Project 1: Dead Reckoning and Tracking

CS 3630: Intro to Robotics and Perception

February 8, 2012

Team: JurassicPork

Members:

John Turner

Tim Pincumbe

Shashank Chamoli

David House

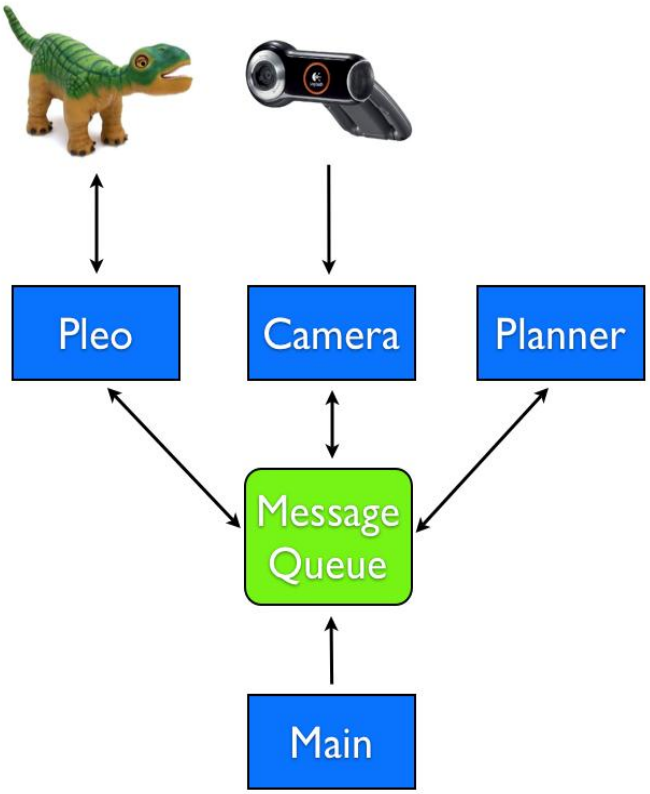
Part I: Odometry

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'.

Below is a sample code for one step in the movement for walking forward:

```
//
// Left Shoulder forward
// Right Hip forward
AddMultiMovement(delay,10,JOINT_RIGHT_SHOULDER,-15,
                    JOINT_RIGHT_ELBOW,0,
                    JOINT_RIGHT_HIP, 25,
                    JOINT_RIGHT_KNEE,-35,

                    JOINT_LEFT_SHOULDER,15,
                    JOINT_LEFT_ELBOW,0,
                    JOINT_LEFT_HIP,-15,
                    JOINT_LEFT_KNEE,0,

                    JOINT_NECK_HORIZONTAL,-15,

                    JOINT_TORSO, torsoAngle);
```

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.

Code snippet from the planner sequencer to perform a square:

```
m_pleoSequence.push_back("turnRight");
m_pleoSequence.push_back("8");
m_pleoSequence.push_back("normalize");
m_pleoSequence.push_back("");
m_pleoSequence.push_back("walkForward");
m_pleoSequence.push_back("7");
m_pleoSequence.push_back("normalize");
m_pleoSequence.push_back("");
m_pleoSequence.push_back("turnRight");
m_pleoSequence.push_back("8");
m_pleoSequence.push_back("normalize");
m_pleoSequence.push_back("");
m_pleoSequence.push_back("walkForward");
m_pleoSequence.push_back("7");
m_pleoSequence.push_back("normalize");
m_pleoSequence.push_back("");
m_pleoSequence.push_back("moo");
m_pleoSequence.push_back("");
m_currentSequence = 0;

m_logFile->out("sending first command to pleo");
vector<string> firstMsg;
firstMsg.push_back("pleo");
firstMsg.push_back("walkForward");
firstMsg.push_back("7");
SendMessage("pleo",firstMsg);
```

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.

