Project 1: Dead Reckoning and Tracking

Group Name:
Gundam

Group Members:
Larry Freil
Farzon Lotfi
Alex Strange

Part 1: Odometry

Algorithm:

For this part of the project we used a simple forward-and-turn-right algorithm. The code was a simple loop where we repeatedly sent the Forward command to the robot for a certain amount of time, then a turn 90 degrees command, and then repeat the whole process four times. In order to determine how long we needed to send the forward command, we timed the robot using a stopwatch and just sent a continuous forward command from the web interface and measured how many seconds it would take the robot to go one meter. Then using that value, we determined how many times we needed to send the robot the Forward command in order to get it to achieve this distance. We tested various turning speeds and times to determine one that most closely achieved a 90 degree turn. Once we had both of these components, we just put them together in sequence to get a square.

Questions:
1 & 2:

Displacement from desired location:

	First Corner	Second Corner	Third Corner	Final Corner
1st Trial	.17m	.09m	.12m	.21 m
2nd Trial	0.25m	0.36m	0.25m	.03m
3rd Trial	.18m	.22m	0.18m	.05 m
Average	.2 m	.22m	.18m	.1m

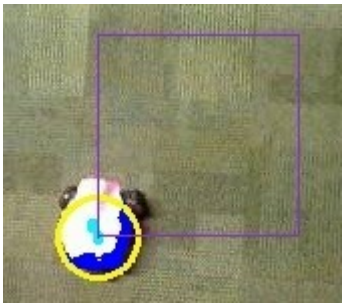
All final orientations ended up being within +/- 5 degrees of the original position during the 3 trial runs for the above graph. However, we have sometimes observed larger errors in the final orientation depending on how smoothly the robot performed its run.

3: The reasons for some of this error is the inherent lack of precision in controlling the robot by just telling it to go Forward over a period of time. Without the robot actually doing the timing, the time has to be estimated by the controlling laptop. Therefore interruptions and delays in communication can sometimes cause jittery robot movements, and can affect the final position of the robot. Also, without the ability to measure exactly how far the wheels have turned, there is no way to correct for the variations in the speed the wheel motors turn when the battery voltage starts to drop. To account for some of this, we tried to always keep the robot as fully charged as possible so the robot would behave more consistently.

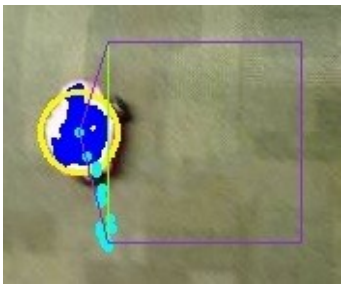
In addition to just the difficult in ensuring the proper speed of the robot, it is hard to get the robot to turn precise amounts for similar reasons. Also, there is an additional fact that the robot does not have a precise Turn 90 degrees command, so to get the robot to turn correctly, we had to try a mix of turn for a time, and turn by 20 degree increments (which there is a command for) in order to approximate a 90 degree turn to the best of our abilities.

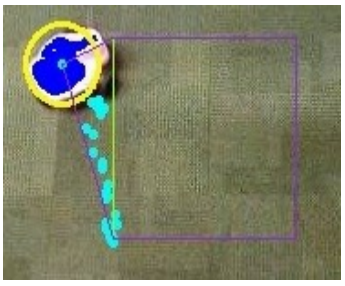
Part II: Tracking

Algorithm:

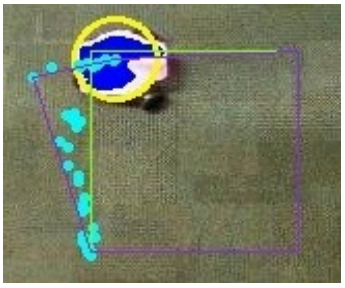


In order to track the robot, we decided to go with the blob detection method discussed in class. To do this, we needed the robot to be a distinctive color that easily differentiated it from its surroundings. Therefore, we covered the robot with a red piece of construction paper. Then we modified and made some improvements on the code Ana showed us in class in order to make it work better with our robot. We also found that in many cases, the construction paper would reflect light from the overhead lights back into the camera, thus causing the construction paper to appear white. To account for this, we increased threshold, which caused the algorithm to detect brighter colors in general. Also, we added to the algorithm a filter that would remove any detected blobs that were to large to be our robot. This fixed a problem we occasionally had where if the camera tried to auto-adjust the brightness and contrast settings, it could temporarily change almost every pixel to a color that would be detected as the desired color. Filtering out large blobs prevented any of these background blobs from being detected. The dark blue in the images to the right is the detected robot blobs.

