**Project 1: Dead Reckoning and Tracking**

**CS 3630: Intro to Robotics and Perception**

**Team: JurassicPork**

**Members:**

**John Turner**

**David House**

**Tim Pincumbe**

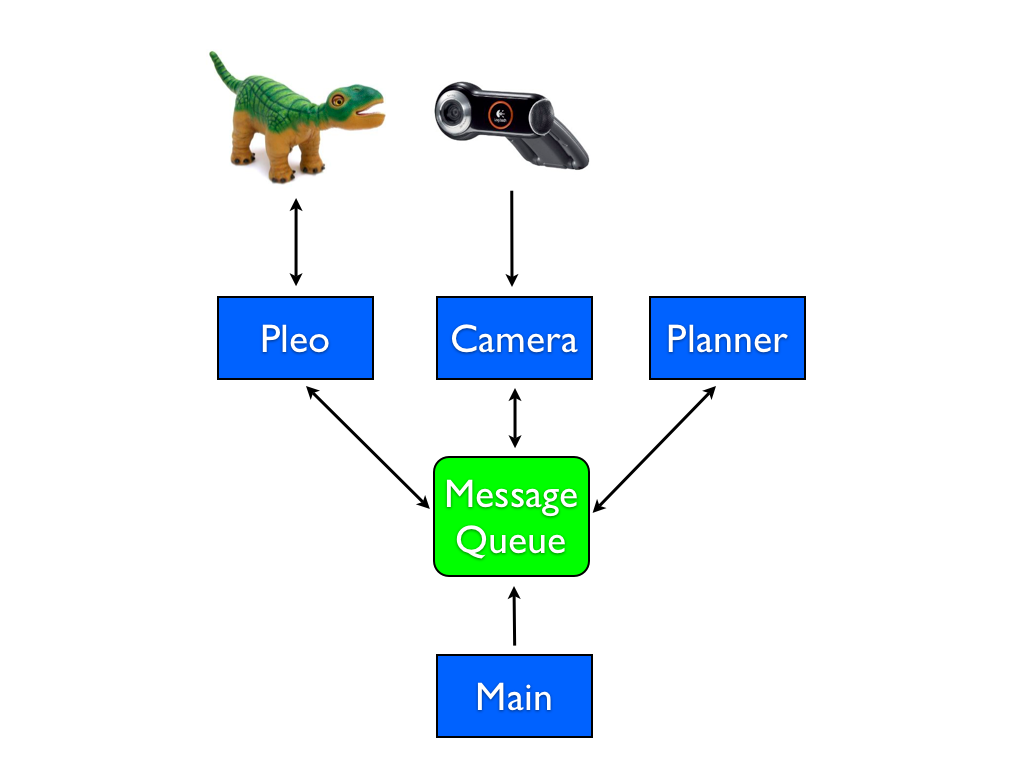
**Shashank Chamoli**

**Part I: Odometry**

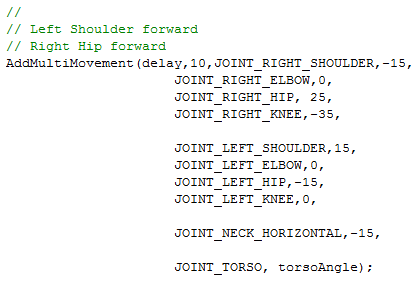
Algorithm:

In order to accomplish the first project, we decided to build our own framework that was multi-threaded. We used the MiGIO code as a reference, but the only part that was kept the same was the serial interface calls to the Pleo.

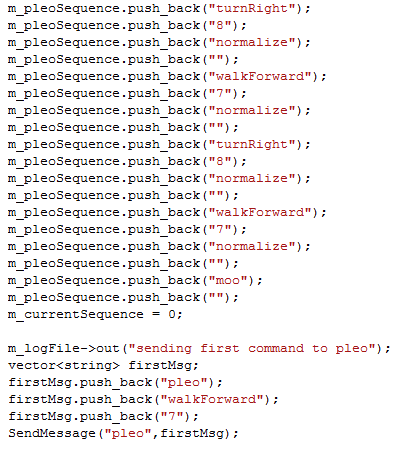
The framework was built around the concept of sub-systems, with a sub-system for the Pleo, the camera, and a planner. Each sub-system runs in its own thread, has its own log file and can communicate with the other sub-systems through a shared queue. This allowed us to separate the work for project 1 very well, and we believe it will also provide us a head start into the further projects in the class.