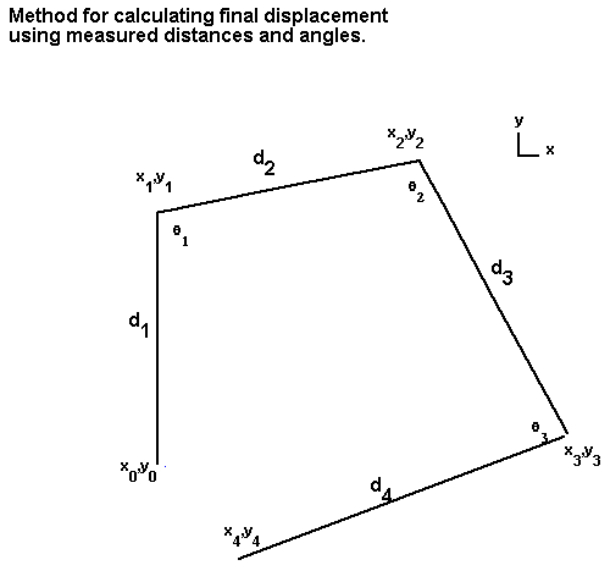
Questions Part I:

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 100cm	Right 90^o	FW 100cm	Right 90^o	FW 100cm	Right 90^o	FW 100cm	Right 90^o
Trial 1	**95cm**	**~95^o**	**95cm**	**~95^o**	**95cm**	**~90^o**	**105cm**	**~100^o**
Trial 2	**100cm**	**~90^o**	**90cm**	**~80^o**	**105cm**	**~80^o**	**95cm**	**~95^o**
Trial 3	**100cm**	**~90^o**	**85cm**	**~100^o**	**105cm**	**~90^o**	**90cm**	**~90^o**
Average	**98.3cm**	**91.7^o**	**90cm**	**91.7^o**	**101.7cm**	**86.7^o**	**96.7cm**	**95^o**

2. When the trajectory completes, how far is the robot from its initial position and orientation? We used the following mechanism to calculate the final displacement of the trajectory:

Answer:



(values are exaggerated in picture to illustrate relationships)

Where $x_2 = x_1 + d_2*\cos(\theta_1)$
And $y_2 = y_1 + d_2*\sin(\theta_1)$
And $x_3 = x_2 + d_3*\cos(\theta_1 + \theta_2)$
And $y_3 = y_2 + d_3*\sin(\theta_1 + \theta_2)$

This is continued for all subsequent points, calculating the sines and cosines to find each subsequent coordinate.
And then the final displacement is the Euclidian distance from x_0,y_0 to x_4,y_4

Trial 1:	Displacement: 19.2cm
	Orientation $380^o = 20^o$ clockwise from beginning orientation
Trial 2:	Displacement: 24.4cm
	Orientation $345^o = 15^o$ counterclockwise from beginning orientation
Trial 3:	Displacement: 13.8cm
	Orientation $365^o = 5^o$ clockwise from beginning orientation
Average:	Displacement: 19.13cm
	Orientation $370^o = 10^o$ clockwise from beginning orientation

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.

Answer:
There are many possible explanations as to why our robot did not consistently finish at its initial position. First, we made a few assumptions when beginning the project and initially developing the movement code. The biggest were assumptions regarding the robot's sagittal plane symmetry. We assumed the dimensions of the various components of the robot were equal and the weight was evenly distributed across the left and right sides of its body. We also assumed that the responses of equivalent joints would be the same e.g. telling the left knee to bend 45^{o} would have the same end result as telling the right knee to bend by the same amount.

We used these assumptions in designing our walking gait, but experience showed us that occasionally such assumptions were faulty. This required us to adjust our code either by varying the desired angles we assigned to each joint or, on occasion, completely rebuilding the entire gait in an effort to avoid problematic configurations of joint angles.

For example, during our experiments we found that the knee on Pleo's right hind leg did not respond consistently to our commands to straighten, often remaining bent in the middle of a pushing step; while the left knee did not always respond to a bend command as accurately as the right, often undershooting our desired angle and dragging the foot on the ground. To compensate for this we not only had to use a different angle for the right knee than the left, but we also had to include a head-swinging motion, where the head would swing to the opposite side of Pleo's body than the knee being bent, so that the robot would pivot on his two planted legs in such a way as to lift the hind leg off the ground. This would minimize the negative fallout from improper angles on his hind legs (since they were only bent as we brought them forward in the gait). We also applied packaging tape to his left hind leg's toes to assist in compensating for the same thing.

