2009

Robot assisted herding

Pinky Thakkar
San Jose State University

ROBOT ASSISTED HERDING

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Masters of Computer Engineering

by

Pinky Thakkar

December 2009

The Designated Thesis Committee Approves the Thesis Titled

ROBOT ASSISTED HERDING

by

Pinky Thakkar

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

November 2009

Dr. Leonard P. Wesley	Department of Computer Engineering
Dr. Haluk Ozemek	Department of Computer Engineering
Dr. Winncy Du	Department of Mechanical Engineering

ABSTRACT

ROBOT ASSISTED HERDING

by Pinky Thakkar

This thesis addresses issues relating to the development of a robotic capability to assist a human in performing a complex task. It describes research and experiments involving a robot that assists a human to herd an animal. The novel focus of this work is that the robot is able to perceive the human's intentions by interpreting the human's movements without any explicit communication from the human. Furthermore, the robot is able to detect if the human is absent or unable to herd, and in that case, it herds the animal autonomously. A herding framework is developed based on low-stress herding techniques that enable the robot to start and stop herding, herd the animal forward, and turn the animal. Experiments were conducted to demonstrate the autonomous and assisted herding behavior of the robot. A conclusion is presented showing promising results that validate the approach for designing a robot with assisting and herding capabilities.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Leonard P. Wesley, for his continued guidance and understanding during this endeavor. I also appreciate the time and effort of the other members of my thesis examination committee, Dr. Haluk Ozemek and Dr. Winncy Du. I would also like to thank Mr. Jagdish Thakkar and his staff for granting me permission to use his farm and for teaching me techniques on how to herd animals. Finally, I would like to thank Dr. Burt Smith for his guidance.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Are We Ready to Have Robots in Our Lives?

Having autonomous mobile robots in our daily lives to help with everyday mundane tasks is no longer a novel concept. In the last few years, several commercial robots such as *Aibo*, *Nuvo*, and *Robosapien* have become available (Sony Corporation, n.d.; ZMP, Inc., n.d.; WowWee Robotics, n.d.). By January 2005, approximately 1.5 million Robosapiens were sold (Reichenbach, Bartneck, & Carpenter, 2006).

According to the survey conducted by Cerqui and Arras at the Robotics Pavilion at the Swiss National Exhibition Expo. 02, people no longer associate robots with a "wow" factor, nor do they perceive robots as "bad." Seventy one percent of people who participated in the survey had a neutral opinion about robots, believing that robots could be used for good or bad applications; 69% thought that robots could contribute something to our personal lives; and only 2% thought that robots were a "bad thing," because they perceived robots as replacing humans (Cerqui & Arras, 2003).

The United Nations Economic Commission for Europe (UNECE), the International Federation of Robotics (IFR), and the United Nations (UN) all forecast that demand for service and personal robots will increase within the next few years. Given this growth prediction, it is to be expected that new generations of robots will be designed for applications in services, domestic, and entertainment environments (Cerqui & Arras, 2003).

The work presented in this thesis describes the development of a personal robot that helps in herding a cow. Today, dogs are commonly used in herding, particularly for rounding up errant cattle. However, according to Burt Smith (1998), dogs need considerable training and careful watching before they are able to herd cattle successfully using low-stress herding techniques. Humans are not natural herders. Like dogs, they also need extensive training before they can successfully herd cattle using low-stress herding techniques. The advantage of using a robot to herd, or to assist in herding, is that if a robot is programmed to herd an animal using low-stress herding techniques, then the robot will always herd the animal using this technique. Another advantage of using a robot is that it does not suffer from mood swings, does not find a task mundane, and can function until it wears down.

In addition to exploring the use of robots in low-stress herding environments, this thesis also explores situations where there is limited communication between the robot and the herder, and almost no communication between the robot and the animal. This allows us to demonstrate that there may not always be a need to model extensive communication between the robot and the herder. Instead, herding can be effectively accomplished by the robot if it carefully observes the actions taken by the herder and plans its behavior depending on these actions.

Scientific Contribution

The goals of this research are two-fold: to develop an autonomous mobile robot that is able to obtain human instructions and intentions by perceiving and interpreting the

movements of a human; and to carry out autonomous herding at the direction of, or in the absence of, the human. An innovation of the work described here involves developing cattle and human behavior models, as well as perception techniques, which will be used along with a specific herding technique, called low-stress herding (Smith, 1998), to carry out assisted or autonomous herding activities. Another innovation is that no special training is required by the human to use the robot. The robot observes the human and understands the intentions based on the movement of the human.

Some researchers have attempted to communicate with assistive robots by developing speech and gesture recognition systems to convey their intentions to the robot (Topp, Kragic, Jensfelt & Christensen, 2004; Yoshizaki, Kuno & Nakamura, 2001). One differentiating aspect of our work is that no overt speech or gesture communication between the human and the robot is used. Instead, human motion relative to the animal and robot is used to communicate intentions and instructions to the robot. Although this work involves herding a single cattle-type animal, and the human is modeled by a remotely controlled toy, it represents an important incremental step towards developing a scaled-up approach that can handle multiple animals and real humans.

Autonomous Mobile Robot

Jones, Seiger, and Flynn (1999) define an autonomous mobile robot as "an intelligent connection of perception to action." Such robots are usually composed of a microprocessor and sensors, and have the ability to move from one location to another. Siegwart and Nourbakhsh (2004) state that a mobile robot needs to address three key

questions – where is it? (i.e., its current location); where does it have to go?; and how will it get there? In order to answer these questions, a robot has to have a model of the current environment and an ability to perceive and analyze the environment (Perception) so that it is able to find its co-ordinates within the environment (Localizaton) and also can plan and execute its movement to go to the next location (Planning and Motion Generation).

The advantages of autonomous mobile robots are many, and include movement, speed (modern robots respond to 1 kHz, whereas humans respond to 1 Hz, or 1,000 times slower), accuracy and precision, duration of effort (robots can work steadily until they fail in some manner, while humans have an attention span of 30 minutes under observation and need breaks), robustness, and dollar cost.

However, there are two main challenges with autonomous mobile robots, namely recovery and safety (Sheridan, 2002). According to Steinfeld (2004), failure in autonomous robots will occur since they will come across conditions which they have not been programmed to understand and hence, will not perform in an acceptable or desired manner. Because it is not possible to take into consideration all the potential scenarios that a robot will encounter, it is inevitable that there will always be failures. When developing an autonomous mobile robot, it is important to design the robot so that when the failure occurs, it falls in a safe and recoverable state. In this state, the robot should not be permitted to harm humans, animals, property, and itself.

