University of Pennsylvania
Center for Sensor Technology
Philadelphia, PA 19104

SUNFEST REU Program

SONY AIBO MOTION CALIBRATION AND MONITORING/CONTROL SYSTEM

NSF Summer Undergraduate Fellowship in Sensor Technologies
Microsoft Fellowship Recipient
Brian Corwin (Computer and Telecommunications Engineering) – University of Pennsylvania
Advisor: Dr. Daniel Lee

ABSTRACT

The Sony Aibo are robotic dogs used to test new ideas in artificial intelligence, computer vision, and robotic motion through the application of having the dogs play soccer. An important part of building the system the dogs use is debugging. There are two key parts to this process: calibrating the dog's motions, and monitoring the dog's sensory outputs and control input. Both parts of the process help the developer to understand how the dog perceives the world and acts in it. Each of these questions was addressed with a different system. A magnetic positional sensor device was used (along with dog control and data processing programs) to obtain calibration information about the dog's various walks and kicks. A client/server system was developed to allow multiple users to access different sensory outputs, the dog's command input (only one user at a time), and the dog's standard output in order to facilitate system development and debugging.

Table of Contents

1. INTRODUCTION

The Sony Aibos, like any intelligent agent, interact with the world through a sequence of perception, evaluation, and action. In this project, the Aibo's actions were calibrated, and a system to monitor the Aibo's perceptions (and also control its actions directly) was created. Calibrating motions determines how commanded motions (and their expected results) compare with the actual motions performed. Information about this can be used to adjust the system, so the dogs' expected actuations are carried out as accurately as possible. Controlling the dog (or letting the dog control itself) while monitoring its different sensory and system outputs is an important step in understanding and debugging the dog's sensor, information processing, and self-control systems. In other words, in order to understand what the dog is doing, one must "see" the world as the dog does by viewing its sensory outputs. A system using a sensor device (for precise positional information), associated program, and calibration scripts was used for motion calibration, and a client/server program was created to allow for dog control and observation of dog outputs by multiple users at once. In the future, it is hoped, these systems will be integrated to give the developer more tools.

This paper outlines the design of these two systems as well as the results from two iterations of motion calibration. Section 2 provides background on the hardware and operating system of the Aibo, the software systems developed by Dr. Daniel Lee for the Aibo to use for soccer, and the principles, hardware, and operating system for the Nest of Birds (NOB) sensor device used in the motion calibration. Section 3 discusses the creation of a control program for the NOB as well as calibration scripts. The section also covers the results of some of the motion calibrations done for the dogs. Section 4 covers the design, implementation, and use of the monitoring and control server for the Aibo. Section 5 covers the design and use of the first prototype client created to interact with the server. Discussion and conclusions are presented in Section 6 and future work and recommendations in Section 7.

2. BACKGROUND

2.1 Sony Aibo and Robocup Soccer

The Sony Aibo (see Figure 1) is a robotic dog used by universities for research, particularly to participate in Robocup Soccer. This is an event sponsored by the Robocup Federation in which robots of different types (including the Sony Aibo) play soccer; in the Aibo case the game is played with teams of four [1, pp. 1-4]. All perception, decision, and actuation activities are performed by the Aibos themselves, although communication and coordination is allowed (with strict rules) between dogs through a wireless network [1, pp. 1-4].

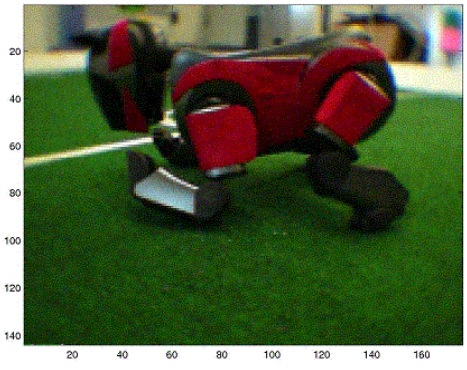

Figure1: The Sony Aibo.

These tournaments create an activity through which new ideas about computer vision, robotic motion, and artificial intelligence are developed and tested. After the annual tournament, each team writes a report describing their system, so that groups learn from one another and advance scientific development.

The Sony Aibos are themselves a powerful platform on which to work. Each dog has four legs, each with a motor in the hip, knee, and ankle that allow for three degrees of freedom in each leg [2, pp. 7-12]. The dog's head and neck has three motors that give it three degrees of freedom (pan, tilt, and roll) [2, pp. 6-12]. A camera in the dog's head has a focal length of 161 pixels (2.18 mm), operates at 25 frames per second, and takes images that are 352 x 288 pixels in size [2, p. 20]. The Aibo also has a short-range distance sensor, two-channel microphone, and speaker [2, pp. 19-21]. Each dog also has a slot for an IEEE 802.11 wireless Ethernet card. The Aibo operating system also has a built in Application Programming Interface (API) using Sony's OPEN-R technology. The technology allows custom programs to be written that operate the dog [3], running off memory sticks inserted into a slot on the dog. A TCP/IP networking stack allows the Aibo to communicate with other Aibos and computers over a wireless Ethernet [4].

2.2 Aibo Software

Dr. Daniel Lee and colleagues have created an extensive software system that runs the Aibo team. At the lowest level, code written in C++ interacts directly with the Aibo's API to handle some of the basic functionalities of the dog, such as its vision system. On top of this is the core high-level decision-making program consisting of a state machine [5, p. 1]. Previously this layer was implemented in C/C++, but a more flexible solution was found in running a PERL script on a PERL interpreter embedded in the Aibo [5, p. 1]. PERL had several advantages including the ability to recover from errors, ease of development, and extensibility that allowed the low-level C++ functionalities to be called through a limited interface, allowing for maximum implementation encapsulation [5, p. 2].

2.3 Important Aibo Inputs/Outputs

Although the Aibos perform autonomously during competition, the system in Section 2.2 does have inputs and outputs to which other computers can connect, assisting in debugging and development (communication with the Aibo is described in Section 2.1). The system connects specific outputs and inputs to different ports in the dog's networking stack; for example, port 59000 is the standard output for the dog (where system information and error messages are shown), so if another computer connects to a dog's IP address on port 59000 it can retrieve these messages. In addition, 1001 is the input port, 6006 is the "blob vision" port (see Section 2.4), and 6000 is the camera port. A special format for the packets sent to these ports is specified by the OPEN-R protocol. Each packet has a total size (integer) value telling the total number of bytes in the packet and a number of data (integer) value that is the number of data elements in the packet, in addition to the data itself. For the input, the data is a string that represents a command to

be executed by the dog. This takes advantage of a useful feature of PERL: Since PERL is interpreted commands can be executed on the fly [5, p. 2].