Unfortunately, such measures sometimes caused the robot to respond in a very chaotic manner. For example, Pleo's response to the head swing, where he would pivot around an axis formed by the opposite hand to the direction of the head swing (which would be planted) and the opposite hind leg to the planted hand, was difficult to predict with 100% certainty.

Most of the time this maneuver had the desired effect, but on occasion, the robot would either not pivot, get stuck in mid step and possibly not move forward at all, or else swing too forcefully turning in the direction of his head bob and experience a gross deviation from trajectory. While our three trials were thankfully devoid of the more extreme forms of the latter, more egregious errors, small deviations due to both of these error conditions were common and were at least

partly to blame for many of the errors we saw in the forward displacement.

We also assumed that the proprioception sensors within the robot would be consistently accurate in their reporting of the robot's joint angles, which proved to not always be the case. The robot's joints often undershot the desired angle while reporting back that they had in fact reached it. Without being able to give the robot feedback this was impossible to compensate for and also contributed to inappropriate gait response especially with regard to turns.

When sending a command to move two joints on the same limb, such as a hind leg, the order in which the joints would respond did not always remain the same. For example, usually the knee joint would bend almost simultaneously with the hip when both were given commands to bend, but this was not always the case and sometimes the knee would bend much later. This would cause the robot to catch in certain parts of a step and not complete the movement properly.

Another issue that impacted movement was the surface walked upon. The robot would either not move or move backward on smooth surfaces where as it would move properly on carpet. Even on different types of carpet the robot would sometimes get stuck and not move correctly, or else move inconsistently, favoring one side or the other and turning during a maneuver intended to be straight.

The last major issue we had was that we sometimes queried the serial port too fast. Most of our commands were sent in loops to have the robot move multiple steps. If the time between the loops was set too quickly the serial port communications seemed unable to maintain an accurate connection and some movements would either not be received or else not responded to correctly. This would cause the robot to move erratically, again deviating from our expected trajectory.

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. Stilman. 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.

```
threshold (hue, hue1, HuethresL,255, CV_THRESH_BINARY); // get lower bound for hue
threshold (hue, hue2, HuethresH,255, CV_THRESH_BINARY_INV); // get upper bound for hue
hue3 = hue1 & hue2; // multiply to get color range

// apply thresshold for Sat channel
threshold (sat, sat1, SatthresL,255, CV_THRESH_BINARY); // get lower bound for sat
threshold (sat, sat2, SatthresH,255, CV_THRESH_BINARY_INV); // get upper bound for sat
sat3 = sat1 & sat2; // multiply 2 matrix to get the color range

// apply thresshold for Val channel
threshold (val,val1,SatthresL,255, CV_THRESH_BINARY); // get lower bound
threshold (val, val2,SatthresH,255, CV_THRESH_BINARY_INV); // get upper bound
val3 = val1 & val2; // multiply 2 matrix to get the color range

// combine sat and hue filter together
HnS = sat3 & hue3;

// erode and dialation to reduce noise
erode(HnS,erd,cross,Point(-1,-1),erosionCount); // do erode
dilate(HnS,dia,cross,Point(-1,-1),erosionCount);// do dialate

// combine sat, hue, and val filters together
HSV = sat3 & hue3 & val3;
```

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.