Human-Robot Interaction

In the field of human-robot interaction, humans and robots share an environment either to pursue their individual goals or to collaborate to achieve a common goal. When a human interacts with another human, social protocol dictates the behavior of each person, e.g., when riding in an elevator people wait for his or her turn either to enter or exit the elevator. However, due to the lack of such a protocol between a human and a robot interaction, it is important to define rules that govern the behavior of both agents (Bruce, 2005).

Another important point to consider in human-robot interactions is that the agents often do not know the goals or intentions of the other agents in advance. The intentions are often perceived by either communication or by observing the actions of the other agents. Communication is a challenging issue because we must first determine what to communicate and when, and then perhaps provide an appropriate response. Also, it is important to understand that when gaining information through observation, time plays an important role (Bruce, 2005), e.g., observations taken at various times will help the robot determine the direction in which a herder is going.

In this thesis, individual actions for a herder and a robot are defined so that an animal can be herded in a certain direction. We focus on observation as the primary method for gaining information about the intentions of the other agent (human).

Challenges

The success of human-robot interactions largely depends on the appropriate

behavior between hardware and software technology and the behavior of the human. The introduction of robots into many industries has led to improved product quality, increased productivity, and the removal of people from hazardous or monotonous work. However, it has also proved to be hazardous for personnel unfamiliar with their movement flexibility and their speed of motion. Information available from France, Japan, Sweden and the United States, among others, seems to indicate a potential risk for all personnel interacting with robots. Hence, safety and reliability remain the most important issues in the field of autonomous mobile robots working side-by-side with humans (Rahimi & Karwowski, 1992).

Overview

The remaining chapters in this thesis explore in detail the development of assisted herding using a robot. In Chapter 2, we describe the related research work in the area of robot-animal interaction and assisted robotics. Chapter 3 outlines herding techniques, including low-stress herding. Chapter 4 describes the animal and the human model, and our approach towards implementing autonomous and assisted herding behaviors. Chapter 5 describes the methodology used for object recognition, localization, and tracking. It also explains how the animal is herded autonomously in different directions and how the behavior of the human is perceived and inferred by the robot. In Chapter 6, we demonstrate the experiments that were conducted to test our hypotheses and we discuss the results in detail. In Chapter 7, we conclude our discussion and suggest the areas for future work.

CHAPTER 2

Background

Related Work

Trafton et al. (2005) used person-to-person interaction to model the human-robot interaction. In their research, they studied the ability of people to take one another's perspective and reasoning when interacting and they designed a robotic system that was able to simulate and reason about the world from the perspective of the human with whom the robot was collaborating. Such a system facilitates effective communication between a human and a robot.

For example, consider a scenario where a robot is helping an astronaut in space. The robot's task is to give various tools to the astronaut, as instructed by the astronaut. The astronaut asks for a wrench. The robot sees two wrenches, but the astronaut can only see one. In this scenario, the "perspective-taking" robot will consider the perspective of the astronaut and will immediately deduce that since the astronaut can see only one wrench, he must be asking for the wrench that he sees. Hence, the robot is able to hand over the correct wrench to the astronaut.

Even though such a robotic system would be an invaluable tool in human-robot interactions, there are many challenges in developing a perspective-taking robotic system. The perspective-taking robot needs substantial computational power to represent the work from its own perspective and also the human's perspective. Moreover, the world is constantly changing and, therefore, the robot would have to constantly update

representation of the world from its and the human's point-of-view. Another major challenge is to determine how to represent perspective, knowledge, and information.

Vaughan et al. (2000) demonstrated an automated herding capability in both a simulated and real world context where a mobile robot fetched ducks and directed them to a predefined goal position. The experiment was conducted in an arena seven meters in diameter, where the ducks moved about freely. The setup consisted of a robotic vehicle that herded the ducks to the goal, an overhead video camera that viewed the entire arena, ehich enabled the system to keep track of the movement of the robot and the ducks, and a workstation that took the video-feed from the video camera and calculated the trajectory of the robot. The algorithm used to calculate the trajectory was based on three forces:

1. The attraction of the robot towards the center of the flock with a magnitude that was proportional to the distance between the robot and the center of the flock.

2. A repulsion force between the robot and the flock with a magnitude proportional to the inverse square of the distance between the robot and the center of the flock, ensuring a safe distance between the robot and the flock.

3. A repulsion force between the robot and the goal position of a constant magnitude that ensured the robot's position remained behind the flock with respect to the goal.

Butler et al. (2004) developed a virtual fence algorithm to herd cows. Their approach used a smart collar equipped with a GPS unit, a PDA, wireless networking, and a sound amplifier. The virtual fence was a fenceless area defined by a polygon. The

GPS unit was used for monitoring the location of each cow within the virtual fence. As soon as a cow approached the fence, the animal was given a sound stimulus of the magnitude, inversely proportional to the distance between the animal and the fence. This ensured that the animal always stayed within the boundary of the virtual fence.

Despite the advances of previous work, some technical gaps remain. Ultimately, truly purposeful and autonomous robot-assisted herding must be accomplished within un-instrumented environments. Furthermore, it must be anticipated that herding in outdoor natural environments will require robots to carry out tasks in contexts that present varying degrees of perception and terrain challenges. Dust, precipitation, and noise can impair perceptions. Terrain elevation changes and obstacles can preclude line-of-sight perception and movement along optimal paths. Being able to carry out assisted herding tasks when communication is limited or absent, presents challenges that have not been completely addressed, to date. Indeed, the work described here does not completely address these herding challenges. However, it does begin to bridge some of the technical gaps that remain between successful previous work and a truly autonomous assisted herding capability. This is accomplished by significantly limiting the amount of "instrumentation" required to control the movement of the animal, as well as limiting the communication between human and robot.

CHAPTER 3

Herding

Herding Techniques

Humans have been domesticating animals for centuries. Domestication has brought prosperity to society and has been one of the cornerstones for human progress, trade, and commerce. One of the most critical aspects in domesticating animals has been the actual upkeep of the stock. Herding animals is a very important part of the domestication process, whether it is for point-to-point transport of livestock, grazing, commerce, or predator avoidance.

There are four unique ways of moving an animal from one location to another (Smith, 1998). They are described below in order of increasing stress to the animal and increasing cost per weight: leading, herding, driving, and physically transporting.

1. Leading behavior causes the least amount of stress in animals. Animals often follow a leader when suitably motivated (better forage, shelter, etc.). The leader's position is in the front of the herd and at the edge of their flight zone, up to a maximum of around 100 feet.
2. Herding is a fear or pressure-based method, wherein the herder is a predator or a super dominant, exhibiting pressure on the herd, and causing it to move in a certain direction. Herding can cause a large amount of stress in animals if done incorrectly and, in some cases, herding may border on driving.
3. Driving animals is one of the oldest forms of moving animals. It consists of scaring an animal in the direction you want it to go, by positioning yourself in

such a way that the animal is between you and where you want it to go.

