Smart Products Final Project Report

Yuan Gao and Tyler Morrison

April 29, 2019

All of our code can be found in a remote Git repository at `https://github.com/tymo77/smart-products`. A video of the robot picking can be watched here: `https://www.youtube.com/watch?v=LKYOw6bc5pM`. This method was very reliable during testing. I'd estimate our reliability was greater than 90% however, Tyler made a simple mistake during the demo that caused it to be slightly off. A video of the working elevator code can be seen here: `https://www.youtube.com/watch?v=miOiRZVIvwA`. These two sets of code are not dependent. They can be compiled and run separately so, in order to simplify development, they are two different executables, written and compiled in the `project` and `project_motors` directories.

Section 1. Communication logic

To realize a stable and reliable communication between Pi and camera, we utilized the logic shown below

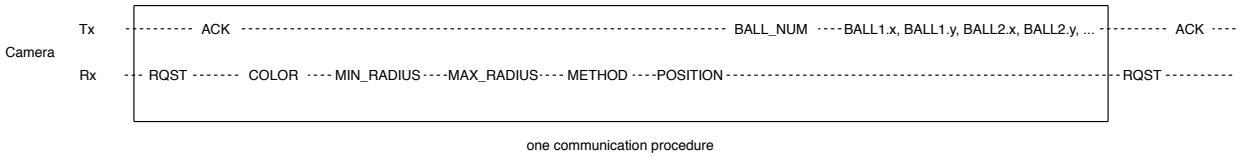

Figure 1: Example of our communication protocol.

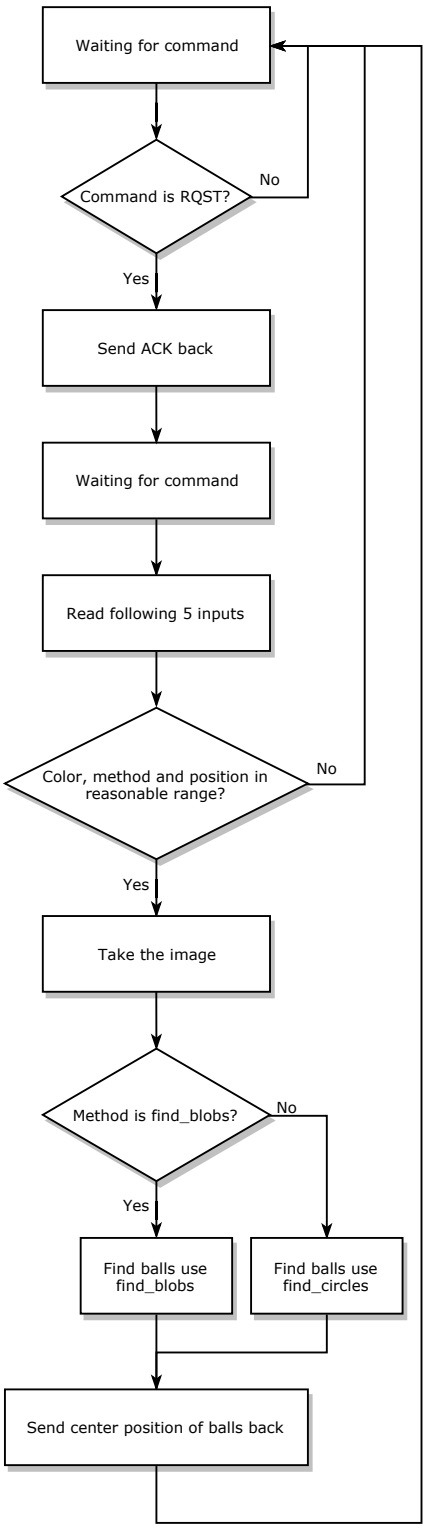

Figure 2: Camera software.