2.4 Aibo Vision System

Each image from the camera comes out in YUV pixel format. Existing computer vision systems cannot deal with the complexity of even the simplest photograph, so in order for the Aibo to extract relevant information from the images, the photos must be simplified into a form that the Aibo's processing power and known computer vision techniques can handle. The simplification system is based on the needs of the Robocup soccer field; on the field important visual cues for the dog are distinct colors (the ball is bright orange, the goals are yellow and blue, etc.). The distinct colors give the dogs specific visual cues to look for; i.e., they only need to recognize this particular subset of visual stimuli.

Each pixel of the image is mapped to one of the important cue colors (orange, blue, green, etc.) or to nothing, in a method similar to that used by Carnegie Mellon University's Aibo team [6, p. 2]. In essence, every pixel that is close to green is mapped to green, every pixel that is close to orange is mapped to orange, etc. and everything else is ignored. Although the number of colors has been limited there are still thousands of individual pixels. The image is processed again, so that an area with a large concentration of a particular color pixel is united into one box.

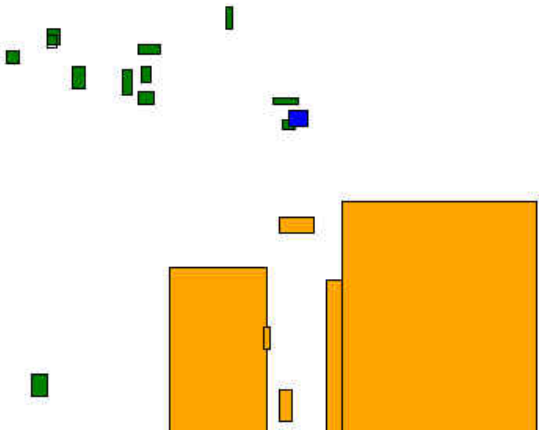

Figure 2: Blob view image.

Each box (which is also called a "blob") has certain characteristics: the position of each corner and of the centroid (the center of mass of the pixels). After processing, the image becomes an array of bounding box structures that stores the information on all "blobs" in the image. This is simple enough for the dog to process; for example, in order to find the ball the Aibo simply looks for an orange blob. Figure 2 shows a blob view image.

2.5 Nest of Birds (NOB) Design, Operating System, and API

The NOB (see Figure 3) is a magnetic sensor device with a central unit, a transmitter, and four sensors. The sensors and transmitters are attached to the central unit by cables. The device is connected to a computer via a serial port in its central unit. The NOB works by magnetic field emission and electromagnetic induction. The transmitter and the sensors all have electrical coils in the x, y, and z directions [7, pp. 2-3]. The

transmitter turns on each coil one at a time, and the magnetic field emitted by the coils induces an electric current in the coils of the sensors [7, pp. 3-4]. The amount of induction in each coil is inversely proportional to the distance to the transmitter and orthogonality of the angle of the coil to the field. Thus by comparing the induced voltages in each sensor coil (x, y, and z) when each of the transmitter coils (x, y, and z) is on, the device can tell the relative position (x, y, and z) of the sensor to the transmitter and the azimuth, elevation, and roll of the sensor relative in the coordinate system of the transmitter. The NOB has a range of 3 feet from the transmitter [8, p. 8].

Figure 3: Nest of Birds device

The API of the NOB consists mainly of commands represented by single-byte positive integers. Sending a certain byte to the NOB via the serial port tells the NOB to perform a particular action; for example, writing number 66 asks for a data point from a sensor [8, p. 88]. Many functions in API are superfluous to the simple system needed for motion calibration; so only key features will be discussed.

The NOB has certain parameters that help control its actions (all can be examined and some can be changed): the error status of each bird, the address of the transmitter, etc. The examine/change functions are used to manipulate these parameters, and thus control the actions of the NOB [8]. In this way, for example, the transmitter mode, bird hemisphere (direction of the coordinate system of the sensor), and bird data mode are all set. Each bird (sensor) has an address; these addresses are from 1 to the number of birds in the device (here, 4). The status of each bird can also be obtained, which will tell much about its state: is it master or slave, running or not, error or none, etc. The auto-configuration function tells a bird to be the master bird, through which all communications to other birds must go, and how many birds there will be. Without the auto-configuration, only one bird can be used at a time [8, pp. 131-134].

3. AIBO MOTION CALIBRATION SYSTEM

The motion calibration system was designed around the NOB magnetic sensor device in two parts. First, a C++ program was designed to initiate and configure the NOB and to obtain data from the device when asked. Second, Matlab scripts were written to obtain data from the NOB program and send motion commands to the dogs to get raw data, to process the data, and to plot it. Both of these programs were designed to work on a Linux computer. The C++ program was a completely separate program from

the Matlab scripts. The Matlab script simply executed the C++ program and redirected the program's output so that it could be parsed and used by the Matlab script.

3.1 Nest of Birds Control Program

The NOB consists of four sensors and a central unit (including the transmitter), so this kind of configuration was modeled using C++ classes, which is consistent with the principles of good object-oriented design [9, p. 85]. A class called Bird was created that contained data pertinent to an individual sensor, such as the bird address, hemisphere, data mode, and status. The class has methods that are important to a sensor. There is an auto-configure method, set data mode, set hemisphere, get status, get error, and get data packet. The system is made to work only in the Point/Angle data mode.

A class called Nest is a model for the whole device. It contains an array for all the Bird objects in the Nest as well as data members and methods of its own that are important for the NOB as a whole. It contains the transmitter address and mode (as well as methods to get and set these). This class constructor initializes the whole system by connecting to the NOB, configuring it, and creating a certain number of Bird objects based on the number used in the NOB (4 was always used).