4. Physically transporting animals over long distances is becoming increasingly common due to the strides made in technology, veterinary medicine, and transport methods. One of the biggest challenges in physical transport, besides the cost, is to shrink the impact of the animals' experiences due to stress and lack of sufficient food, and also to reduce the time that it takes for the animal to get back to its original weight (compensatory gain).

There are three distinctive zones, which a prey animal responds to, when encroached upon by what it believes to be a predator: recognition zone, flight zone, and fight zone (Smith, 1998).

1. The recognition zone is the outer extreme of the animal's sensory perception, and can either be vision or smell. An animal will notice an entity entering this space and at this point it begins to evaluate whether the entity is harmless or a predator.
2. The flight zone is the ring inside the recognition zone, which if penetrated, has a high probability of making the animal move. The term was introduced in 1934 by Heini Hediger, a curator at the Berlin Zoo (Smith, 1998, p. 115). The flight zone is typically oval in shape, as shown in Figure 1.
3. The fight zone is the innermost ring and is closest to the animal. When a super dominant or predator enters this zone, the animal may decide to fight or flee, depending on a host of conditions, such as the number of intruders, its health, physical and emotional condition, the size of the intruder, and many other external factors. The size of the fight zone depends on a myriad of factors, including, but

not limited to, the animal's health, the stress it is under, insects, its breed, its previous experiences in being handled, the size of the herder, the weather, obstructions between the animal and the herder, whether the animal is feeding or drinking, eye contact, and if the animal is alone or in a group.

Low-Stress Herding

Traditionally, herding was accomplished by the brute force approach of inducing fear. This caused significant stress in the animals. Stress causes reduction in weight gain, meat quality, milk production, reproduction performance, and immunity from diseases. In the work reported here, herding is accomplished by using low-stress herding technique where the flight zone of an animal is used to control and maneuver the animal in the desired direction. The key here is that the animal is herded from the boundary of its flight zone, which causes anxiety, not fear, in the animal. Thus, the animal perceives the human as its protector, and not its adversary (Elsasser, 2002; Fanatico, 1999; Grandin, 1999).

The other advantages of low-stress herding techniques are that they facilitate improved performance with respect to weight gain, feed efficiency, reproduction performance and immunity (Siegwart & Nourbakhsh, 2004; Sheridan, 2002; Steinfeld, 2004; Rahimi& Karwowski, 1992; Trafton et al., 2005).

The concept of low-stress herding revolves around keeping the herder at the boundary of the flight zone. Penetration into the flight zone leads to higher fear in the animal being herded and the stress is a direct function of the depth of penetration into the

flight zone with the speed of penetration.

Definitions of the some of the key-terms used in low-stress herding methodology are:

Herding: Controlling and maneuvering the animal in the desired direction.

Flight Zone: The flight zone is also known as the flight distance. This is the area around the animal, which when intruded by a perceived predator, causes anxiety in the animal. The response to anxiety is to move away, but if this area is deeply penetrated, the animal's response is to panic and flee. For an animal, the flight zone is an egg-shaped space, as shown in Figure 1. Its size is dynamic and changes depending on several factors, such as the behavior of the herder, weather conditions, and the state of excitement of the animal. The animal's flight zone plays a key role in low-stress herding techniques (Smith, 1998; Trafton et al., 2005).

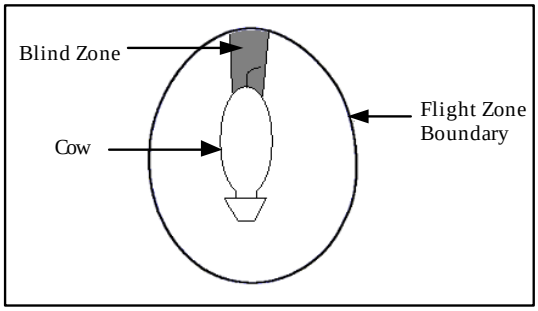


Figure 1. Flight zone.

Animal Vision: Cattle have wide-angle vision and can see behind themselves without turning their heads. However, they have a small blind spot of about 15-30 degrees

behind their rears, as shown in Figure 1. The handler avoids staying in the blind-spot area for too long as it can lead to panic in the animal (Smith, 1998).

Outrider Position: This is “the position the herder takes on or adjacent to the boundary of the flight zone” (Smith, 1998).

CHAPTER 4

Approach

Animal Model

Richard Vaughan's model attempts to capture flocking behavior of real ducks (Vaughan et al., 2000). The ducks are represented by a circle and have a heading. The movement of the ducks is modeled by the following forces: (a) The attraction force of the ducks towards each other, (b) A repulsion force of the ducks towards each other to avoid collision, (c) A repulsion force of ducks to the wall of the arena to avoid collision, (d) A repulsion force of the ducks to the robot. The ducks do not have sideways or backwards movement.

Our research consists of just one animal, cattle, and is represented by a remote controlled toy-car with an orange colored-stick attached to it. The movement of the animal is governed by the penetration of the robot in the flight zone, as described in Chapter 4.

The animal model described here consists of the following properties: (a) The flight distance is the distance between the animal and the edge of the flight-zone, (b) The herding angle is the 45 degree angle behind the animal, (c) The maximum herding angle is the 60 degree angle behind the animal, (d) The minimum herding angle is the 30 degree angle behind the animal, (e) The blind zone is the angle behind the rear of the animal, which the robot enters when it has to turn the animal.

These properties are illustrated in Figure 2.

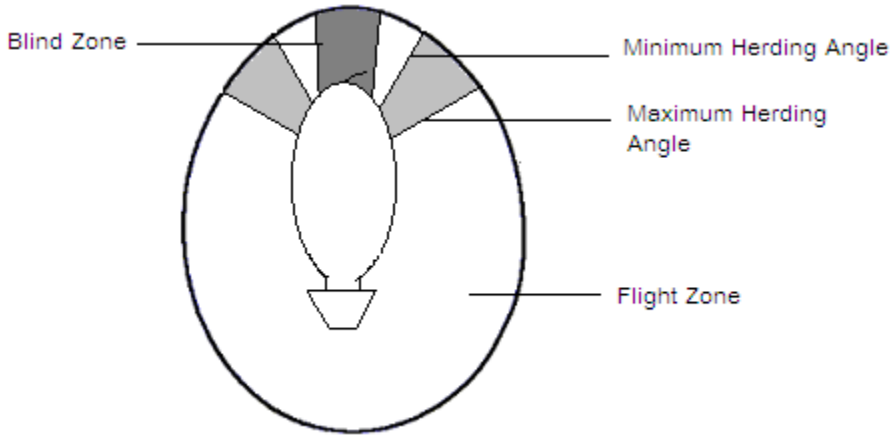


Figure 2. Properties of the animal model.