The sub-systems are started from the main entry point in the program from an interactive command prompt it provides. The user can start specific sub-systems or all of them. Users can also issue commands to the sub-systems to control their behavior. For example, we have a 'walkForward' command in the Pleo sub-system that the user can initiate directly from the interactive prompt. The architecture of the system can be seen in the following diagram:

The Pleo sub-system handles all the communications and behaviors of the Pleo. We used the existing serial functions provided by MiGIO to communicate with the Pleo for sending joint commands and for retrieving sensor values. We first implemented basic movement in this sub-system with commands like 'liftHead' and 'lowerTail'.



These commands simply provide for a specific set of joint values that are sent to the Pleo. We also implemented a movement sequencer so that we could create more complex movements like walking and turning. These sequences are just an array of joint angles that we send to the Pleo over time. There is a delay between each send when we are sequencing, and we have the option of repeating the sequence over and over if necessary. For many of the sequences, we added a cycles parameter instead, so we can instruct the Pleo to perform the walkForward movement sequence a particular number of times. This was key in allowing Pleo to move a fixed distance.



The Camera sub-system implements the OpenCV code that captures images from the camera and displays Pleo's current position and path. It will be described in Part II.

The Planner sub-system is the area of the code that determines the overall behavior for the other sub-systems. For this project, we implemented two planner behaviors: performSquare and performTriangle. These behaviors will instruct the Pleo and Camera sub-systems to perform their behaviors to implement the overall desired result. For example, the planner is the one who decides how far Pleo will walk forward, and when to turn. In the future, we believe that the planner sub-system will grow more complex as it interprets more input from the camera and instructs Pleo in a more dynamic way.

For the perform square part of the project, we used observation and trial and error to get pleo to walk approximately 1m and turn 90 degrees. We used this observation to instruct the planner on exactly how many cycles of the walkForward and turnRight commands to issue to the Pleo. Because we are using dead reckoning for this project, we didn't incorporate any adjustments from the camera data into the commands to the Pleo. Unfortunately due to variables that are out of our control such as friction on the carpet and drag of the USB cable, these movements aren't very precise.

Questions:

1. For each segment of your path, what is the difference between the target displacement and the one you achieved? Repeat the experiment at least 3 times and report averages.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Expected | FW 1m | Right 90o | FW 1m | Right 90o | FW 1m | Right 90o | FW 1m | Right 90o |
| Trial 1 |  |  |  |  |  |  |  |  |
| Trial 2 |  |  |  |  |  |  |  |  |
| Trial 3 |  |  |  |  |  |  |  |  |

2. When the trajectory completes, how far is the robot from its initial position and orientation?

\*\* Do 3 squares. Turn down the capture rate so we have fewer dots on the path. Capture 3 screenshots, calculate average displacements. (tim/shashank)

3. What are the reasons for this error? Discuss the assumptions you made about how the robot functions and list any possible causes that it does not do exactly what you expect.

- We first assumed Pleo was symetric (with regard to measurements and weight distribution) along the X axis, but we found this wasn't completely true – his left hind leg knee was not quite as accurate as the right hind leg knee, often undershooting the desired angle, and we had to compensate for this.

- When moving two joints on the same limb, the order that they changed wasn't consistent, so our final movement was impacted.

- Surface impacted performance of the walk, even in different parts of the carpet.

- His feet can get caught on the carpet and this throws off the gait and direction.

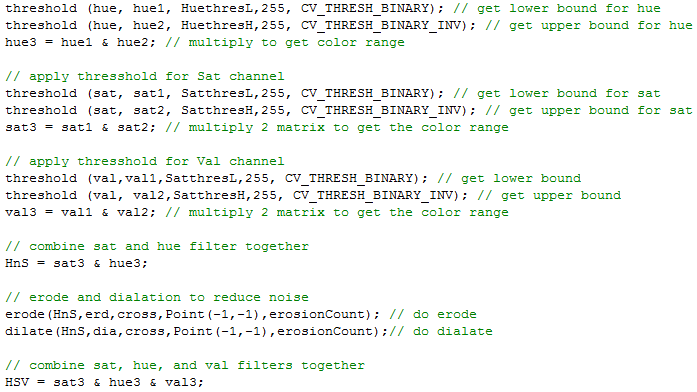
- Querying data on the serial port too fast seems to cause it to slow down when sending joint commands.