The main function of the program creates a Nest object with 4 Birds and then sets up the Birds (all have upper hemisphere and position/angle data mode) and the transmitter (pulsed mode and address is bird 4). Then the program then simply reads points from the device. The positions and angles are 2-byte integers (± 32767). The data bytes read from the NOB are first processed (bit manipulation) and then converted to the appropriate units (centimeters and degrees). The positional integers are relative to 3 feet. Thus to convert to centimeters:

$$(pos/32767)*(36\text{ in}/3\text{ ft})*(2.54\text{ cm}/1\text{ in}) = 0.0027906125*pos$$

Angle integers are relative to 180°, so to convert to degrees:

$$(angle/32767)*(180\text{ degrees}) = 0.0054933332*angle$$

3.2 Aibo Motion Calibration Matlab Scripts

There are three basic steps to the motion calibration: 1) Obtain calibration data and store it to a file, 2) process the data and store it in another file, and 3) graph the results. The dog can be commanded through a Matlab interface. For this reason and for ease of development, Matlab was used to create these parts of the calibration software. Similar Matlab programs were developed to calibrate the dog's kicks.

The first program runs the experiment. One of the sensors is attached to the dog and the dog is placed close to the transmitter (about a foot away, to keep the sensor within transmitter range). The NOB is started and an initial position reading is taken.

The dog then walks in the way specified on the command line, x, y, and θ (in millimeters and degrees). Another reading is taken after the dog is done walking, and both the start and end readings are written to a file. The dog then tries, through a simple negative feedback mechanism, to get to the start position, so that the dog does not eventually drift out of sensor range. The dog repeats this process many times.

The data must be processed for two reasons. First, the motions that the dog is commanded to do are all relative to the end of his head and not where the sensor is (taped to a ruler on the dog's side to keep the sensor away from the magnetic interference of the dog's servo motors). So the sensor positional information and the known distance from the end of the head to the sensor must be used to get the head position. Figure 4 shows a simple diagram of the dog; the distances d1 and d2 have been measured for the Aibo and Xsen, Ysen, and θ are obtained from the NOB. The transformation from sensor to head coordinates is:

$$Xhead = Xsen - d1*Cos(\theta) - d2*Sin(\theta)$$
$$Yhead = Ysen - d1*Sin(\theta) + d2*Cos(\theta)$$

The second reason is to get the data in the right coordinate system. The x, y, and θ displacements should be in the coordinate system of the dog's start position, with the end of its head as the origin. This has two implications. First, x and y translations must done to get the dog's head to be the origin rather than the transmitter. Second, the dog's coordinate system maybe at an angle with the transmitters, so a coordinate system rotation is needed (the θ change is the same for all reference frames: $\theta_{FINAL} - \theta_{INIT.}$):

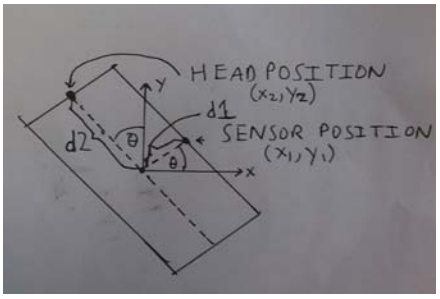

Figure 4: Coordinate system diagram.

$$Xmove = + (X_{FINAL} - X_{INIT})*Cos(\theta_{INIT}) + (X_{FINAL} - X_{INIT})*Sin(\theta_{INIT})$$
$$Ymove = - (X_{FINAL} - X_{INIT})*Sin(\theta_{INIT}) + (Y_{FINAL} - Y_{INIT})*Cos(\theta_{INIT})$$

3.3 Aibo Motion Calibration results

The C++/Matlab system was used to calibrate the x, y, and θ of the dog. Only one direction was done in a trial (e.g., the dog would never do a forward and then walk left). The x and y motions were tested from -20 to +20 cm in 2 cm increments and θ from -100º to +100º, in 20º increments, with 10 to 20 data points for each walk.

Two important pieces of information needed to be obtained from the data: the ratio between the value of the real action and the commanded action, and whether there were any significant aberrant movements (e.g., if the dog was walking forward, did it drift right or left, or turn). These types of conclusions are much harder to quantify since there are no expectations for the pattern they should follow; the ratios of

command to actual movement are expected to have a linear relationship. To find these ratios for each movement type (x, y, and θ) the average actual displacement for a particular distance command was found, and a linear regression was performed on these data (see Table 1):

Motion	Linear Regression
Y (Forward/Back)	0.7757
X (Right/Left)	0.7172
Theta (Counterclockwise/Clockwise)	0.9107

Table 1: Results from linear regressions on calibration data.

These regressions were not sufficient, since the graphs clearly show that for all motion types the results for positive and negative motions were very different (see Appendix A). Therefore, separate regressions were done on the positive and negative motion data (see Table 2):

Motion	Negative Reg.	Positive Reg.
Y (Forward/Back)	0.7111	0.8404
X (Right/Left)	0.6946	0.7398
Theta (Counterclockwise/Clockwise)	0.8987	0.9249

Table 2: Results from linear regressions on positive and negative sections of calibration data.

For the forward and side motions the drifts were small (averages were all less than 5° or 2 cm in magnitude). Also, the linear regressions for each were all less than 0.1, so they showed no trend. The turns actually caused large negative x displacements that increased with the turn's magnitude. Also, the right turn caused a positive x (right) displacement, and the left turn caused a negative x (left) displacement (see Figure 5):

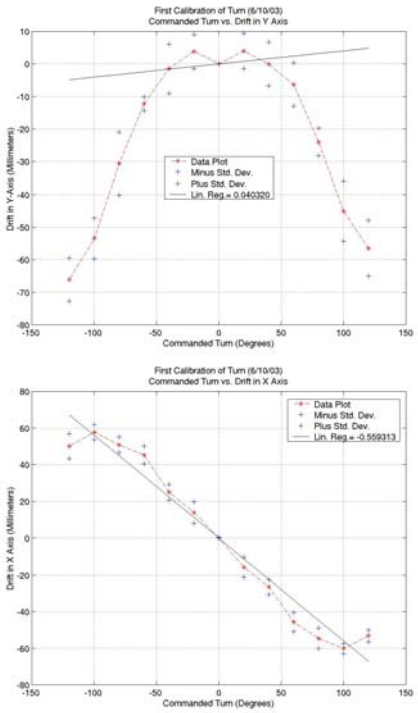

Figure 5: Graphs of x and y displacement during turn calibration.

These results show that the dog's turns, which were supposed to be performed around the tip of the dog's head, were actually being performed at a point approximately 5.8 cm behind this. This was later confirmed visually.