The following is the code for dispalying the expected path, actual path and blob detection:

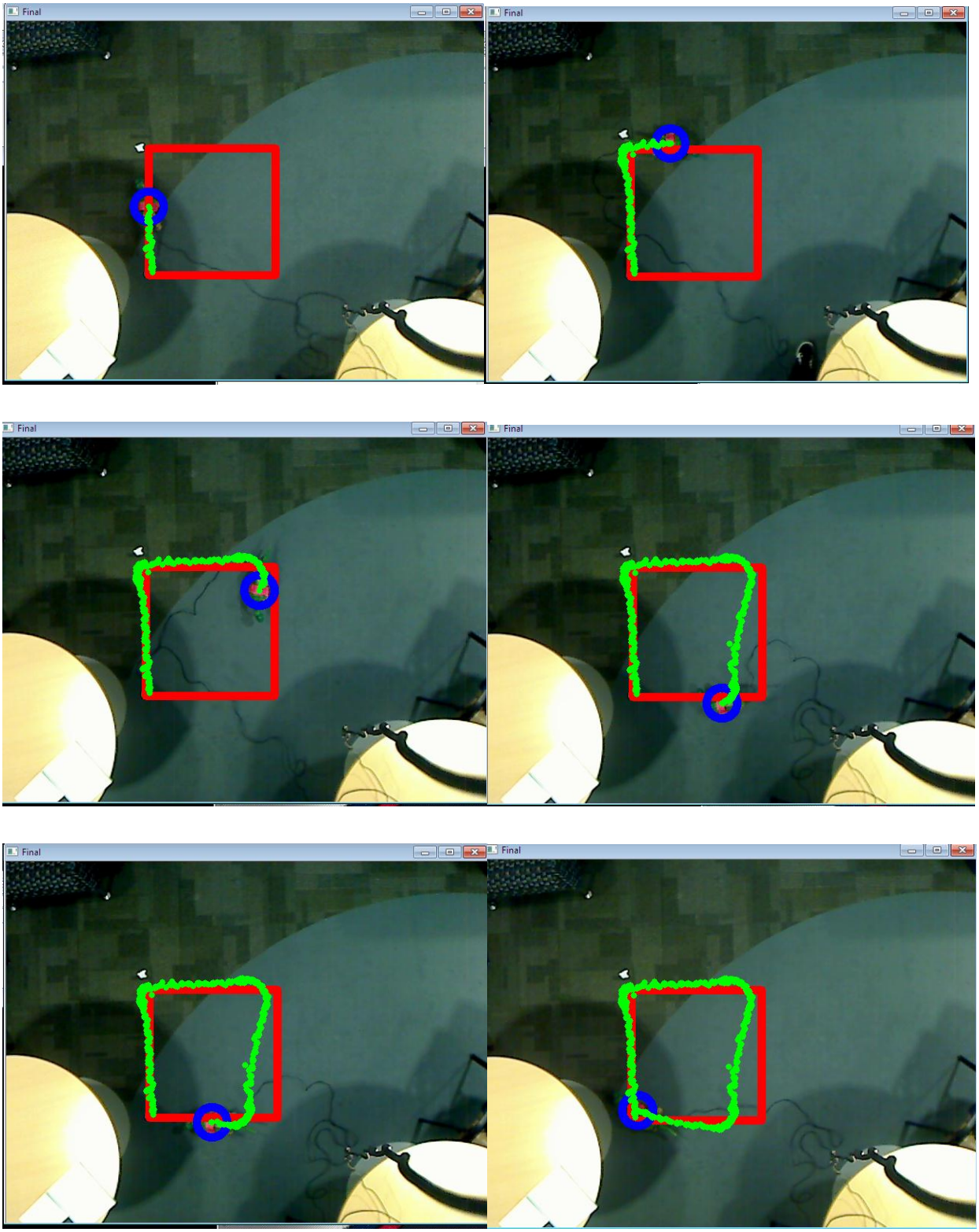
```
//This will add points to a vector to keep track of the path the blob took
if (++pathPoller > 5){ //How many ticks before the path is updated with a new point
    pathPoller = 0;
    if (keyPoints.size() > 0){
        path.push_back(keyPoints[0]);
        mlf->out("%f,%f",keyPoints[0].pt.x,keyPoints[0].pt.y); //This will output the current position to a file to be read
    }
}

//Draw expected path
if ( m_tracking.compare("square") == 0 )
{
    rectangle(final, Point(190, 170), Point(360, 340), Scalar(0, 0, 255), 10); //square path
}
else if ( m_tracking.compare("triangle") == 0 )
{
    //Draw triangluar path
    line(final, Point(190, 340), Point(190, 170), Scalar(0, 255, 255), 10);
    line(final, Point(190, 170), Point(360, 255), Scalar(0, 255, 255), 10);
    line(final, Point(360, 255), Point(190, 340), Scalar(0, 255, 255), 10);
}


//Draws circle around the blob that has been detected
for(int i=0; i<keyPoints.size(); i++){
    circle(final, keyPoints[i].pt, 20, cvScalar(255,0,0), 10);
}

//Draw points from path
for(int i = 0; i < path.size(); i++){
    circle(final, path[i].pt, 1, cvScalar(0,255,0), 5);
}
```