We define the following three temporal movements of the animal relative to the robot:

1. Moving forward: The animal moves in the same direction for a specified time.
2. Turning: In practice, the animal never moves in a zigzag motion. Hence, we define turning as the movement of the animal that is not in the forward direction.
3. Stopped: When the animal is neither moving forward nor turning, the robot perceives that the animal has stopped moving.

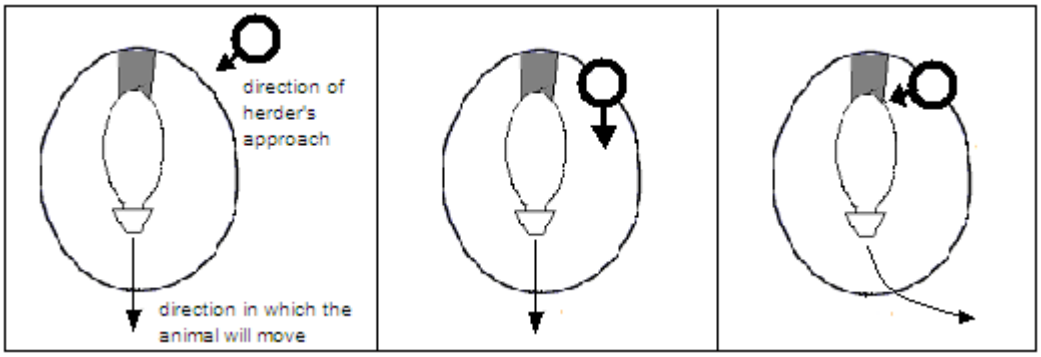
The animal model described here is generic enough to be applied across species that can be herded using low stress herding techniques.

Autonomous Herding

This section describes the approach used for autonomous herding.

Finding the flight zone. The flight distance is a predefined value that is not known to the robot. In low-stress herding techniques, the animal is always herded from

the edge of the flight zone and within the maximum and minimum herding angles, unless it has to either turn or stop the animal. Hence, the first step toward herding is to determine the flight zone of the animal. When herding multiple animals, the flight zone is determined by a zigzag motion. However, when herding an individual animal, the flight zone is determined by making a slow diagonal approach towards the rear of the animal, as shown in Figure 3(a). When the animal moves forward, the herder has reached the edge of the flight zone. Other approaches to determining the flight zone might be equally or more effective, such as the shoulder approach. However, because of the geometry of our animal model, we are employing the diagonal approach.



(a) Initial motion (b) Herding forward (c) Turning the animal

Figure 3. Low-stress herding techniques.

Herding forward. To move the animal in the forward direction, the robot keeps penetrating the flight zone very slowly. In order to get the robot out of its flight zone, the animal responds by moving forward.

Turning the animal. To turn the animal, the robot employs the tail-head approach,

in which the robot turns towards the rear of the animal and slowly moves towards it, as shown in Figure 3(c). As soon as the robot reaches the blind zone area, the animal turns its head to look at the robot and at the same time moves forward. This causes the animal to turn. The robot continues to approach the animal until the animal has turned the desired amount.

Stopping the animal. To stop the animal, the robot stops following it. The animal continues its motion until the herder is out of the flight zone, at which point, the animal stops and turns to look at the herder. When it finds that the herder is not moving in its direction, it relaxes and gets engaged into other activity, such as grazing.

Human Model

The human is represented by a remote controlled toy-car with a pink colored-stick attached to it. In addition to a color tag of pink, it consists of three temporal human movements relative to the animal and the robot. When the human moves towards the robot, the robot will either initiate herding or turn the animal. When the human moves away from the animal, the human intention is for the robot to turn the animal. The human will stop moving when the animal has been herded to goal location.

Assisted Herding

This section describes our approach for assisted herding.

Initiate herding. The robot and the human are always on either side of the animal. When the robot perceives the human moving towards the animal, it interprets this as a

command to initiate herding. The robot initiates herding by first determining the flight zone and then herding the animal forward, as described in Chapter 4, *Autonomous Herding*. Once herding is initiated successful (i.e., the animal starts to move), the robot begins to herd the animal in the forward direction.

Turning the animal. To turn the animal, the human either moves towards or away from the animal, depending on the direction in which the animal is to be turned. If the human wants to turn the animal, then the human moves towards the animal, but if the human wants the robot to turn the animal, the human will move away from the animal to let the robot turn it. Hence, if the robot perceives that the human is moving towards the animal, then it interprets this as a command for it to move out of the flight zone so that the human can turn the animal. However, if the robot perceives that the human is moving away from the animal, then it interprets this as a command for it to turn the animal.

Stopping the animal. When the robot perceives that the human has stopped moving, it interprets this as a command for it to stop herding.

CHAPTER 5

Method and Implementation

This research uses an ActivMedia robot, an off-the-shelf mobile platform that contains basic components for sensing and navigation (ActivMedia Robotics, 2002; Mobile Robots, Inc., 2009). The robot is battery powered and has integrated sensors, managed by an onboard microcontroller and mobile-robot server software. The Pioneer robotic camera system is equipped with a laser range finder; bumper sensors; and a pan, tilt and zoom camera.

Object Identification, Localization, and Tracking

The following modules are developed to identify, locate, and track objects:

Object identifier. The vision system helps to identify the object. The camera is calibrated to find the maximum angle of the visual field. Two sticks are kept on either side of the robot so that when the image is taken, the sticks appear at either end in the image. With the help of the laser tracking system, the angle of each stick is measured. It is found that the angle made by each stick with respect to the robot is approximately 20 degrees.

The PTZ camera acquires images starting from -90 degrees and going to +90 degrees, in an arc of 15 degrees increment. It forwards these images to the vision module for processing. The vision module defines each object as a range of hue, saturation, luminosity (HSL) values. It was determined right at the start of the project that the design of the vision system would be simple, effective, and as fast as possible. Hence,

instead of using sophisticated object recognition mechanisms to identify the human and the animal, we use color tags and a simple algorithm. This ensures that time is spent on the main focus of the actual problem, and not on the vision system.

When the vision module receives an image, it converts the RGB value of each pixel to the HSL value and checks if it is within the specified HSL range of the object of interest. If this pixel is within the HSL range, then the next 10 consecutive pixels in the same row are also checked to see if they are within the HSL range. If they are, then it means that the object is within the image. If the object is not in the image, then the PTZ camera rotates by 15 degrees, acquires the next image, and forwards it to the vision module for processing. This process is repeated until the object is found in an image. If the object is not found in a 180-degree sweep, the PTZ camera makes a second sweep across a 180 degree arc to locate the object. If the second attempt also fails, then the robot stops trying to locate the object.