Three kicks the paw, and right and left kicks were calibrated. The x, y, and θ displacements for 30 kick trials were obtained for each kick type. For the paw kick, as expected, the dog never turned more than 6° and never moved more than 2.5 cm forward, back, left, or right. For the left and right kicks the dog would move to the left and right on average 15-16 cm and 2-4 cm backwards. The dog also changed its angle by 50°-60° each time. This again was expected; the dog must turn about

its center to kick the ball to the side, which explains the left and right displacements.

4. DOG CONTROL AND MONITORING SYSTEM: SERVER

4.1 Purpose and Basic Design

The purpose of this system is to create an integrated monitoring and control system for the dogs that can be used for development and debugging purposes. The system is designed to have an intermediate system between the dogs and the client-- a server – for several reasons. First, the dog's networking system is such that only one person may connect to a TCP port at a time. Having to handle many connections would add a heavy load on the dog's systems. Second, the server can control and regulate access to information. Third, the server can also do some data processing. For instance sometimes more information will be sent out from the dog than is needed. The server can eliminate this unneeded data before sending it to the client. Lastly, for high-volume, error-flexible outputs such as video the server can take the data from the dog via TCP and then use UDP to multicast it out, which reduces the load on the server and network [10, pp. 469-470, 528].

The system has a Linux server that consists, of four servers, written in C++ and a Microsoft Windows client written in C# for the .NET framework. Each server is for one type of input or output. One is for clients to send commands for the dog to execute, one is for dog text outputs, another is for the dog's "blob vision" (see Section 2.4), and one is the camera server.

4.2 General Server Libraries

A key feature of any server is its ability to handle the myriad errors that can occur in a network. The most important errors occur during reading, writing, and connect. Reads and connects can potentially block forever if the end point cannot be reached, since they will just keep trying. For writing, if a write occurs on a closed socket in Unix a signal will be sent that will shutdown the program unless it is caught. In order to deal with these and other issues, a class of static wrapper functions for regular socket systems calls (bind, read, close, etc.) was created. These wrappers handle the important read, connect, and write problems with timeouts and signal handlers and also throw exceptions when these problems occur or if a system call returns an important error.

These functions were then used to create classes for sockets. A base class was created called Socket{}. This class lays out the basic interface implemented by all the later derived socket class types (TCP client socket, etc.), by declaring an important set of virtual functions. This interface consists of a constructor that creates the socket and, if required by the socket type (as for a TCP listening socket), binds it to a port and IP address. The destructor of the class closes the socket. The only other functions are read and write (the read function requires that a timeout be specified). Several types of socket classes are derived from the base socket class: TCP client, TCP listening, TCP server, and multicast server sockets. This interface makes any socket class

match the simple theoretical model of a socket: a connection to another computer that is created to send and receive information and then be destroyed. This is one of the guidelines for good object-oriented design [9, pp. 76-101].

4.3 General Server Functions and Control Flow

Although each server is made separately because each has different requirements, large parts of the control flow of each server are similar. Each server works on the "one thread per client" design paradigm (also, for some, one thread per dog), rather than a "one process per client" paradigm [10, pp. 752-754]. This is done because threads are faster to create, and sharing data between threads in global variables is easier than sharing data between processes by IPC. Each server also has the same main loop (see Figure 6).

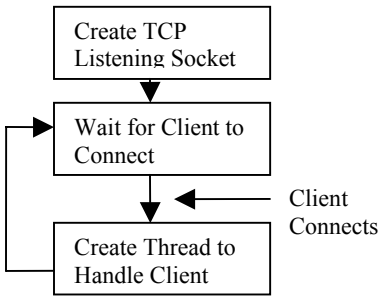

Figure 6: Flow chart for server main loop.

This loop is ubiquitous for all the servers, so it is a universal function. Then each server enters a per-client thread to handle that client's needs. The resulting control flow is shown in Figure 7:

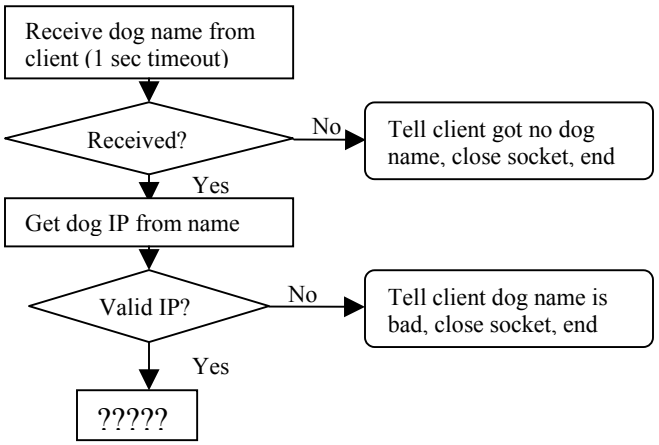

Figure 9: Flow chart of beginning of per-client thread, where the request dog domain name is retrieved and processed.

After this the servers act differently. Each server has to remember what dogs are being used, to know what to do with a new request for a dog. Three of the servers have to associate other information with any dog that is being used (the multicast port for that dog, the number of clients using it, etc.). So each server has a list of dogs that is implemented using the C++ Standard Template Library map container class. The map

stores these dog-specific objects, with each object mapped (stored and looked up) by the IP address of the dog [11, pp. 466-473].

The servers next look for the requested dog in the map. For the input server, if the dog is found then the client is told that the dog is being used, closes the socket, and ends. The output type servers allow for multiple clients to access the same dog, since this does not cause a problem. For these servers the flow of control is shown in Figure 8:

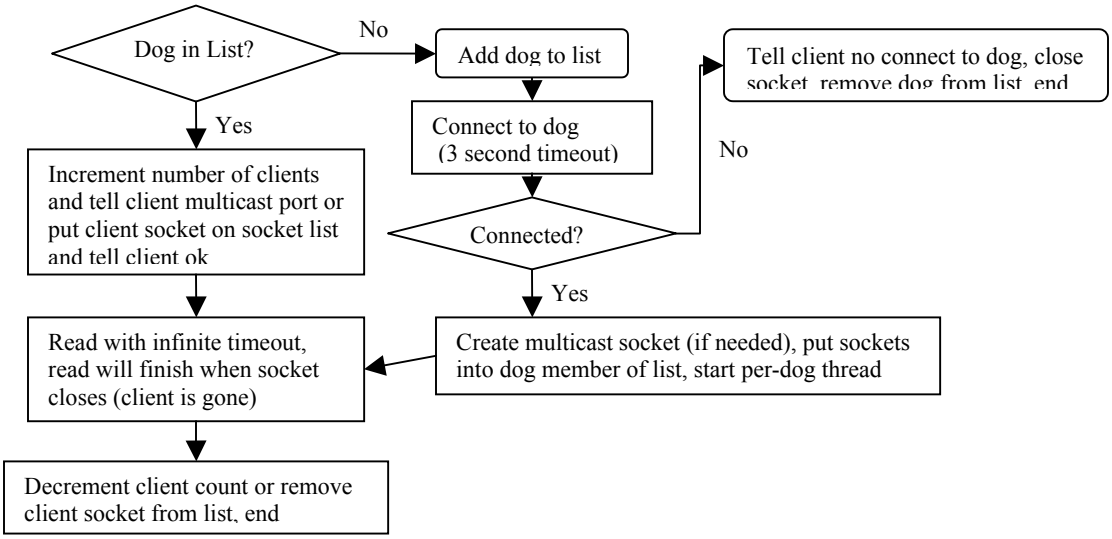

Figure 8: Flow chart of per-client thread (for output type servers).

The main difference here between servers is between the output server and blob and camera servers (see Section 4.2). Finally, the per-dog thread for these servers is shown in Figure 9:

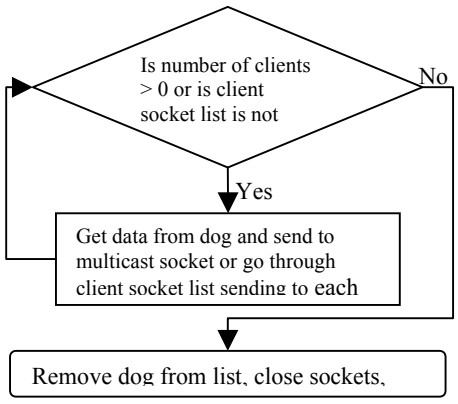

Figure 9: Flow chart of per-dog thread (for output type servers).

4.4 Server-Specific Details

4.4.1 Input and Output Servers

The input server does not need to have per-dog threads, since each dog can only be used by one client. After the dog is connected to the server loops infinitely, waiting for commands with a 5-minute timeout. If a read error or timeout occurs, the server assumes the client is done with the dog, closes the socket, removes the dog from the list, and ends the thread. The strings (commands) that the server gets from the client must be packaged into an OPEN-R packet (see Section 2.3) before being sent to the dog.

The output server's dog list contains objects with the TCP socket connected to the dog and another map of TCP sockets, each connected to a client for that dog. The output data from the dog are sent to each client one at a time by TCP sockets and not through a multicast, for reliability; the client cannot miss error messages from the dog, and multicast uses UDP, which does not have guaranteed delivery. But since a new message must be sent to each client one at a time, if a new message comes in before the last is sent, messages might be missed. Two assumptions mitigate this problem. First, it is assumed that the number of clients will be very low (around 5 per dog at most). Second, it is assumed that all clients will be on the same local Ethernet as the server. Thus there will be few clients to write to, and each write will be fast, so messages will not be missed.

4.4.2 Blob Server

This server works via multicast. Each object in the dog map has the TCP socket connected to the dog, the multicast socket, and the number of clients using the multicast. A data set is requested, and the dog then sends the server an OPEN-R package. After the total size and number of data comes a special header that specifies the size of the two-dimensional array that holds the blob information (the rows are integer values for the blob description structure, and the columns are each individual blobs). A two-dimensional array of this size is created, and the data are then read from the dog into this array. The important data for displaying blobs are extracted from the array (color, lower left corner, and upper right corner x and y) from each blob structure and are then sent to the multicast. The server then waits for 0.2 seconds before getting and sending the next set of blobs.

4.4.3 Camera Server

This server works almost identically to the blob server except that there is no special header. The dog sends the OPEN-R packet header (total size and number of data) and the compressed JPEG version of the last image taken by the camera. This file (array holding the JPEG) is so large (between 2000 and 10000 bytes) that it must be written to the multicast socket in pieces. Retrieving the JPEG from the dog and writing it to the multicast takes so long that no pause is done. In fact, the size of the JPEG completely controls the speed at which images are sent.

JPEG is an image compression format, and thus the size of the image is related to its complexity (simple pictures can be compressed more than complicated ones). Thus the more complex the image the dog sees, the longer it takes to receive and multicast, and the fewer frames per second (FPS) the camera achieved. A test was done to quantify the

relationship between FPS and JPEG size. The FPS at an instant was calculated as FPS = 1/(time of last receive and multicast). As the camera server transmitted, the dog's image was made more and less complex by putting a piece of white paper in front of the camera. The graph made using the 860 data points collected is shown in Figure 10:

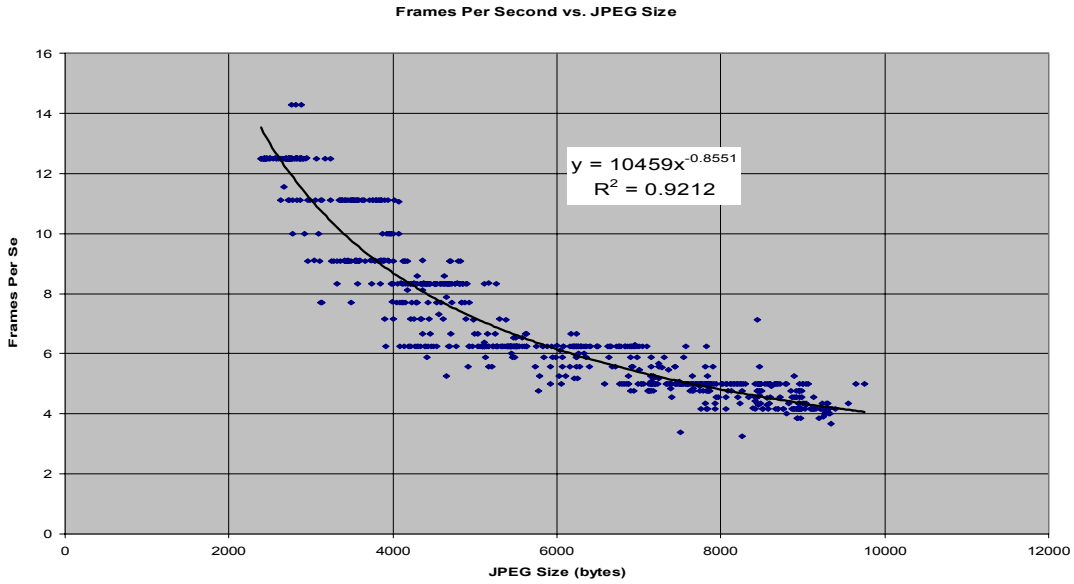

Figure 10: FPS vs. JPEG size for camera server

The relationship of this graph was hypothesized to be: FPS α 1/JPEG size. The regression of the graph supports this, since its estimated function is close to FPS=1/(JPEG size), and the R^2 is above 0.9. The graph also shows the limits of the FPS: the camera works between approximately 4 and 13 frames per second. Four FPS is not very fast, and certainly does not produce streaming video, but it is much better than the 1 to 2 FPS achieved in the previous dog camera system design.

5. DOG CONTROL AND MONITORING SYSTEM: CLIENT

5.1 Client Basic Design

Most of the design and features of the client program come directly as a consequence of the design and control flow of the server, but several important features and design concepts should be emphasized. As mentioned earlier, the client is a Microsoft Windows program written in C# (with Visual Studio .NET) for the new .NET framework. The client has a main window where the user can specify the dog and function (blob vision, etc.) desired, and a new window for this will be created. The client is multithreaded, spawning new threads for each output type service (e.g., a new thread is created to handle the continuous receiving and displaying of data from a blob observer, camera, or output server socket), so the main thread is not over-burdened. When exceptions are caught or error messages are sent from the server, a message box showing that error is displayed, and the window it came from is shut down (along with any threads). Screen shots of the main window and some other windows are in Appendix B.

The input window has a text box for writing commands and buttons to send commands to the server and clear the text box. The input window can save and load scripts stored in .txt files. The output window has a read-only text box that shows the output text from the dog. It can also save the output of the dog to a .txt file.

The blob observer, camera, and remote all have user controls created to handle these particular graphics displays. Each one uses the .NET framework's System.Drawing library and its Graphics class. For the blob observer, for each blob a FillRectangle and a Rectangle (for black outline) are drawn in the user control window (see Section 2.4). For the camera observer, the JPEG data are read into an array, which is read into a Stream object, which is read into an Image object, which is then displayed. The network transmissions and these steps account for the almost one-second delay between the time an image is captured by the camera and when it shows up on the client window.

5.2 Remote Control Client

The fifth client window is the remote control (see Appendix B). This program connects to both the input server and camera server, allowing the user to navigate the dog around a room. Certain keys are bound to sending specific commands to the dog (using callbacks). Thus there are keys to make the dog go forward, backward, right, and left, and to turn right and left. Also, if the user clicks on a certain point in the camera image, the program takes the position of that click and computes what command it should send to the dog to have it point the camera in that direction.

To compute this value requires understanding how the dog is commanded to move its head. A command called PointHead has three parameters: the x, y and z (in mm) coordinates (body's coordinate system) of the object that the dog should try to look at. For the remote, the dog is commanded to look at an object 1 meter away at the azimuth and elevation where the screen was clicked. Calculating the point head parameters requires two steps. First, the x, y, and z coordinates of the camera vector are calculated for the dog's head's coordinate system. Second, these coordinates are transformed into the dog's body's coordinate system.

The distance in u and v (pixels) from the center point of the camera to the click point are found. The focal length of the camera is 161 pixels. Thus a vector from the camera center to the click point is created (see Figure 11). The azimuth (Φ) and elevation (Θ) of this vector are calculated from u, v, and the focal length.

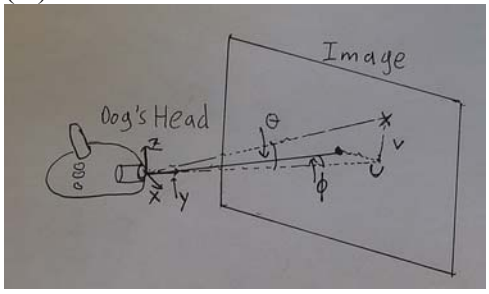

Figure 11: "Click" vector from camera to image diagram

$$\Phi = \text{ArcTan}(u/161)$$
$$\Theta = \text{ArcTan}(v/(u^2 + 161^2)^{1/2})$$

Another vector in this same direction, only 1 meter in length, is the vector of the new point where the dog should look. The azimuth and elevation are used to find the x, y, and z coordinates of the end of this vector in the head's coordinate system.

$$Xhead = 1000*\text{Cos}(\Theta)*\text{Sin}(\Phi)$$
$$Yhead = 1000*\text{Cos}(\Theta)*\text{Cos}(\Phi)$$
$$Zhead = 1000*\text{Sin}(\Theta)$$

The head is at a certain azimuth and elevation relative to the dog's body coordinate system. This azimuth and elevation can easily be determined by knowing the x, y, and z coordinates in the body's coordinate system of the last PointHead command. These angles are used to translate the new PointHead coordinates from the head's coordinate system to the body's coordinate system.

```
Xbody = Xhead*Cos(Az.) + Yhead*Sin(Az.)*Cos(El.) - Xhead*Sin(Az.)*Sin(El.)
Ybody = -Xhead*Sin(Az.) + Yhead*Cos(Az.)*Cos(El.) - Zhead*Cos(Az.)*Sin(El.)
Zbody = Yhead*Sin(El.) + Zhead*Cos(El.)
```

The only factor unaccounted for is the pitch (dog twists its head). The previously stated algorithm works on the assumption that in order to look at a point the dog turns and raises or lowers its head, but does not twist it. This is only true for small azimuths and elevations. For large angles the dog's head twists and the algorithm breaks down. Two measures correct this: the angles to which the dog can be commanded to move its head can are limited, and if a problem occurs the "look front" button can be pressed, which will reset the dog to look forward.

6. DISCUSSIONS AND CONCLUSIONS

The motion calibration system developed here may not be precise enough and easy enough to use for the required needs. The system worked well only for short-range, repetitive motions such as walks and kicks. One problem is that the NOB itself does not have a large range. This could be fixed by adding an extended range transmitter, but then there is the problem of wires. Even if a large area can be covered the dogs cannot be allowed to move anyway they wish, because they will become tangled in the cords. Wireless sensors can be obtained, but then the magnetic interference from the dog's motors will probably interfere with the transmissions.