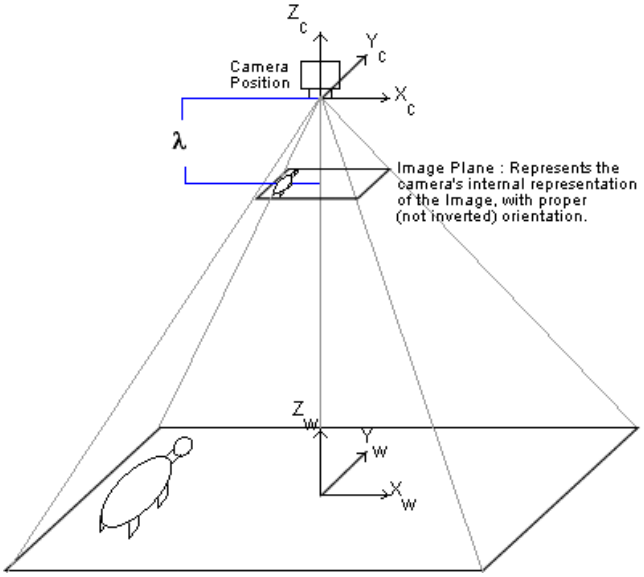
Pleo performing a square:



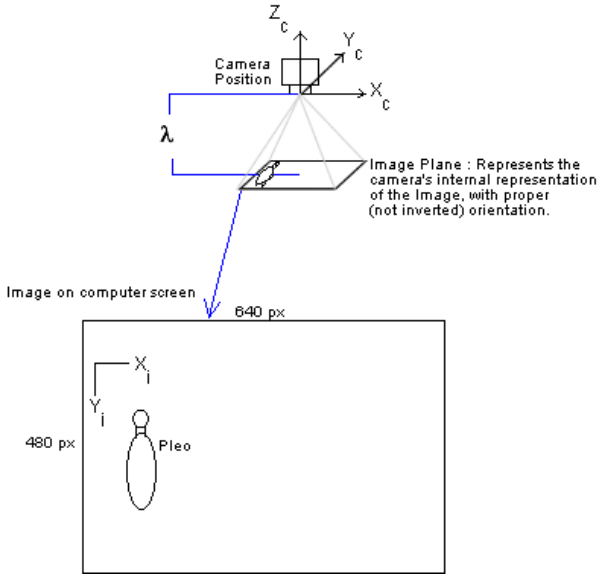
Questions Part 2:

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).

World Frame to Camera Frame transformation :



Camera Frame to Image(pixel) Frame transformation :



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?

Answer:
First we need to calculate λ, the focal length of the camera. We can find this value through this equation, using similar triangles :

$$x_C = \lambda \frac{X_W}{Z_W}$$

Where X_w is a particular length in the World Frame, Z_w is the above the X_w,Y_w plane where the camera can be found, and x_c is the length of X_w in the Camera Frame, upon the Image Plane.