- Age of pleo - noticed eratic behavior of some joints, where they would not extend fully, or would report themselves normalized when they were still obviously offset.

**Part II: Tracking**

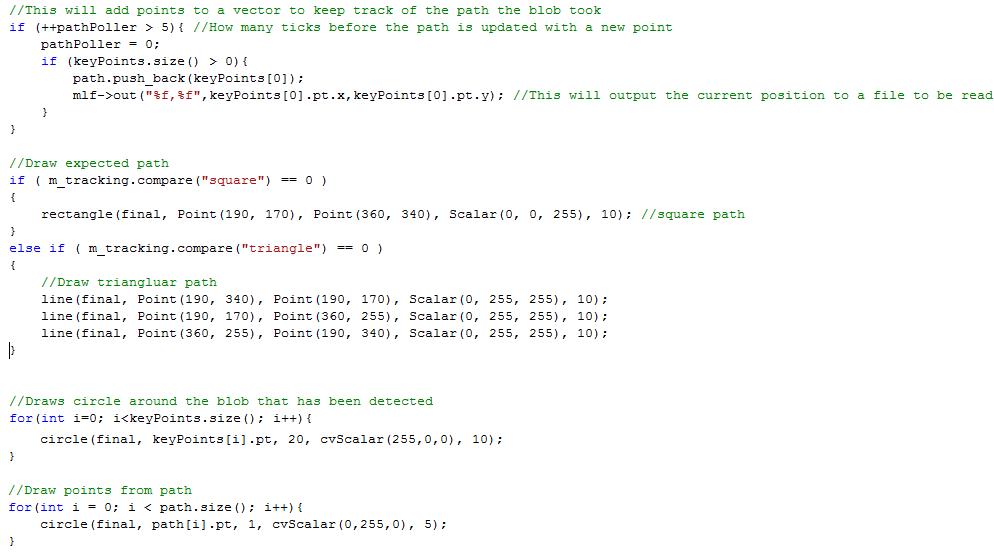
Algorithm:

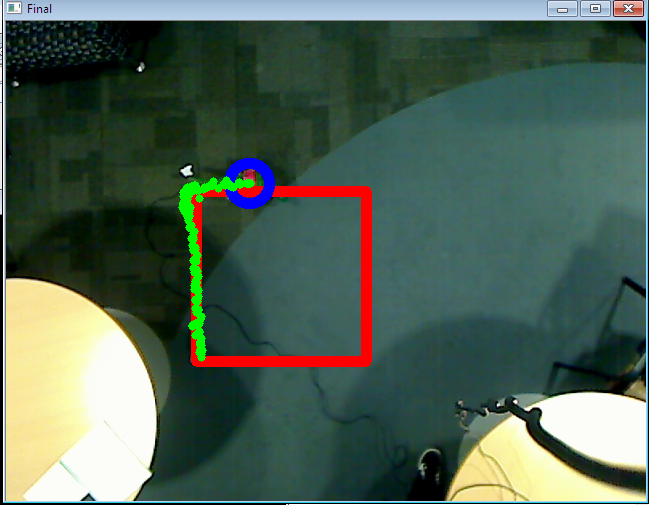
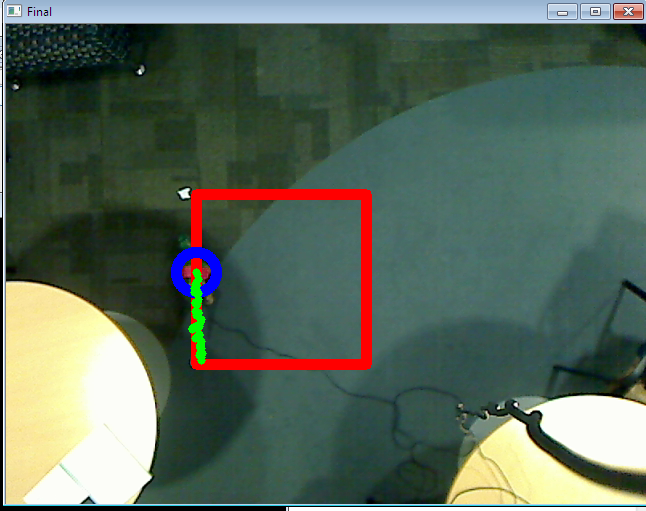
To track Pleo we used OpenCV to capture an image and draw dots representing his position on it. We used the simple blob detector built into OpenCV to do the actual tracking of the robot. The starter code for the tracking system was provided by Dr. Stillman. We start by capturing the image from the camera then blurring it to smooth the image for the blob detector. We then convert this blurred image to Hue, Saturation, and Value types. We again blur the image to smooth out the conversion. Once we have a smooth HSV converted image we can set the thresholds for hue, saturation, and value. We do just this next by setting the hue, saturation, and value thresholds for the color we wish to track, which was a red piece of paper attached to Pleo. Once we have the thresholds we multiply the lower and upper bound to get the range in a single variable. The following is a snipet of how we set the thresholds and set the range in a single variable.