The motion calibration system does do a good job at discovering major trends in the dog's movements, such as discovering that the dog did not turn precisely around the end of its head. It can also tell if the calibration factor for a motion is greatly off, but between natural inconsistencies in movement, uncertainty, interference from the dog's

motors, and the somewhat crude system (using the ruler) to counteract this, precise results about the dog's motions are probably not obtainable.

The server/client system does offer some valuable tools. This system is a good basic platform for debugging the Aibos. The existing client already has many features that make it a valuable for system debugging, and these tools can easily be added to.

7. FUTURE WORK AND RECOMMENDATIONS

The motion calibration system has produced some good results, but has not reached its full potential and may never do so. Given the facts of limited system range and motor interference, the NOB does not seem to be the best choice for a positioning system. Given the time and effort required to set up the extended range transmitter and perhaps get wireless sensors, it may be better to try another positioning system. It is also now clear though, that there are great benefits to having a positional system especially if it is integrated with the server system.

There is clearly great potential in the server system. The infrastructure and libraries are there to easily add servers for other types of sensory outputs (such as sound or positional information from some external device). There is also great potential for creating more complex and useful clients (such as the remote control) that interact with multiple servers. Examples include a better color mapping system that could interact with the input, camera, and blob servers to allow for easier creation of color maps, and creation of a program for teaching the dogs where they are on the field that would interact with the positional, input, and output servers.

8. ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Daniel Lee of the University of Pennsylvania, for all of his assistance, expertise, and patience. I would also like to thank Paul Vernaza and David Cohen for their help with the Aibos, and Janice Fisher for her assistance with this paper. Finally, I would like to thank the Microsoft Corporation for their generous support of my project and the SUNFEST program.

9. REFERENCES

1. Sony, *Sony Four Legged Robot Football League Rule Book*, Sony Corporation, 2003.

2. Sony, *OPEN-R SDK: Model Information for ERS-210*, Sony Corporation, 2002.

3. Sony, *OPEN-R SDK: Programmer's Guide*, Sony Corporation, 2002.

4. Sony, *OPEN-R SDK: OPEN-R Internet Protocol Version4*, Sony Corporation, 2002.

5. Dr. Daniel Lee, David Cohen, and Paul Vernaza, Autonomous Robot Soccer Team Implementation for Robocup 2003 Legged League*, Unpublished,* 2003.

6. Manuela Veloso et al., CMPack-02: CMU's Legged Robot Soccer Team, *Unpublished*, 2002.

7. Ascension Technology, *Flock of Birds: Technical Description of DC Magnetic Trackers*, Ascension Technology Corporation, Burlington, VT, 2002.

8. Ascension Technology, *Flock of Birds: Installation and Operation Guide*, Ascension Technology Corporation, Burlington, VT, 2002.

9. Cay S. Horstmann, *Mastering Object-Oriented Design in C++*, John Wiley & Sons, Inc., New York, NY, 1995.

10. W. Richard Stevens, *Unix Network Programming*, Vol. 1, Prentice-Hall, Upper Saddle River, NJ, 1998.

11. Matthew H. Austern, *Generic Programming and the STL*, Addison, Wesley, and Longman, Inc., Reading, MA, 1999.

10. APPENDIX A: CALIBRATION GRAPHS

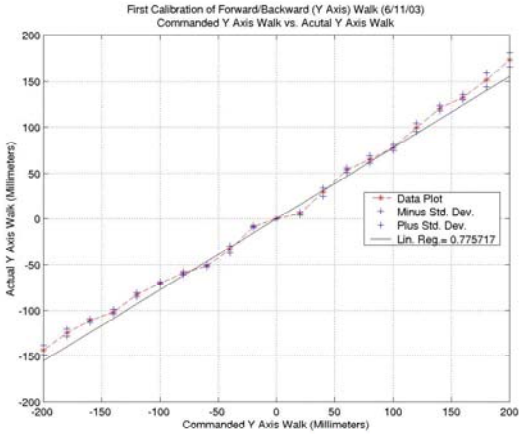

Figure 1: Forward motion calibration (single linear regression).

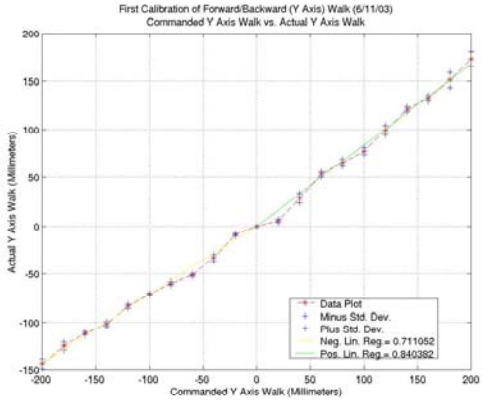

Figure 4: Forward motion calibration (positive and negative linear regressions).

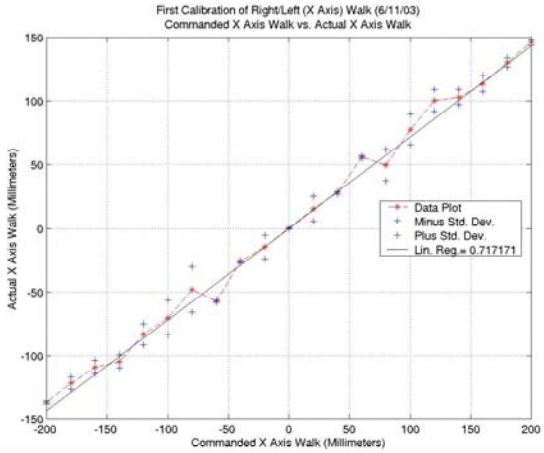

Figure 2: Side motion calibration (single linear regression).

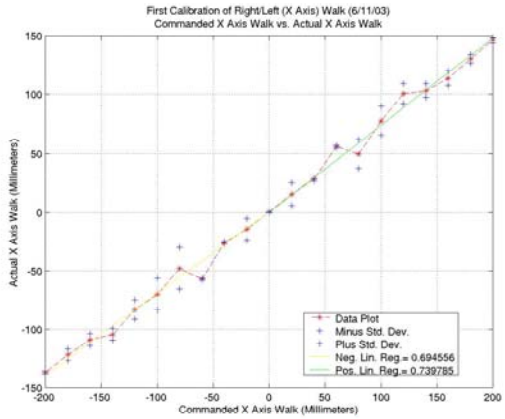

Figure 5: Side motion calibration (positive and negative linear regressions).