We observed a 1 meter distance for the square in the X_w, Y_w plane, in the x direction and the y direction, 2.85 m from the camera, which ended up being translated to 170 pxls on the screen (i.e. Image frame (pix)) in the same direction, yielding a focal length equation as follows :

$$\lambda = 170px * \frac{2.85m}{1m}$$

Yielding a distance of 484.5 pixels. This is asuming pixels as Camera Frame distance units, which doesn't really make sense in the real world but is usable in our example since the Camera Frame is primarily important for transitioning from World Frame to Image (pixels) Frame for our project, and pixels are the Image(pixels) Frame unit we are using.

The matrix equation we used to make this transformation is :

$$\overrightarrow{x_c} = T_c^W \overrightarrow{X_W}$$

Where x_c is the vector representation of a point in the Camera space, X_w is the vector representation of a point in the World space, and T_c^W is the Transformation matrix from the World Frame to the Camera Frame.

$$\begin{bmatrix} x_C \\ y_C \\ \lambda \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda \frac{X_W}{Z_W} \\ \lambda \frac{Y_W}{Z_W} \\ \lambda \\ 1 \end{bmatrix} = \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ \frac{Z_W}{\lambda} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2.85 \\ 0 & 0 & \frac{1}{\lambda} & 0 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2.85 \\ 0 & 0 & \frac{1}{484.5} & 0 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

This transformation converts coordinates in the World Frame to coordinates in the Camera Frame through a projection matrix, where X and Y coordinates in the World are scaled down by a factor based on the focal length λ of the camera (in pixels, explained above), and, since we are considering the Camera equivalent to a pinhole camera, the Z coordinate is effectively ignored with regard to the image itself, except to preserve the Scaling factor through the transformation. It does come into play in the translation from axes

Because we are using homogeneous equations, the actual values for x_c and y_c in "camera pixel units" can be found by multiplying the resultant x and y values given by the matrix above by a constant $\lambda/Z_w = 170$ pixels/meter.

This transformation is useful because it takes a 3-D space (the world-frame) and transforms it into a 2-D space while preserving fairly accurately the relative distances and angles being displayed within the XY plane. This creates an accurate mapping of points in 3D to a 2D representation, making recognizable and realistic 2D pictures of the world possible. This is also necessary when dealing with 3D objects via a computer, since a computer has no mechanism for displaying 3D images accurately to a user. It also converts the units from physical distance (meters) to pixels, for representation on the screen

This transformation can be used in reverse to take image data and use it to calculate real-world quantities, such as using a picture of a set of displacement locations and calculating the real-world displacement they are representing.

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?

Answer:
To calculate the transformation from the Camera Frame to the Image(pixel) Frame, we just rotated the axes and translated the x_c and y_c coordinates, since they were already in units of "pixels" from our previous transformation and choice of λ.

$$\overrightarrow{x_{pix}} = T_c^{pix} \overrightarrow{x_c}$$

$$\begin{bmatrix} x_{pix} \\ y_{pix} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 340 \\ 0 & -1 & 0 & 240 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ \lambda \\ 1 \end{bmatrix}$$

This transformation moves the coordinate axes from their location in the center of the image in the Camera Frame to the upper left hand corner of the Image (Pixel) frame, the common location for the origin for images on computers, and it also performs a 90^o counterclockwise rotation around the X_c axis, also to facilitate the display of the image on the computer.

This transformation is useful because it enables us to illustrate an image on the screen of the computer from the information given by the camera, referenced to axes common to computer graphics.

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

$$\overrightarrow{x_{pix}} = T_c^{pix} T_W^c \overrightarrow{X_W}$$

$$\begin{bmatrix} x_{pix} \\ y_{pix} \\ 1 \end{bmatrix} = \begin{bmatrix} 170 & 0 & 0 & 340 \\ 0 & -170 & 0 & 240 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

This will translate the World coordinates measured in meters from the center of the X_w,Y_w frame to the appropriate image (pixel) coordinates.

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

Answer shown above.

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?

