Rovio Lego Collector

Joe McCourt and Jun Nishiguchi

Abstract— **Our objective is to have the Rovio robot find and collect Legos on the floor into one pile. Machine vision techniques are used to find the relative Lego positions. The Rovio robot moves towards and centers in on the closest detected Lego. When the Lego is no longer in view it rotates around until it is facing the origin of the Rovio ceiling localization and and drives to the origin. It then back ups and scans around for more Legos to collect.**

I. INTRODUCTION

Legos are awesome, but they can sometimes be tedious to clean up. A Rovio robot could be able to collect Legos left on the floor and organize them by size and shape, greatly increasing Lego fun. Our Rovio performs tasks that incorporate multiple topics we have covered in this class including control, localization, and machine vision.

The first task is to find and identify the objects in the world. We worked on using image processing techniques to find object pixel coordinates, then developing a method to calculate the position of the object on the ground.

We are able to detect red, yellow, light blue, orange, black and white colored bricks. An SVN repository of all of our code is located at http://code.google.com/p/roviolego/ .

II. THE ROVIO ROBOT

The robot we are using is the WoWee Rovio robot. Lego bricks are our objects of interest. We have attached a simple plow to the Rovio so it can push Lego bricks. Localization data is collected using the base station beam which projects on the ceiling. The primary sensor we are using for processing is the Rovio's camera.

The Rovio acts as a web server which we can connect to and control remotely. Using the TA's Interface.py, we are able to receive real time image data from this webcam. We are primarily using ROS to interface with the robot and the controller we are using is written in C++. Initially the image processing was done in Matlab but we decided to port it to OpenCV due to the slow speed of Matlab.

III. MACHINE VISION:

To find the pixel coordinates of the Lego we first take the raw image and an image that is a solid color which matches the color of the Lego we wish to find. Then we subtract and threshold the two images so that only the areas with colors near that of the subtracted color remain. This is repeated for different RGB values of the colors of the Legos we wish to detect. Next, we apply a morphological opening algorithm to filter out noise. Then we use a blob detection algorithm to find the objects in the binary image and mark their average pixel locations.

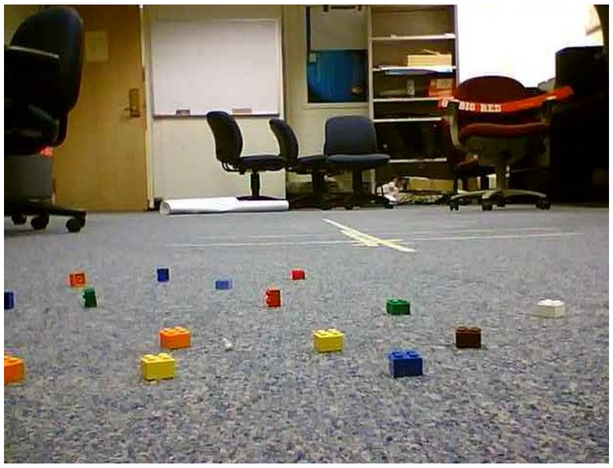

Fig. 1. Original Image

It is possible to use HSV values, however RGB values gave us good results. The algorithm threshold for each color is different due to the fact that each color can potentially introduce more noise to the binary image. For example, when detecting the green and blue Legos, there were situations when "blobs" were detected in the carpet because of the color of some carpet regions. Since there are three channels, the absolute difference in pixel values are then compared to a threshold to see if it qualifies as a pixel match. Once again, the optimal threshold for this differs according to the color. Fig. 2 shows the result. Notice how the black portions of the image are where all three image channels "match" the Lego colors. This threshold can be changed depending on how much noise the next phase can handle.

The next phase includes a combination of morphological filtering techniques such as opening and closing. The opening technique for example can filter out small non-Lego objects. It should be noted here that at this stage it is not fully necessary to get rid of all small objects. This is because some colors such as yellow change dramatically with lower light levels. If the Rovio mostly sees a mostly shadowed yellow Lego, then it will only recognize the brightly lit part, since that is the color it is looking for. Therefore, in some situations it can be worth using closing in order to increase the size of such "potential" Legos. The result of the subtraction and thresholding procedure is given in Fig. 3.