Object locator. Once the object is found in an image, an approximate angle of the object with respect to the robot is calculated using the following formula:

$$\theta = (\alpha + \beta / 2) - (\beta * \lambda / \delta) \quad (1)$$

where,

θ = Angle to the object with respect to the robot

α = Angle of camera at which the image is taken

β = Range of view of the camera

δ = Horizontal resolution in pixels of the acquired image

λ = Pixel position of the object within the acquired image

This is illustrated in Figures 4 and 5.

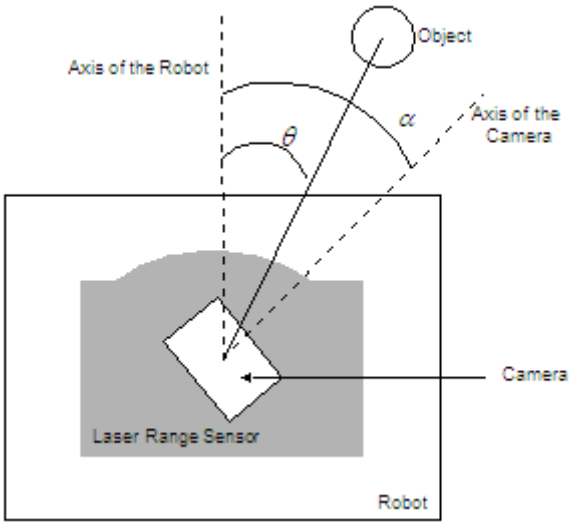


Figure 4. Top view of the robot.

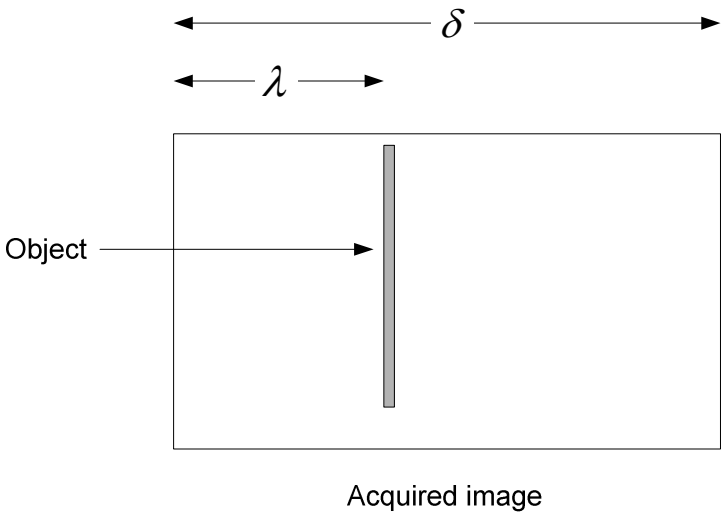


Figure 5. Location of the object within an image.

Next, the laser-range finder along with ARIA's ArSick C++ library is used to calculate the exact distance and the angle of the object in relation to the robot.

The laser range finder is directed at an area, $\theta + \phi$, and $\theta - \phi$, where ϕ is a small value and ARIA's getClosestObjectDistance() function is used to calculate the angle and the distance of a nearest object in the specified range, as shown in Equation 2.

Object tracker. The object is tracked using the SICK laser-range finder. Once the location of the object is determined, the readings are taken every few milliseconds using the following function call:

$$O(\theta', d') = getClosestObjectDistance(\theta - \phi, \theta + \phi) \quad (2)$$

where,

O is a vector representing the location of the object

θ' is the angle to the object at time t2

d' is the distance to the object at time t2

ϕ is a very small value

This is illustrated in Figures 6.

In order to make sure that we are tracking the same object, we compare consecutive readings from the laser-range sensor. If the difference is within a predetermined threshold, then it means that the robot is tracking the same object. However, if the difference is outside the threshold, then it means that the object being tracked is now lost. In such case, the robot initiates the *Object identifier* routine, followed by *Object locator* routine, and then the *Object tracker* routine.

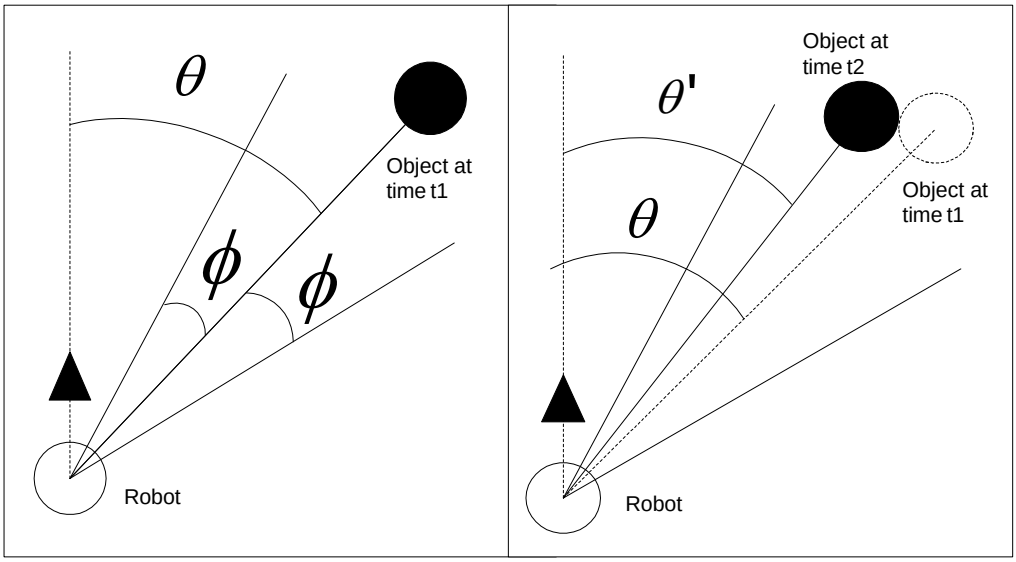


Figure 6. Object tracking.

Autonomous Herding Methodology

This section describes the algorithm used for autonomous low-stress herding techniques. It involves developing separate modules for initiating herding, moving the animal forward, turning the animal, and stopping the animal.

Initiate herding. This module helps to determine the flight zone. It is broken into two subroutines: Initial Position and Initiate Movement.

1. In the initial position subroutine, the robot attempts to position itself diagonally, between 30 and 60 degrees, to the rear of the animal, as shown by the shaded region in Figure 7. Figure 8 shows the flowchart that describes this subroutine.