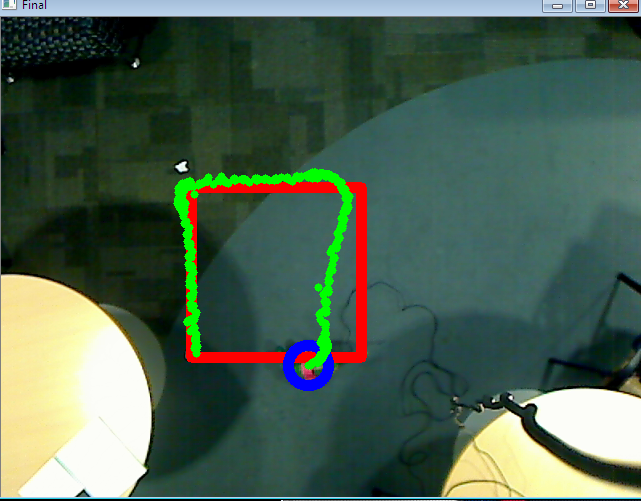
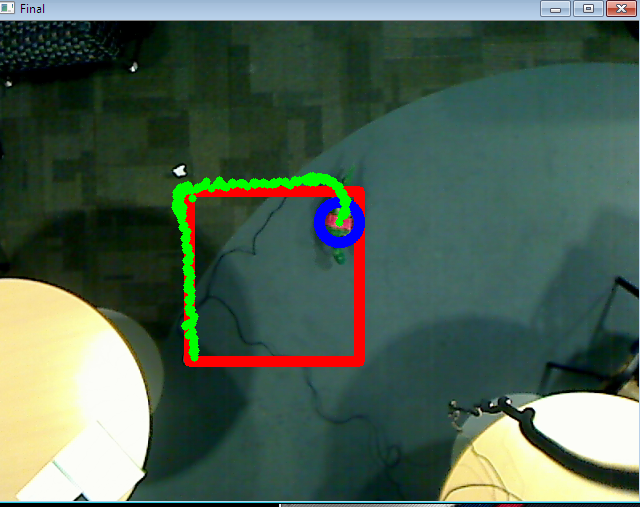