Here we define two handshake codes: request and acknowledgment, two color codes for red and green, two method codes respectively represent `find_circles` and `find_blobs` function and three position codes that stand for three positions of the dobot. Minimum and maximum radius are used to limit the size of circle detected by `find_circles` function and increase the precision. These nine defined codes range from 1 to 9 and are different with each other to avoid misreading. After receiving and verifying the validness of all input parameters, the camera will detect all the balls in desired color at region of interest determined by Dobot position and return the number of balls and then the list of balls' center position.

Section 2. Color selection

We have two methods to detect the ball and the ways to determine the color are different. For `find_circles` function: Because we can't set the preferred RGB value range in the function, after getting the list of circles, we derive the average RGB value in the detected ball area and determine whether a ball is red or green based on the R and G value. For `find_blobs` function: In `find_blobs` function, LAB value can be used to find blobs in certain color. We obtained the LAB value tuple for red and green by capturing the ball region in the image. Find_blob function give us rectangle regions in desired color and the center of the blob is the center of the ball.

Section 3. Overall Algorithm

First we input coordinates for the ramp drop off locations and the home location to scan the scene from. Then we scan the scene, pick a target and move toward it. We stop above it, scan and move to a set height above it. We repeat this process several times to hone in then move in to pick it. We then lift it and carry it to the appropriate ramp and release. After all of the desired colors have been collected and placed, we lift the ramp gates to load the elevator and then lift the elevators to the appropriate height and release the appropriate ramp each time.