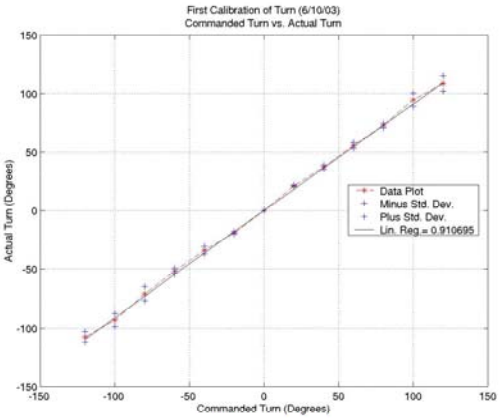

Figure 3: Turn calibration (single linear regression).

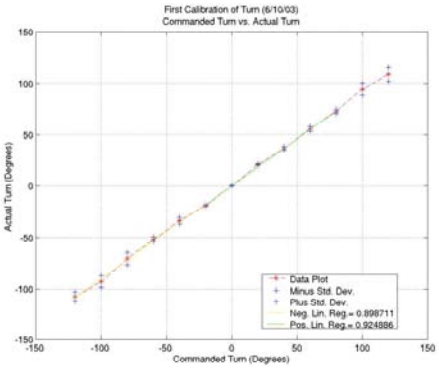

Figure 5: Side motion calibration (positive and negative linear regressions).

11. APPENDIX B: CLIENT PROGRAM SCREEN SHOTS

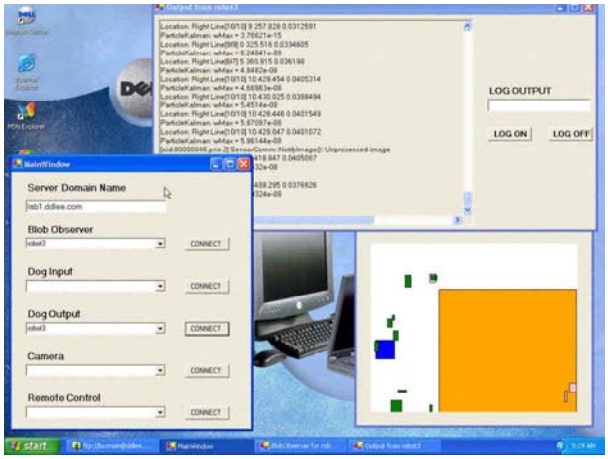

Figure 1: Picture of main window and sub-windows for client program.

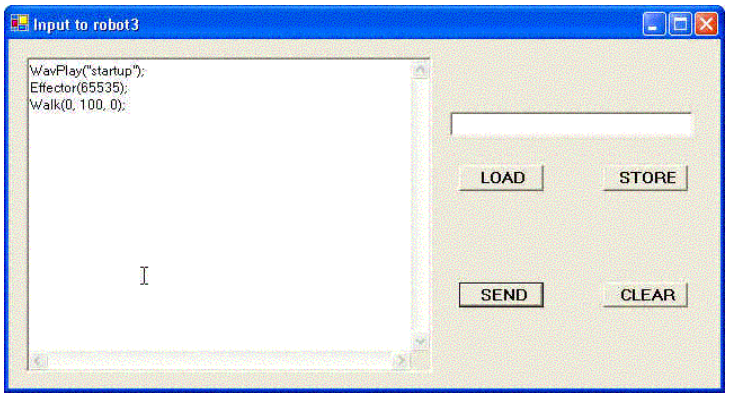

Figure 2: Window of input client program.

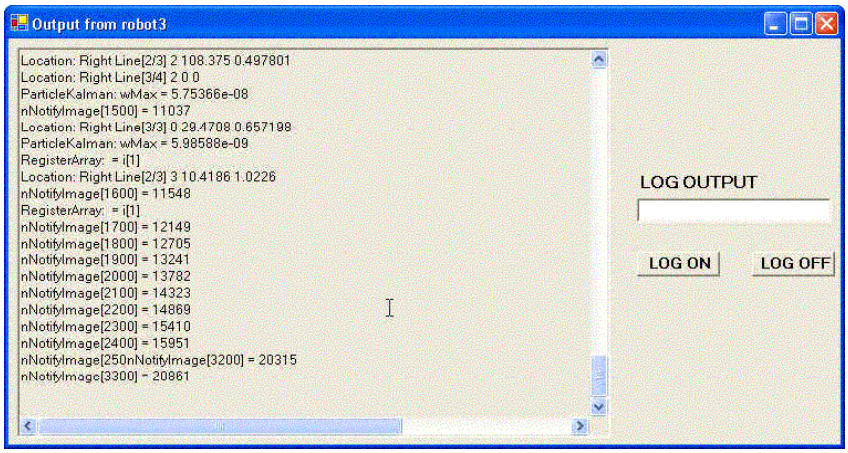

Figure 3: Window of output client program.

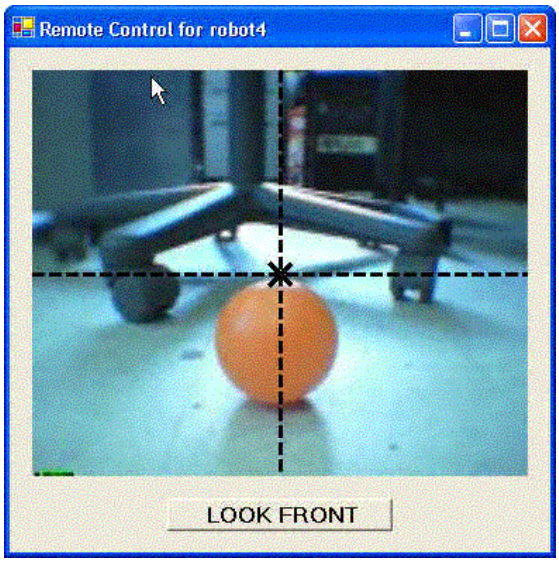

Figure 4: Window of remote control client program.