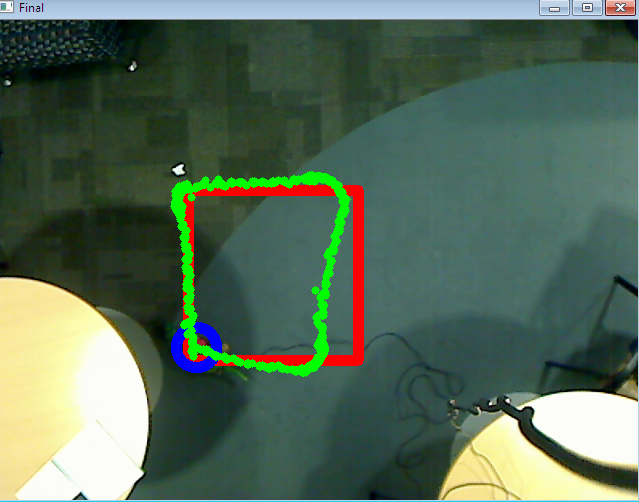
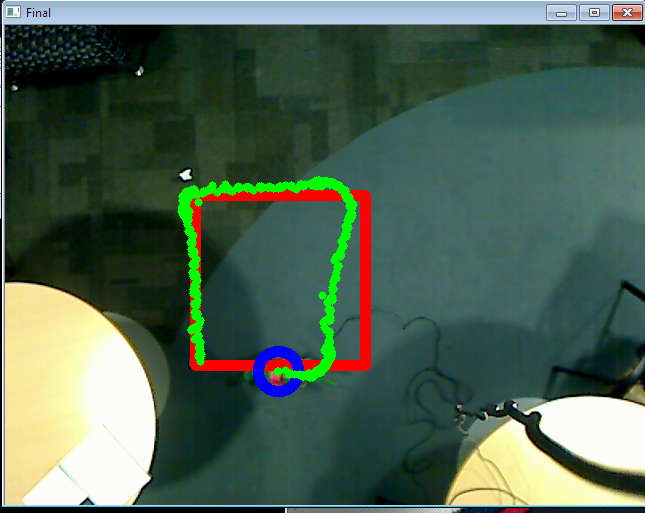
To distinguish the paper we used to track we found that only the hue and saturation made a difference to actually detect, but using val cut down on noise. Once we have the ranges for hue, saturation, and value we multipled them together to get a range for HSV in a single variable. We then eroded and dilated the image to reduce the amount of noise in the image. This creates a black and white image that allows the blob detector to work more efficiently.

With the image complete for tracking we created a blob detector using OpenCV’s simple blob detector. Since the camera was hung from roughly 3m we had to constrain the size of the blob that could be detected. Once we had the parameters set we started the detection of the blob and put those values into an array of OpenCV Key Points. We used these points to draw a circle on the screen to show us how it tracked. We also kept track of these points to build the path of the robot every five frames. Using these path points we drew dots to show how he moved along the expected path. We also outputted these points to a file for later analysis.









Questions Part 2:

Assume that the camera is pointing straight down and the world frame is the point on the ground that shows up at the center of the camera image with Xw pointing right in the image and Yw pointing up. Assume the robot starts somewhere in the lower-left corner of the image facing upwards (Yw direction).

1. Use a homogeneous transform to represent the relationship between the world coordinate frame and the camera coordinate frame. Explain in words what this transform does. Why is it useful?

--

2. Use a homogeneous transform to represent the relationship between the camera frame and the image (pixel) coordinate frame. Explain in words what this transform represents. Why is it useful?

--Image frame transform allows for pixel displacements to be used to calculate robot movement/displacement.

3. Give the homogeneous transform that maps world coordinates to image (pixel) coordinates.

-- imagecoords = MiMc(wcoords)