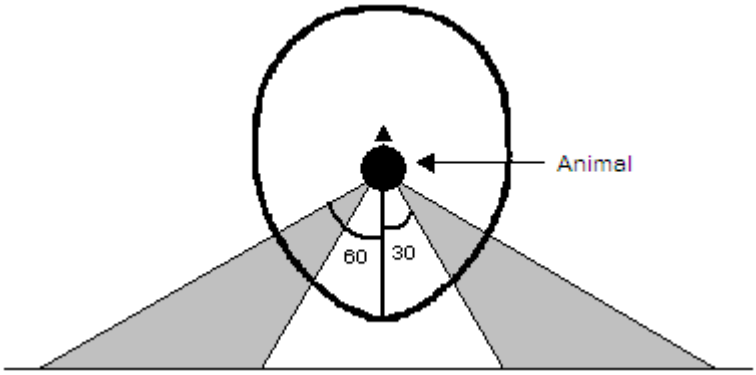


Figure 7. Initial herding position.

The robot scans the environment to locate the animal. With the help of a laser-ranging sensor, it measures the angle made by the animal with respect to the robot. If the angle is outside the range of |30| to |60| degrees, then the robot attempts to turn and move to the initial position as follows:

- If the animal is at an angle between 0 and -30 degrees, then the robot turns -90 degrees and moves backwards until the animal is at an angle 45 degrees with respect to the robot. For example, if the animal is at an angle -25 degrees, as shown in Figure 9(a), then the robot turns -90 degrees and moves backwards, Figure 9(b), until the animal is at an angle of 45 degrees, Figure 9(c).
- If the animal is at an angle between 0 and 30 degrees, then the robot turns 90 degrees and moves backwards until the animal is at an angle -45 degrees.

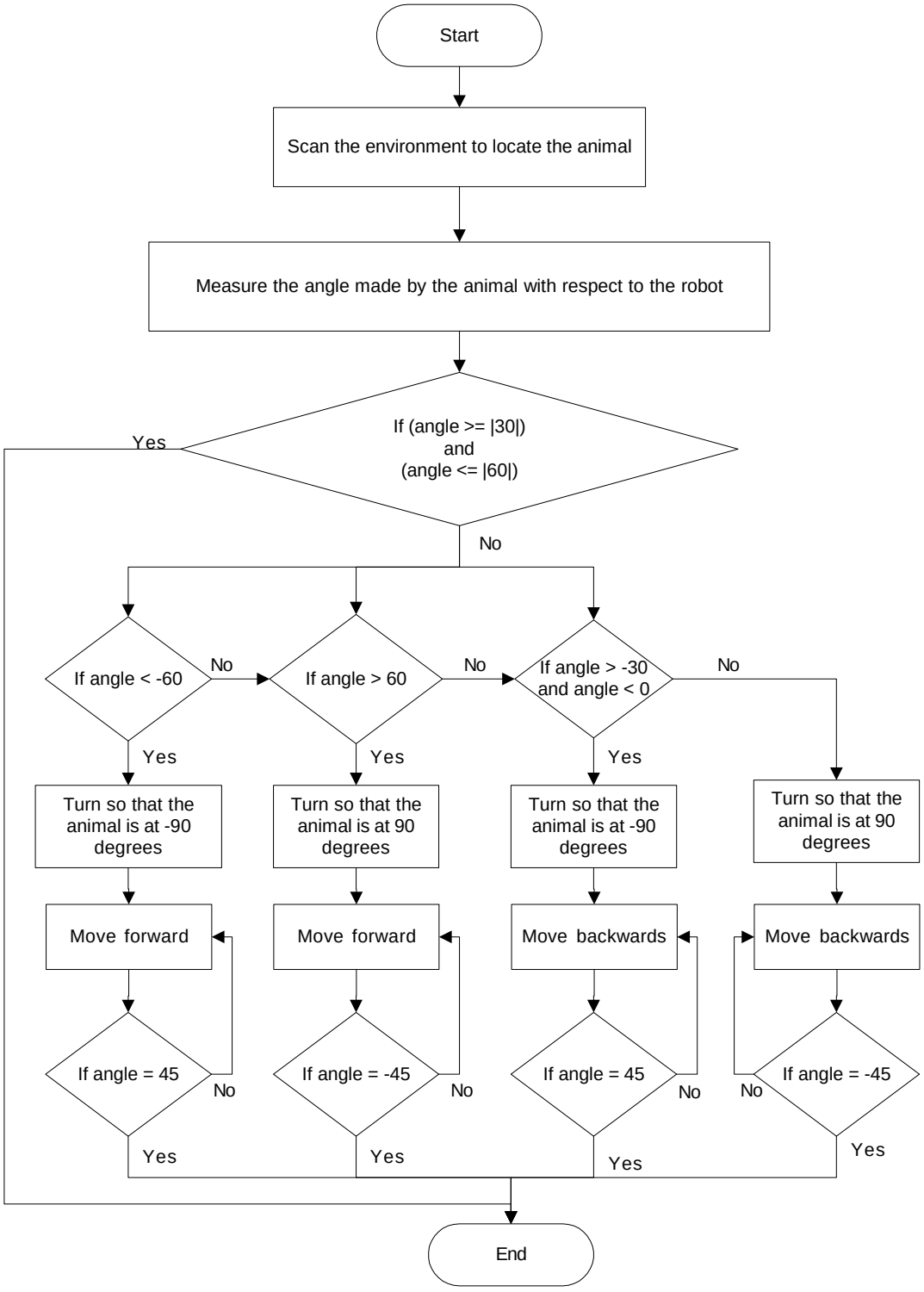
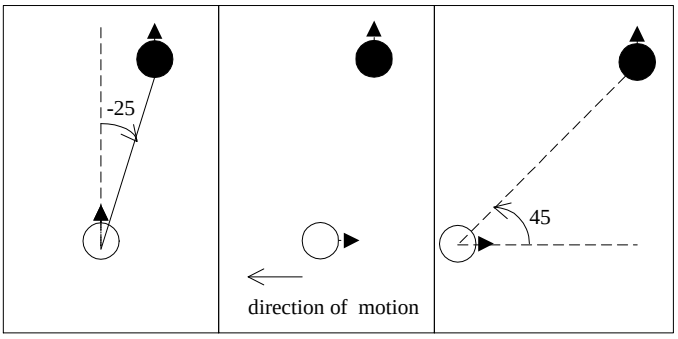


Figure 8. Flowchart describing initial herding position subroutine.



(a) (b) (c)

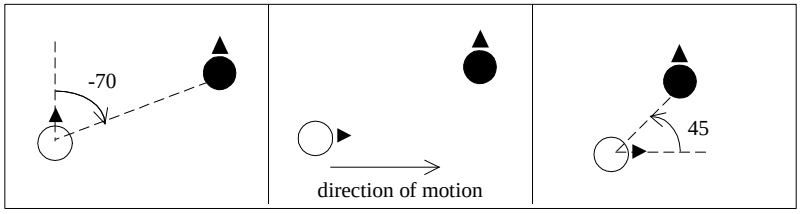
Figure 9. Moving to initial position when animal is at -25 degrees.