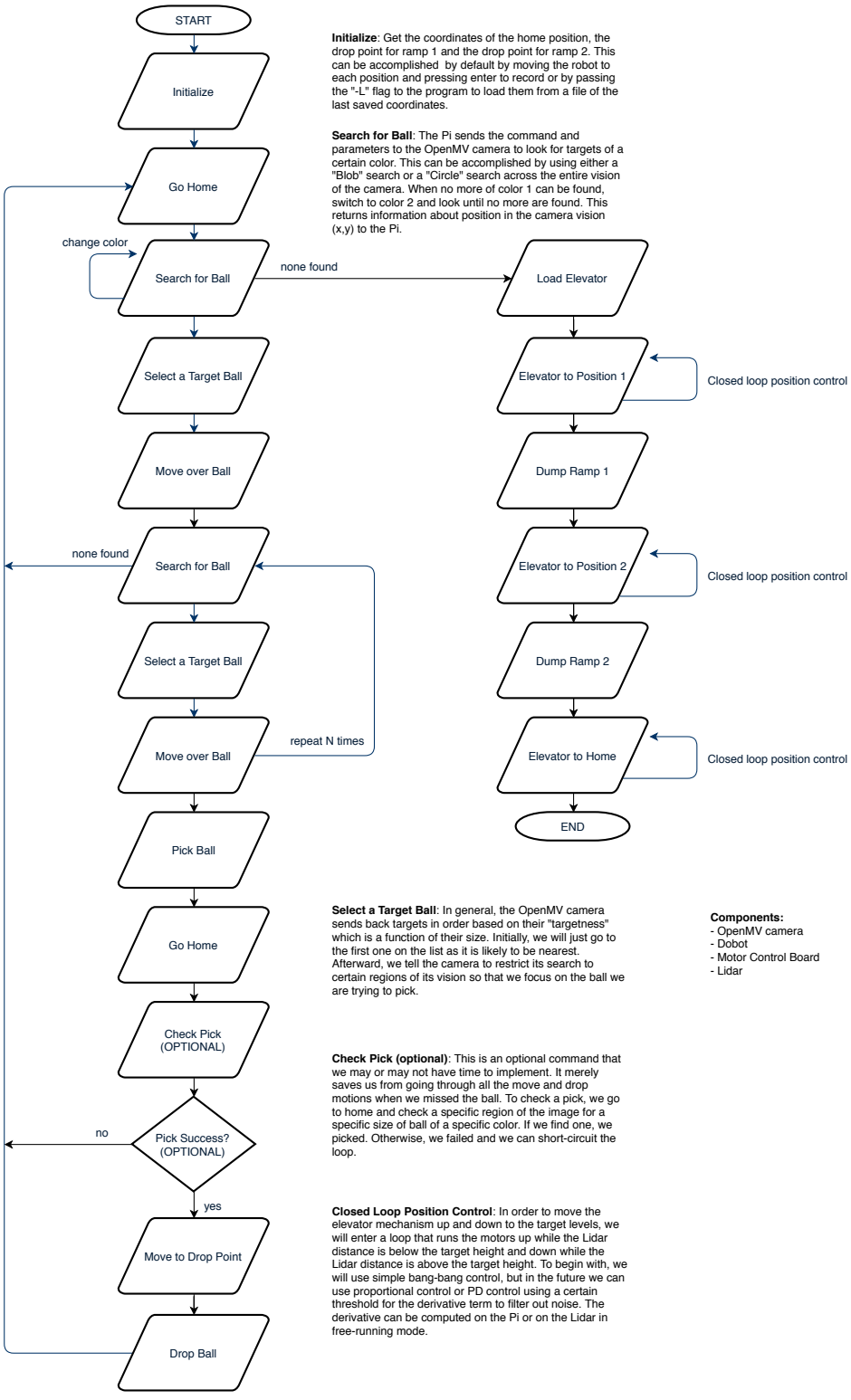

Figure 3: Algorithm.

Section 4. Methods of Note

Here we provide some notes on parts of our approach and workflow that may be unique.

Blobs vs Circles

One of the more tedious parts of the lab is getting the camera to correctly and reliably detect the balls. Some colors work better than others and lighting and background conditions can have significant effects. We ended up toying with the camera settings until we had somewhat reliable detection. We had to fix the white balance and exposure until the image was actually usable. Even then, circle detection failed quite often and it ended up being easier to use blob detection.

Blob detection has some pitfalls but its behavior was overall more consistent. This was what we used in our demo and demo video but we have included all the code necessary to switch between the two at will. The camera is able to use whichever mode the master device requests.

Calibration between camera and robot

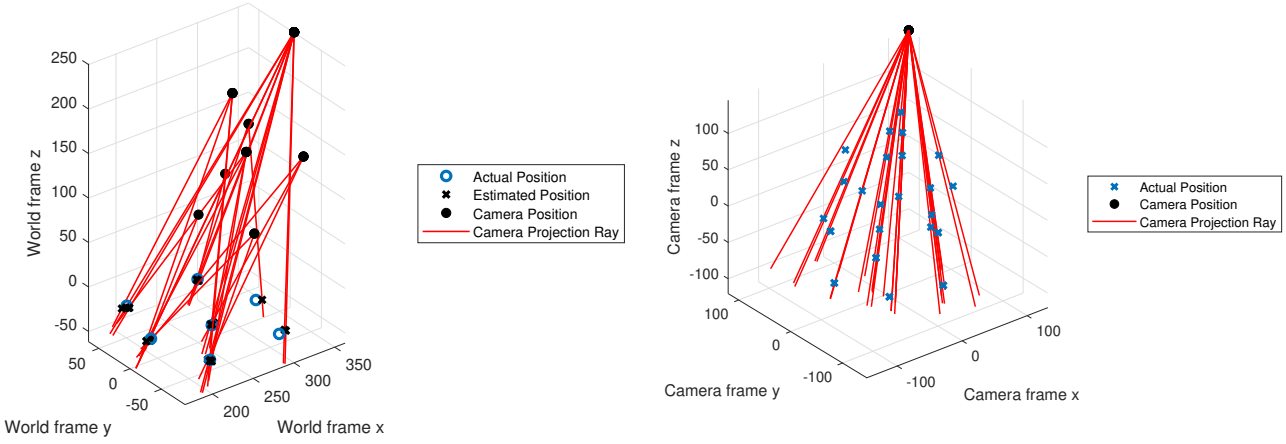

Figure 4: Graphics of the calibration process to translate from xy coordinates in the image to xyz coordinates in the frame of the last link.