The final step allows us to perform noise filtering as well as Lego detection. The blob detection phase uniquely identifies blobs for the colors desired, additionally it allows for only certain sizes. This is how we can avoid identifying large reddish objects such as cones accidentally. A further

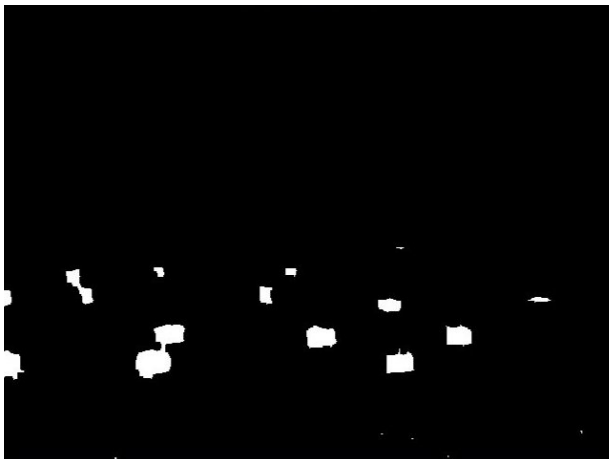
Fig. 2. Binary image

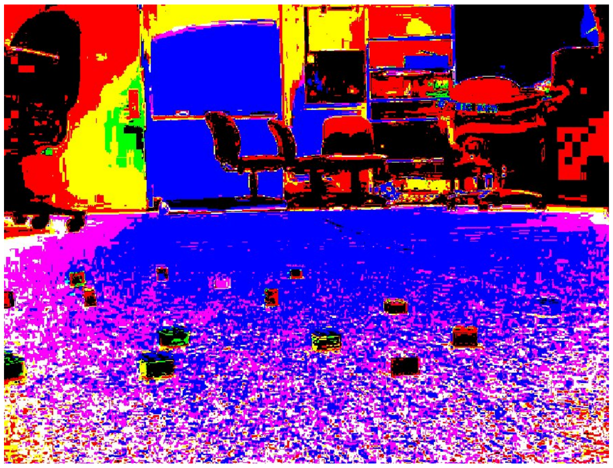
Fig. 3. Background subtraction and thresholding

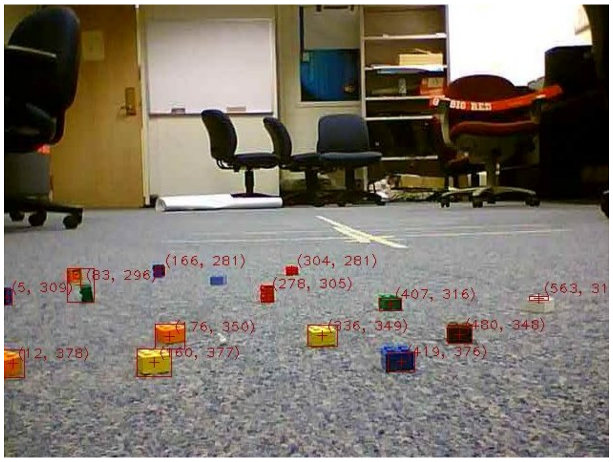

Fig. 4. Blob detection

improvement to this method could be to adaptively change the range depending on how far the center of the blob is from the Rovio since Legos close to the Rovio will appear larger. It turns out that due to the camera position, this step was not strictly necessary. If the camera was an inch from the ground this idea would be required. The blob detection phase will return the pixel location of the lowest detected Lego on the image, or in other words, the closest Lego to the robot.

1) Finding the 2-D ground coordinates of the object: All of the objects we are interested in are on the floor. We can take advantage of this fact by using the pinhole camera model to convert the pixel coordinates of the Lego on the webcam image to an (x,y) coordinate relative to the Rovio. The geometry is such that it can calculate the (x,y) coordinates given that we take calibration measurements of the following three values beforehand from a test image: (1) the number of pixels the test object is located in the image under the horizon, (2) the actual distance from the Rovio to the test object, and (3) the height of the camera from the ground. Note that the test object must be located directly in front of the Rovio for these calibration measurements.

In order to determine where the horizon line is in the webcam, the method we used was to place the Rovio looking down a long hallway and mark the position on the webcam image where the end of the hallway was. For the calibration of the distance to the test image, we used a value of 5 feet. After measuring these values, we are able to calculate the focal length, f = 700px. Using this, we can now calculate a 2-D coordinate for the location of an object on the ground, relative to the position of the Rovio. x_{pixel} and y_{pixel} denote the pixel location with the bottom middle of the webcam image set as the origin. h denotes the height of the camera from the ground (3.5in). The following equations were used to calculate the 2-D ground coordinates of the object:

$$\begin{aligned} x &= \frac{y \times x}{f} \\ y &= \frac{f \times h}{y_{pixel}} \end{aligned}$$

One of the problems that were encountered using the Rovio was that the video feed would drop and the OpenCV code could not retrieve the image data. We were able to circumvent this problem by retrying until an image was loaded. There is also a related issue where the Rovio image data was corrupted but it was still readable and would result in inaccurate data. For this, the controller would disregard a sudden change in the data by confirming it's existence over several frames.