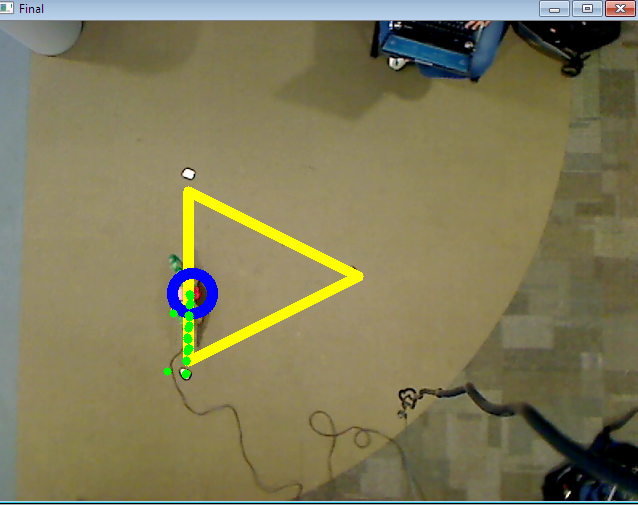
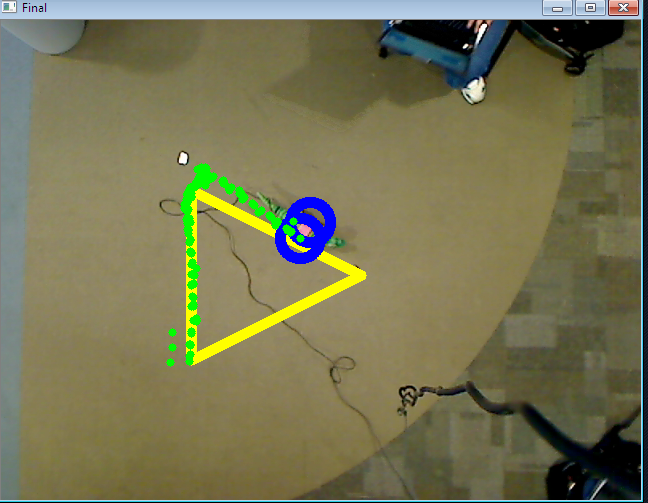
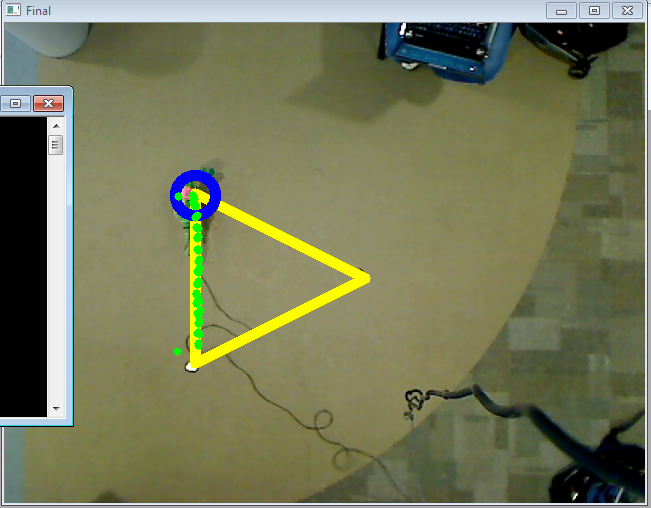
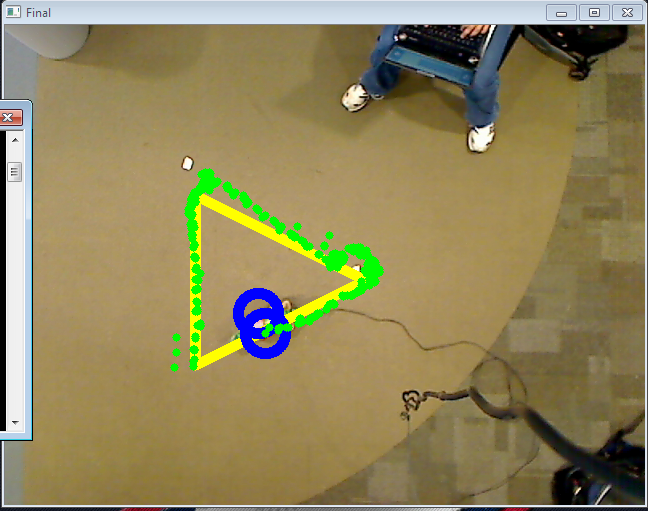
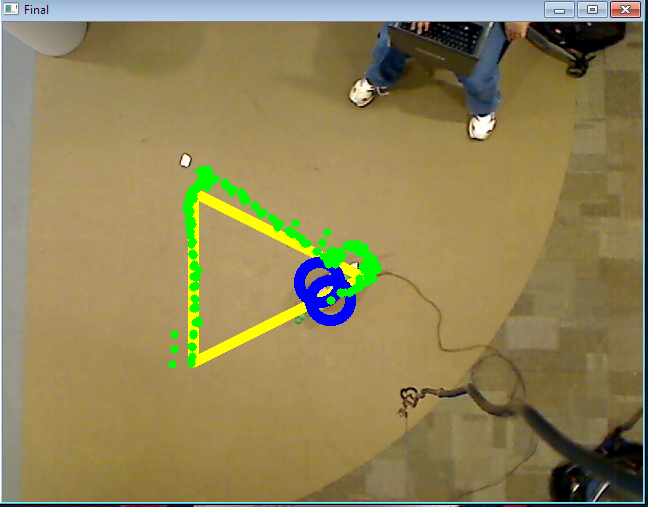
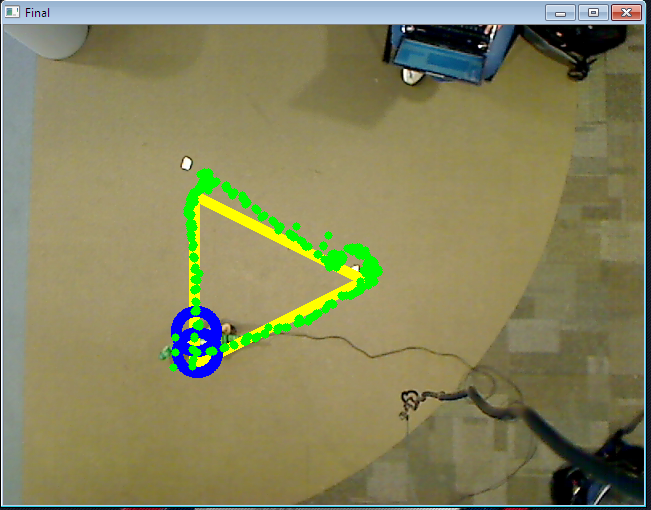
4. Use the transforms above to draw a square in OpenCV that represents the target robot trajec- tory and include a screenshot in the report. Also, include a screenshot of the robot executing the trajectory.