- If the animal is at an angle less than -60 degrees, then the robot turns -90 degrees and moves forward until the animal is at angle of 45 degrees. For example, if the animal is at an angle of -70 degrees, Figure 10(a), then the robot turns -90 degrees and moves forward, Figure 10(b), until the animal is at angle 45 degrees, Figure 10(c).

- If the animal is at an angle greater than 60 degrees, then the robot turns 90 degrees and moves forward until the animal is at angle -45 degrees.

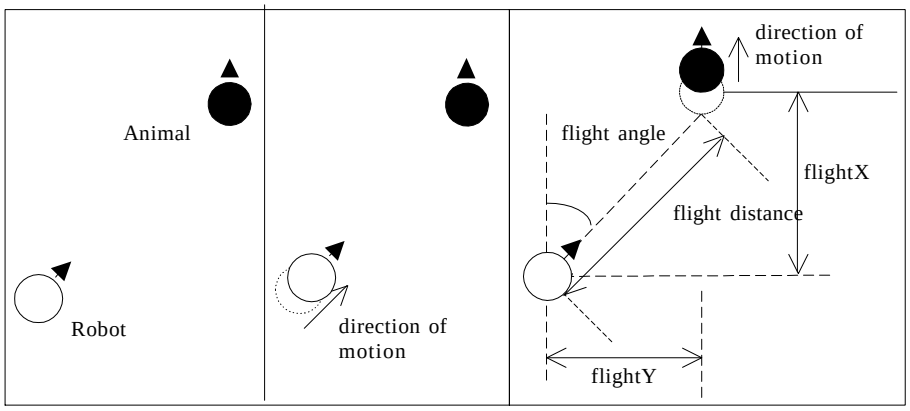
2. In the initiate movement subroutine, to make the animal move in the forward direction, the robot turns in the direction of the animal as shown in Figure 11(a), and slowly approaches it, until the animal moves forward, Figure 11(b). The robot notes the flight distance and the flight angle, and calculates the flightX and flightY distances, Figure 11(c).. The distance flightX is used to keep the robot on the edge of the flight zone. The distance flightY is used in the Herding Forward

routine to herd the animal in the forward direction. Figure 12 shows the flowchart that describes this subroutine.



(a) (b) (c)

Figure 10. Moving to initial position when animal is at -70 degrees.



(a) (b) (c)

Figure 11. Initiate movement.

Herding forward. To move the animal forward, the robot turns in the direction that the animal is facing, takes up the outrider's position, and slowly moves along with the animal, as shown in Figure 13. In Figure 13(a), the robot is outside the flight zone. When it moves forward, it reaches the edge of the flight zone of the animal, Figure 13(b).

As a result the animal moves forward, leaving the robot outside the area of the flight zone, Figure 13(c). The robot again moves forward to reach the edge of the flight zone, Figure 13(d). This process is repeated until the animal has moved the desired distance.

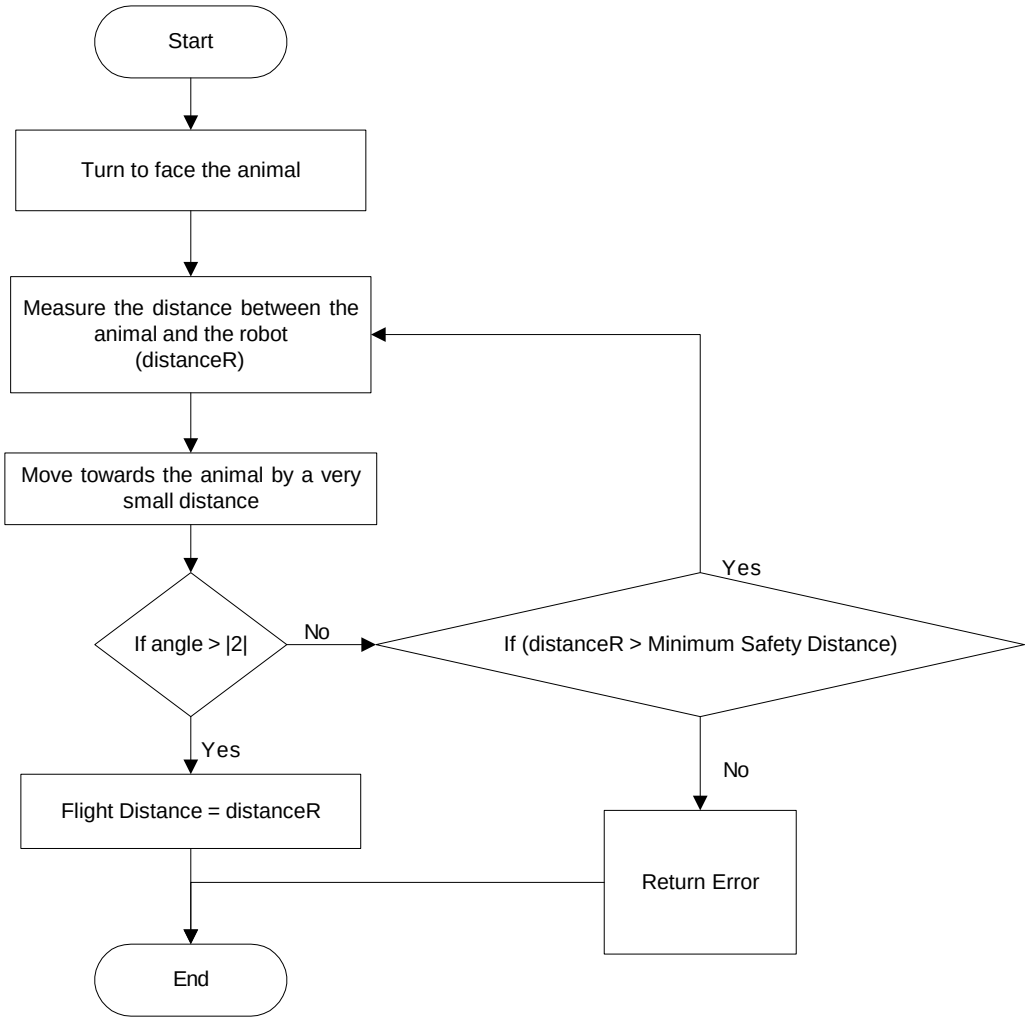


Figure 12. Flowchart describing initiate movement subroutine.

