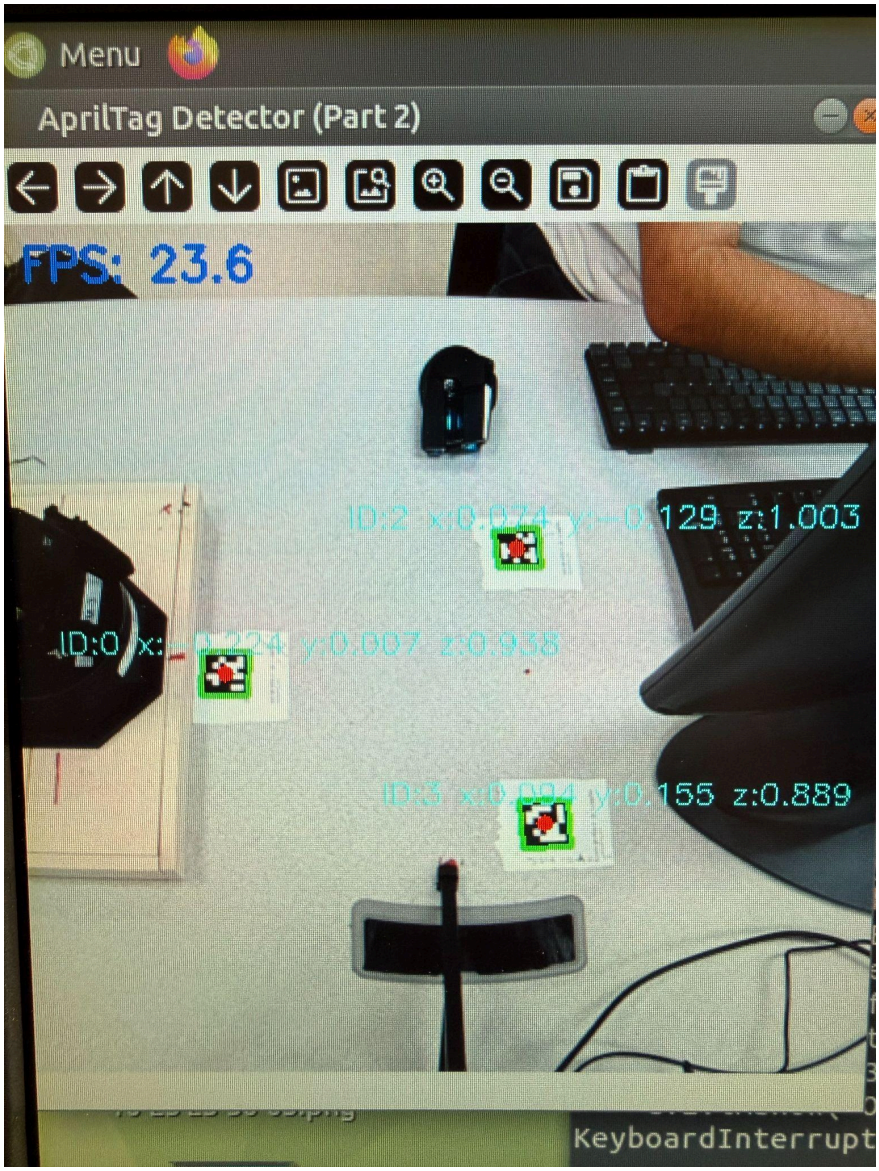
Once object detection and depth estimation were working, the next step was to make the robot understand those measurements. The robot did not inherently know anything about the camera’s coordinate system. Detection results in the camera frame meant nothing until the system knew how the camera was positioned relative to the robot base.

To solve this, I printed three AprilTags and placed them in fixed, measured locations. The main tag was mounted near the RX-150 base, while the other two were placed further away and separated from each other. Their real-world positions (in meters) were recorded in apriltag_known_positions.yaml. The OAK-D detected these tags and estimated their position and orientation in the camera frame.

I collected more than 30 detections per tag and matched those detected camera positions to their corresponding known robot frame positions. Using these pairs, the system solved for a single rigid transformation:

Probot=R⋅Pcamera+tP_{robot} = R \cdot P_{camera} + tProbot=R⋅Pcamera+t

This equation produced a rotation matrix and translation vector that aligned the two coordinate systems. Since the calibration error was small, I saved the result in camera_robot_calibration.yaml. This calibration file was later used by the robot in Part 3.



Demo Video:

https://drive.google.com/file/d/1uaZmm0TPkAjlviKAuaokrvr8LVevyAMQ/view?usp=drive_link

---

Part 3 — Visual Servoing: Object Tracking with Robot Arm

In Part 3, all components were combined. YOLO detected the chosen object (in my demo, a “cell phone”). The script extracted the bounding box center pixel (u,v)(u, v)(u,v) and sampled the depth Z at that location. Once I had valid pixel coordinates and depth, they were converted to 3D coordinates in the camera frame using the standard pinhole projection model:

$$X = (u - cx) * Z / fx$$
$$Y = (v - cy) * Z / fy$$
$$Z = Z$$

This camera-frame 3D position was then transformed into robot coordinates using the calibration:

$$P_{robot} = R @ P_{camera} + t$$

This produced a meaningful position in the robot’s reference frame, which is essential for arm control.

Because the camera readings fluctuated slightly from frame to frame, I applied exponential smoothing to reduce jitter. This prevented the arm from shaking or overshooting. I also clamped the workspace to ensure the robot was never allowed to move behind itself, dive into the table, or stretch too far forward. These limits prevented unsafe motions and kept the arm operating in a safe region in front of the base.

Lastly, I commanded the robot using Interbotix’s built-in inverse kinematics function `set_ee_pose_components(x, y, z, ...)`. This allowed the arm to position its end effector without requiring me to manually compute joint angles. The robot pointed toward the tracked object smoothly. If the phone moved left, the robot tracked left. If the phone moved closer or farther, the robot adjusted accordingly.

Demo Video:

https://drive.google.com/file/d/1jdncxryCSaJx09EgP4J2y4H-35W3F-z5/view?usp=drive_link

---

Results

The system performed well once the workspace limits were properly tuned. The RX-150 pointed at the tracked object smoothly and did not rotate backward or collide with the camera. Depth readings remained stable, and the camera maintained real-time detection. Lateral tracking (left and right) was very responsive, while forward reach was more modest but clearly visible. Even when the target moved quickly, the smoothing algorithm prevented jitter. Safety boundaries and rate-limited commands helped keep the behavior predictable and controllable.

Conclusion

Across all three parts, I built a complete real-world robotic vision pipeline: detect, measure, convert to 3D, and move. Instead of executing pre-programmed joint angles or trajectories, the RX-150 reacted to what it saw in real time. This project demonstrated how computer vision and depth sensing can be integrated with robotic kinematics and control. The robot was not simply executing a script—it was responding to live perception data.