In order to be able to get the position of a ball in the world xyz frame from its place in the camera image we make use of the following ideas: 1) Each pixel on the camera corresponds to a ray of possible locations in 3D space in the camera xyz frame (that was set somewhere in space relative to the camera during calibration). 2) The balls all sit on a plane of (basically) constant z in the world xyz frame. 3) There is a constant 3D rigid body transformation from the camera frame to the frame of the third link of the robot because that is where the camera is fixed.

Points 1) and 2) are implemented by solving equations of line intersection with a plane and getting the equation of the line. Dylan has described methods for this before, but we had already developed our own code for this.

Point 3) is implemented by calibration. Several balls are placed in the workspace with positions recorded by placing the end effector gently on their center top. Then, several images are taken of the scene from different positions and the position of the robot for that image as well as the image positions of the centers of the balls are recorded for each detected ball and paired with their known coordinates in the world frame. The image centers of

the positions of the balls are used to create a set of lines in the camera frame. The known positions of the balls and the robot coordinates for each image are used to construct a set of points in link coordinates that correspond to each line in the camera frame. The correct rigid body transformation between these two is found by minimization of the distance between each point in camera space (after being transformed by the rigid body transformation we are solving for) and its corresponding projection line. The result is an accurate 3D rigid body transformation to convert camera image coordinates to camera xyz coordinates to link xyz coordinates to world frame xyz coordinates – all without needing to measure the position of the camera relative to the third link.

Using Make and command line arguments to speed testing and integration

Throughout the lab, in order to decrease compile times during each iteration of testing, we used a handcrafted Makefile to do efficient compilation. Make will only compile files affected by recent changes. Compiling each library and linking them afterward allows minor changes to be made without waiting for the entire codebase to recompile.

We also made use of command line arguments. By passing different flags to our main routine in `project`, various simple tasks can be accomplished without commenting out lines in the main file and recompiling. We include options such as:

`-L`	Load previous saved home and ramp positions.
`--get-pos`	Print out current robot pose coordinates.
`--go-to`	Move robot to xyz coordinates.
`--suction-off`	Turn off suction.
`--suction-on`	Turn on suction.

Section 5. Reflection

We spent the majority of our time on this project working on the pick and color sort – especially converting from the image pixels to 3D world coordinates. We tried one method which was based on the size of the ball. The location on the screen gave a vector that described a ray on which the center of the ball lay and the size of the ball gave how far along the ray it was. This does not take advantage of the fact that all the balls sit in a defined plane. This method was ultimately abandoned because it was not accurate enough. It is difficult to reliably estimate distance by radius as the visible disk of a sphere changes nonlinearly with distance from its surface. We developed an iterative method to solve this problem but even then, the measurement of the size of a ball has substantial variance and even ± 1 pixel of variance is a big change in distance. Finally we settled on using a calibration point-plane-line method as described above.

In development, we separated the project into two parts: the elevator and motor control and the robot control and vision. The robot control and vision was split into the computer vision and communication and the code on the Pi for calculating positions and controlling the robot.

Our approach in controlling the robot was to make a method that was entirely agnostic to the position of the robot world frame. As its homing process is poorly conceived, it requires a tedious reset to get a consistent world frame relative to the base of the robot. Furthermore, the robots' environment is not totally fixed. Some attempts were obviously made to keep the ramps, ball lattice, and robot base in the same position, but these errors accumulate and – in our opinion – it was better to develop a method that was agnostic of these, with no hard-coded positions. That is why we elected to "teach" the home and drop positions to the robot and added functionality to save them, why we don't home the robot, and why we only hard-code the calibration between the last link and the camera and the camera calibration matrix.

Yuan lead work on the computer vision, and elevator/motor control. Tyler lead work on the code for controlling the robot and calculating positions – including the calibration aspect. With that said, we each had a hand in nearly every part of the project.

We feel that we were able to demonstrate the basic functionality of each part of the project. Given more time, our project could even get significantly better and more reliable.