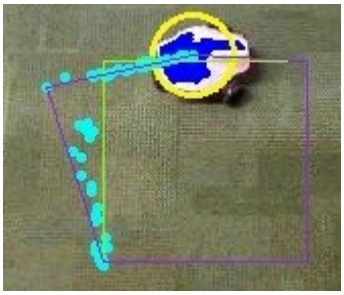




Once we had the robot detected, we used the center of the blob as the robots position. For each frame of the video from the webcam, we detected the robots location, and saved it to a list of all of its locations. Using the first recorded position of the robot, we could use the camera transform to calculate the corner positions of the one meter square and to draw it on the image with a green square (usually hidden under the purple square). Once the robot started moving we displayed the list of robot positions as cyan dots to represent everywhere the robot has been. Also, using the average distance and directions between the past several positions allowed us to calculate the direction the robot was traveling in. Using this information, we could determine which part of the square the robot was in, and update our predicted path accordingly (in purple).



Questions:



1. The world coordinate frame is a 3D coordinate system consisting of scenes from the object coordinate frame that have been rotate and translated into a scene yielding object coordinates in the world coordinate frame. The purpose is to relate objects in three dimensions. The camera coordinate frame is a 3d coordinate system whose purpose is to represent objects with respect to the location of the camera. Therefore, what a Homogeneous transform from the world coordinate frame to the camera coordinate frame would do is translate the points of the world coordinate system into a direction with reference to the camera.[1] For example let's say the object reference frame in the world coordinate frame is (-5,3,2) and we want to translate the point into the direction of the camera (4,3,0,1)

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 4 \\ 3 \\ 0 \\ 1 \end{bmatrix} = \begin{pmatrix} 9 \\ 6 \\ 2 \\ 1 \end{pmatrix}$$

This gives us the point (9,6,2) . This is useful since we may want to look at our scene from a particular viewpoint (the "camera"). What this does is set the camera to the origin of the coordinate system which orients the scene so the camera is looking down one direction of the z-axis. The "up" direction is typically the positive y direction.

2.

The pixel coordinate frame is a 2D coordinate system. Each pixel in this frame has integer pixel coordinates. Therefore to convert to this frame you must first convert the camera frame to an Image plane coordinate frame which will describe the coordinates of the 3D points on to the image plane.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 9 \\ 6 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 13 \\ 13 \\ 9 \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 13 \\ 13 \\ 0 \\ 0 \end{bmatrix} \Rightarrow \begin{pmatrix} 13 \\ 13 \end{pmatrix}$$

Here we perform the Homogeneous transform and drop the last two coordinates since pixels have no depth associated with them.

3.

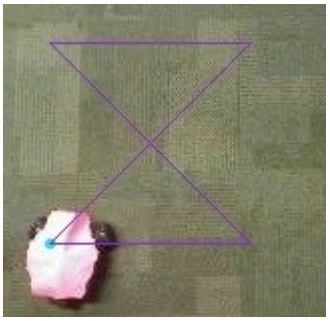
$$perspective\ transform = \begin{bmatrix} 1 & 0 & 0 & -5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This overall perspective transform can be used to convert real world coordinates into pixel coordinates directly, without the use of any intermediate transforms.

4. (see images on previous page)

5. At the end of the first segment, the robot was approximately 31 pixels distant from the desired location. By the end of the last segment, the robot stopped about 53 pixels distant from the desired final location.

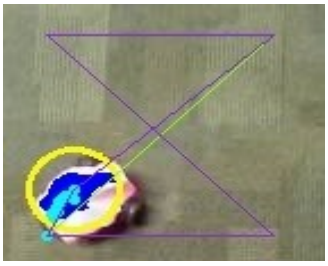
6. Naturally there is a linear relationship between the errors in pixels, and the errors in physical location of the robot because we are not really changing distances from the camera, we are only changing the scale of movement. Therefore, distances in our world coordinate system simply require a scalar (in this case) to covert to distances in our pixel coordinate system.