Answer:
For one of our best trials of the square pattern the robot started at position 197px, 332px. After the first segment the robot ended at 187, 166 when it should have finished at 197,170. This gives us a euclidian distance of $\sqrt{(187-197)^2+(166-170)^2} = 10.77px$
The end point of the square pattern was at 170px, 305px. This gives us a euclidian distance of $\sqrt{(170-197)^2+(305-332)^2} = 38.18px$

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

Answer

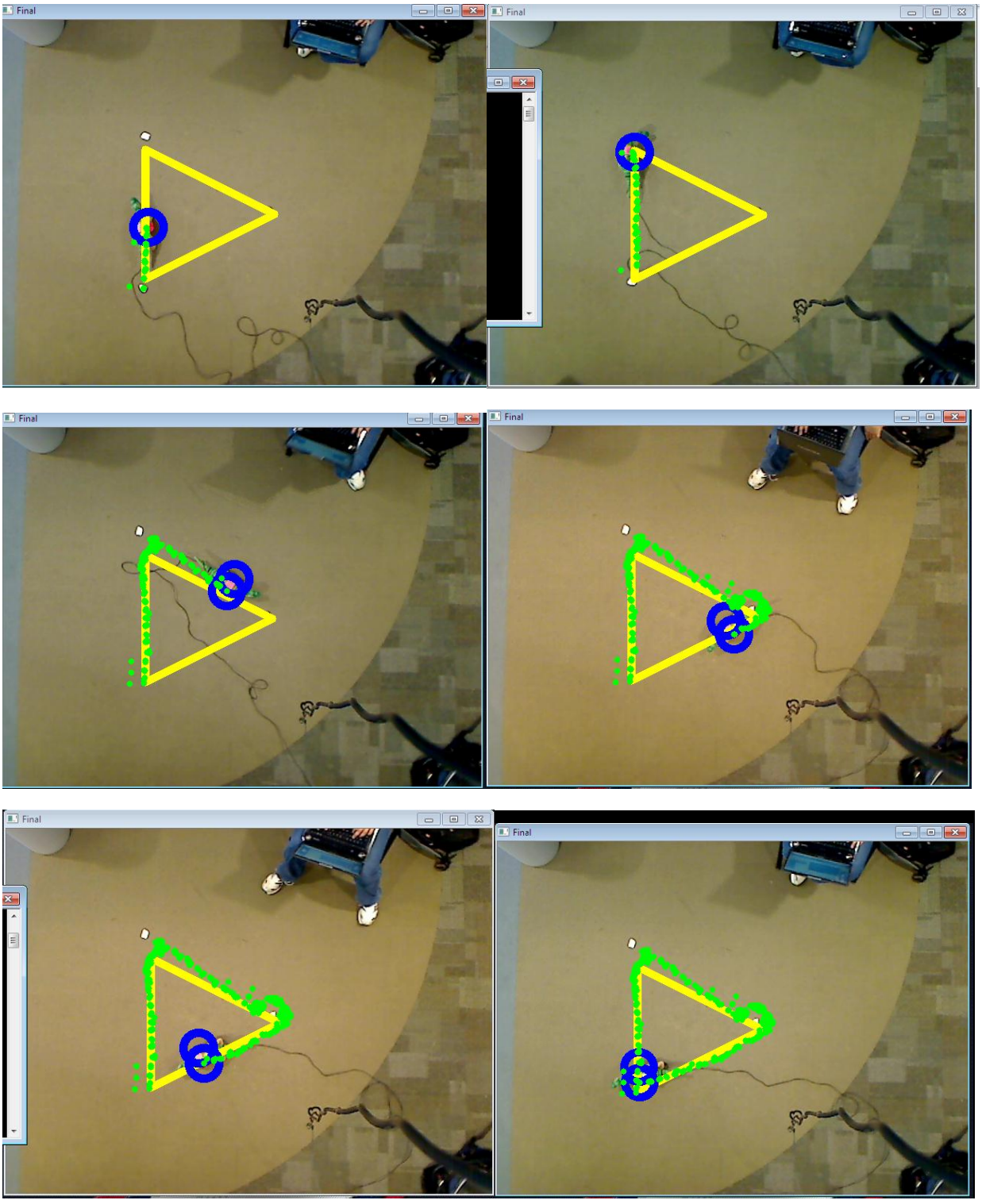
They are related by the λ/Z_w multiplier, in that an error distance of x meters off trajectory in the World Frame (Part 1) equates to an error of 170x pixels in the Image (pix) frame.