IV. Controller

Smooth and consistent movement is quite difficult with the Rovio robot. The rotate in place command rotates the robot a minimum of about 20 degrees; this is too much for the control we were seeking. Fortunately due to the Rovio's unique three wheel design, different modes of motion can be used to change angle. The motion we used was rotating only the rear wheel. This motion makes the Rovio sweep around in a circle and was particularly useful for our project as it can rotate around a Lego without moving it and without requiring any complicated high level path planning. An important note is that since it is a circle, at a certain distance moving one way will switch the way the angle of object changes with respect to the robot.

Originally we thought we could map all the Legos into a global reference frame and then use high level path planing to move them all into a pile. We decided against this when we discovered how inaccurate localization and control was.

One attempt to improve localization was to read encoder values. This was done by modifying Interface.py so that the velocity actually returned accumulated encoder values. In the controller, the different commands would reset the encoder values when they were finished. While doing control based on encoders seemed promising at first, tests showed that even simply moving forward two feet could have errors of a foot or more. Another thing we tried was to put a Kalman filter on the overhead beam localization. While this seemed to work alright it was lagged behind the robot true position too much and didn't handle angle wrap around properly. Without mapping it turned out that only a vague sense of there the origin was needed to group all the Legos together given a long operating time.

To get something basic working the robot follows just a simple behavior. When there are no more commands left, the commands are set to rotate until it finds a Lego, move towards the Lego while rotating if it gets too out of center, rotate to face the origin, drive to the origin, backup, and then rotate in place scanning for more Legos as illustrated in Fig. 5.

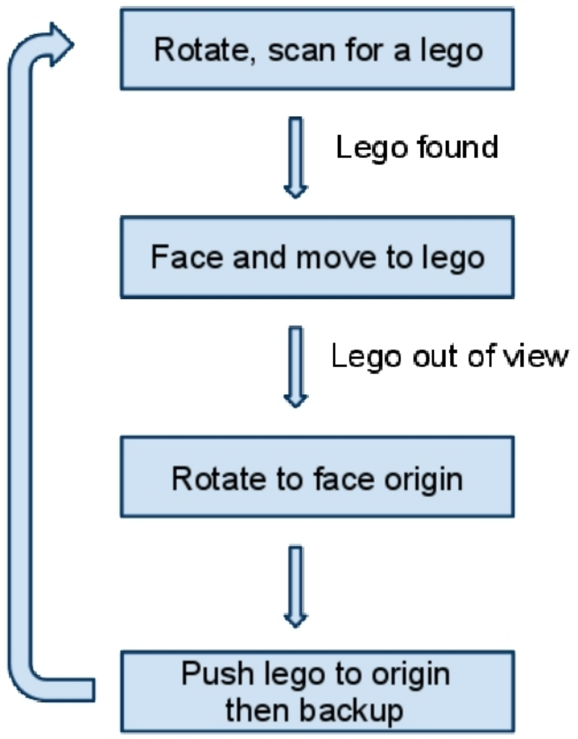

Fig. 5. Block diagram of the controller

To abstract sending the robot commands, we created a stack of commands for the main control loop to execute. The array cmds stores the values of the commands to pass and the array cmdTypes stores what type of command it is. A counter numCmds keeps track of how many commands there are. When a command is finished it simply decrements numCmds, allowing the next command on the stack to be executed, otherwise the current command keeps being used in the main loop.

A problem encountered is that sometimes information from the Rovio halts. A check to help prevent improper behavior when this happens was to have the robot pause and do nothing if the odomotery wasn't updated since the last main loop iteration.

V. Future Works

Localization: What we have now is fairly good mapping of Lego positions, but only with respect to the robot body frame. We need localization working well to map these positions to a global frame. This would make the processed data more useful, especially since the Lego is out of sight when the it is closer than about a foot to the robot. An advanced technique that might work well would be to use SLAM since the Legos provide good markers to identify and track, although there is the added complication that the robot is moving the landmarks.

Planning: Once we have better localization we could use high level planning techniques to gather the Legos more efficeintly. To be able to group Legos by type and color we would need a high level planner that could navigate around obstacles so as to not push the wrong Legos.

A. *Better Lego Identification:*

Legos come in besides many colors, many shapes as well. The 2x2 bricks we used were particularly good for detection since they have roughly the same size from every perspective. More complicated methods would be required to identify different types of Legos.