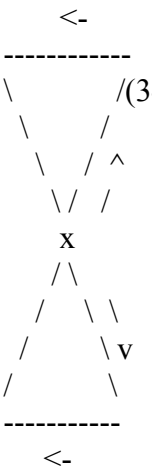
Part 3: Analysis

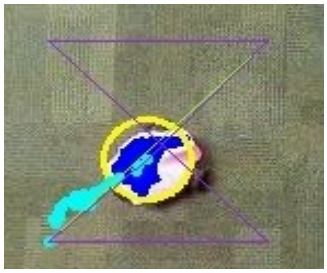
We designed a trajectory for the robot in the shape of an hourglass/figure eight:

The robot units mentioned refer to the number of times we sent the Forward command to the robot.

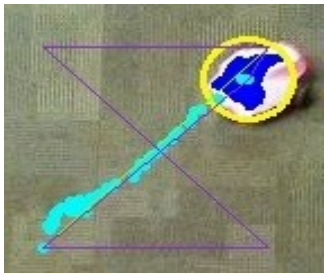


1 meter = 24 robot units

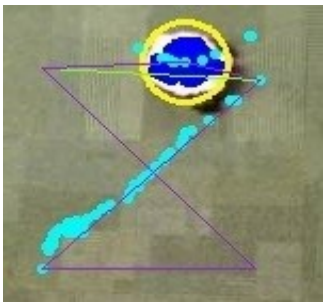




The starting point was in the lower left corner. The trajectory was enclosed in the same square meter as Part I.



	First Corner	Second Corner	Third Corner	Final Corner
1st Trial	.05m	.08 m	.15 m	.04 m
2nd Trial	.06m	.10m	.08m	.03m
3rd Trial	.05m	.13m	.25m	.12m
Average	.05m	.10m	.16m	.06m



The orientation for the first two final positions was within +- 5 degrees of the desired orientation, however during the third trial, there was a momentary loss of signal, and the final orientation was about 20-25 degrees off from the desired direction.

The main difference between this part of the assignment, and the first part of the assignment, is that it seems that sometimes turning smaller amounts reduces the overall error in the turning so that we are more in line with the desired direction we want the robot to travel. Also, due to the rules of trigonometry, if we are slightly off in one of our diagonals, there is a much smaller effect on the next leg of the path.

2.

When using dead-reckoning, there is no way to correct the trajectory, because the only thing we know is that the robot has finished its last predetermined command.

We can improve this by adding sensors. Because the robot heads in vectors with arbitrary angles, we need to correct for two kinds of errors - the possibility of it traveling at the wrong angle and the possibility of it finishing at the wrong distance along the vector.

a. We can plot the expected route on the camera's image and correct the robot's path if it falls off it.

```
Pseudocode:
// at a target point
     drive(next point - current position)
     while (current position != next point) {
          if (next point is not straight ahead)
               drive left/right to correct sideways drift
          if (stopped)
               drive(next point - current position) // correct
                                                   stopping too far/not far
                                                   enough
}
```

b. Without the overhead camera, we need to use some other sensor. On Rovio, the possibilities are:

- using the encoders on the wheels. These are not accessible to us, and the robot itself already uses them to drive forwards. Any errors are caused by inaccuracies here, so reusing their data probably won't help.
- the beacon on the Rovio base. This gives the robot x/y coordinates relative to the base - we can save coordinates for the target points and correct it by driving towards those. This should work if the base is available, but there might be some problems due to the imprecise sensor.
- the onboard camera on the Rovio. In this case, we move the robot to each target point and save the image from the camera. As long as the appearance of the environment doesn't change, when the robot is placed nearby the same area, the program can correct its path by trying to find the same view. Sideways drift can be corrected by shifting the robot so the center of the image is in the same place. Incorrect distances can be corrected by making sure some landmark is the correct size. This should be the best method, but requires the most work to get perception working.

References:

1. Nvidia – The Cg Tutorial - http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter04.html
-Used this to find more information on how to use homogenous transforms.

Contributions:

Larry -
- -Developed our blob detection software and robot tracking
- -Worked on the write-up
- -Taped construction paper to the robot

Farzon-
- -Calculated our homogenous transforms.
- -Wrote the program that drove the robot in a square.
- -Worked on the write-up

Alex -
- -Programmed the alternate hourglass route
- -Assisted the development of the program that drove the robot in a square.
- -Worked on the write-up