Part III: Analysis

Algorithm:

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.

Pleo performing a triangle:



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.

Pictures of triangle on previous page.

Expected	FW 100cm	Right 120^o	FW 100cm	Right 120^o	FW 100cm	Right 120^o
Trial 1	**95cm**	**~120^o**	**100cm**	**~125^o**	**100cm**	**~120^o**
Trial 2	**100cm**	**~125^o**	**95cm**	**~110^o**	**95cm**	**~125^o**
Trial 3	**95cm**	**~125^o**	**90cm**	**~115^o**	**100cm**	**~125^o**
Average	**96.7cm**	**120^o**	**95cm**	**116.7^o**	**98.3cm**	**116.7^o**

Deviations from expected location at the end of the triangle, using the same method as Part 1 for the square:

Trial 1: Displacement: 2.4cm
Orientation 365^o = 5^o clockwise from beginning orientation
Trial 2: Displacement: 8.98cm
Orientation 360^o = 0^o deviation.
Trial 3: Displacement: 14.83cm
Orientation 365^o = 5^o clockwise from beginning orientation
Average: Displacement: 4.37cm
Orientation 370^o = 10^o clockwise from beginning orientation

Homogeneous Transform from World Frame to Camera Frame and Camera Frame to Image Frame are the same as for the Square.

Calculated Trajectory and Robot Performance:

Euclidian Distances

For one of our best trials of the square pattern the robot started at position 197px, 332px. After the first segment the robot ended at 201, 160 when it should have finished at 190px, 170px. This gives us a euclidian distance of $\sqrt{(201-190)^2+(160-170)^2} = 14.87px$
The end point of the square pattern was at 197px, 315px. This gives us a euclidian distance of $\sqrt{(197-197)^2+(315-332)^2} = 17px$

2. Clearly, your robot never achieves the exact trajectory you expect. Let's 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.

Answer:

Yes.

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

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

delta = difference between trajAra and locationAra at index i

distanceThreshold = amount of displacement tracked by camera before robot is considered "off course".

```
if delta > distanceThreshold
    if locationAra is left of trajAra
        move right
    if locationAra is right of trajAra
        move left
```

(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.

Answer:

If the robot has external sensors at its disposal with which to acquire an accurate sense of its 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, since it would not even be able to detect that such displacements had occured.

Other:

Contribution:

Tim Pincumbe took over the OpenCV and tracking part of the project. He modified the sample code given from Dr. Stilman and made it work with the Jurassic Pork Project. He added in code to calculate the saturation and value thresholds along with the hue thresholds to detect the color of the blob better. Tim added code to track the Pleo and build the path using a vector of KeyPoints. These points were used to draw green dots to show the path as well as outputted to a text file. He used these points to determine the expected path of the square and triangle and drew them onto the screen. Tim also contributed to the report with the other team members.

Shashank Chamoli worked with Tim initially to finalize the thresholds needed to correctly detect the Pleo using the OpenCV blob detection. He also helped with getting the sample code (provided by Dr. Stilman) running to provide a base code. Aside from general help with the trial of the Pleo and debugging, he was worked on the recording of data and the report, making sure it had the correct format and looked professional and presentable.

John Turner programmed the gaits – walkForward and turnRight, determining appropriate joint angles for each frame of the gait sequence through trial and error. He also mapped the sequences of moves to perform each of the desired maneuvers – square and triangle. John also contributed to the report, calculating the matrices for part 2, the mechanism for calculating the total displacement in part 1, as well as answering questions regarding gait response and control accuracy.

David House, who has since dropped the class, modified the MIGiO framework to handle multi-threading, and also helped develop the early versions of the gaits we were intending on using, and wrote the first rough draft of this report.

References:

John Turner used notes from another class (CS 3451 Graphics) as well as notes from class.