5. What is the Euclidian distance (in pixels) between the robot’s location at the end of the first segment of the trajectory and the predicted end of the first segment? Similarly, what is the distance between the robot’s position of the whole trajectory and the expected location?

6. Identify the relationship between the errors in pixels and the errors you found in Part I.

**Part III: Analysis**

We decided to implement a triangle as our second shape. We chose an equilateral triangle that starts straight for 1m, then turns 120 degrees, continues straight for another 1m, turns again 120 degrees, and then does a final 1m straight. We utilized our same walkForward and turnRight motion sequences that we created in the Pleo sub-system to implement the triangle. The walkForward didn't need to be changed because our square was also 1m, however we had to increase the cycles of our turnRight to get the 120 degree turn.

Questions Part 3:

1. Design your own arbitrary trajectory for the robot to follow (be creative!). Repeat the steps from Parts I and II for this trajectory and show pictures that demonstrate the expected trajectory for the robot in OpenCV and the actual trajectory the robot followed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Expected | FW 1m | Right 120o | FW 1m | Right 120o | FW 1m |
| Trial 1 |  |  |  |  |  |
| Trial 2 |  |  |  |  |  |
| Trial 3 |  |  |  |  |  |

2. Clearly, your robot never achieves the exact trajectory you expect. Lets think about what can be done to rectify this issue. Answer the following questions:

(a) Suppose you can use the overhead camera while you control the robot. Can you get your robot to achieve each target point on the trajectory? If so, how? Provide less than 10 lines of pseudo-code that might be used to implement your solution.

Yes.

trajAra = array of target points (Xworld,Yworld) to trajectory.

locationAra = array of robot location (Xworld,Yworld) readings read by camera.

calculate the delta from the actual path versus the ideal trajectory if this delta crosses a threshold then adjust movement to compensate in the opposite direction keep doing this

(b) Now, suppose we don’t give you the overhead camera while you control the robot. Is it still possible to improve the accuracy with which your robot reaches the target points? If it is possible, describe a method that could be used. If not, explain why not.

If the robot has external sensors at its disposal with which to acquire an accurate sense of it's orientation and position in the world frame, and some accurate means of proprioception within the robot, with which to maintain real-time knowledge of the robot's actuator movements and subsequent joint-space position and orientation, then this data could be used by the robot to correct itself when it deviates from the expected trajectory. Otherwise, without some source of information relating the robot's world-space position to its joint-space displacements, correcting deviations from its path would be impossible.

**Other:**

**Contribution:**

**References:**