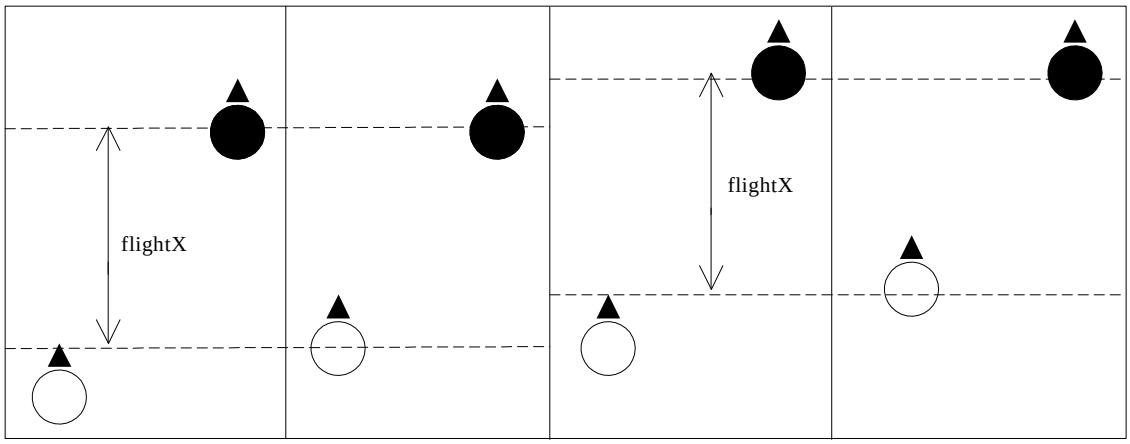
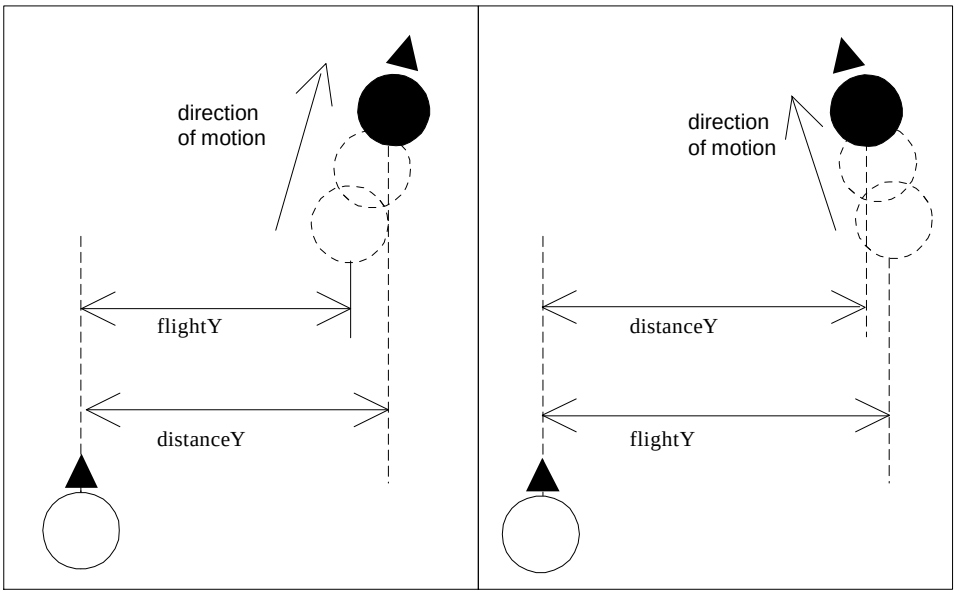


Figure 13. Herding forward.

Since the size of the flight zone is dynamic, it may change in size during the herding process. Flight zones that become larger do not present a problem, as the animal will move forward before the robot has reached the edge of the flight zone. If the flight zone becomes smaller, then the animal will not move forward when the robot reaches the edge of what it perceives or recalls as the flight zone. In this case, the robot moves forward for a pre-specified distance. If the animal resumes its forward motion, then the robot updates its flight zone (flightX) value, but if the animal still does not move, then the robot invokes the *Initiate Herding* subroutine to recalculate the distances, and subsequently invokes the *Herding Forward* routine.

Complicating this task is the requirement that the robot recognize when the animal might not be heading in the desired direction, but may be moving at some angle relative to the desired direction, as shown in Figure 14. The robot recognizes this behavior by comparing the difference between distanceY and flightY. If the diference is

greater than a pre-specified value, then the robot invokes the Initiate Movement routine to recalculate the distances (flightX, flightY), and subsequently invokes the Herding Forward routine; else it continues to herd the animal in the forward direction.



(a) (b)

Figure 14. Animal moving at an angle.

Turning the animal. To turn an animal, the robot turns and moves towards the rear of the animal until the animal is oriented in the desired direction, as shown in Figure 15. In Figure 15(a), the robot is facing the animal and is outside the flight zone. The robot moves forward until it reaches the edge of the flight zone, Figure 15(b). Hence, the animal moves forward, and as a result, the angle made by the animal decreases, Figure 15(c). The robot continues moving in the direction of the animal and, at some point in time, reaches the blind zone of the animal, Figure 13(d). To keep the robot in view, the

animal moves forward with a turned head, this results in its shoulders moving in the direction of its head, resulting in the turning movement, as shown in Figure 15(e). The robot continues to move toward the animal until the animal has made the desired turn, as shown in Figure 16.

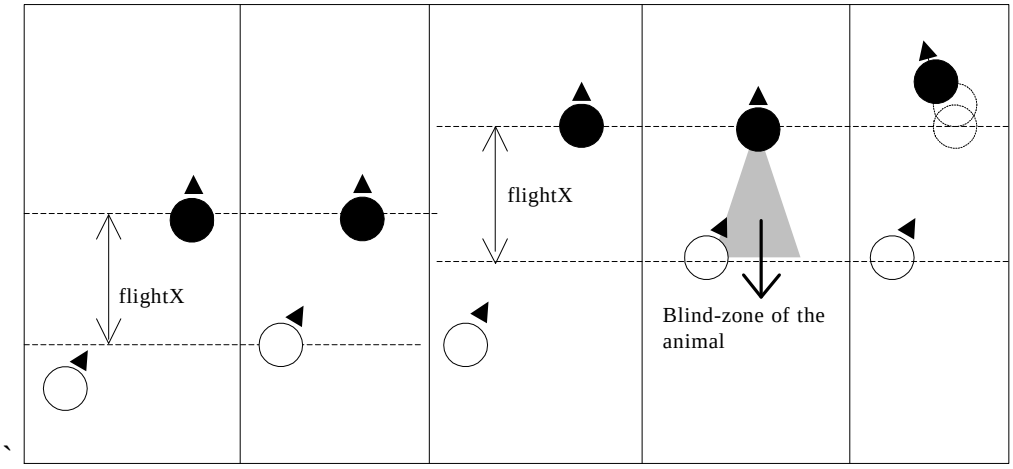


Figure 15. Steps to execute the turn routine.

Stopping the animal. To stop the animal from moving, the robot stops moving.

Assisted Herding Methodology

This section describes the algorithm used for developing assisted herding capability using low-stress herding techniques. It involves developing a module for perceiving human movements. Based on the value returned by the module, an appropriate herding module is invoked.

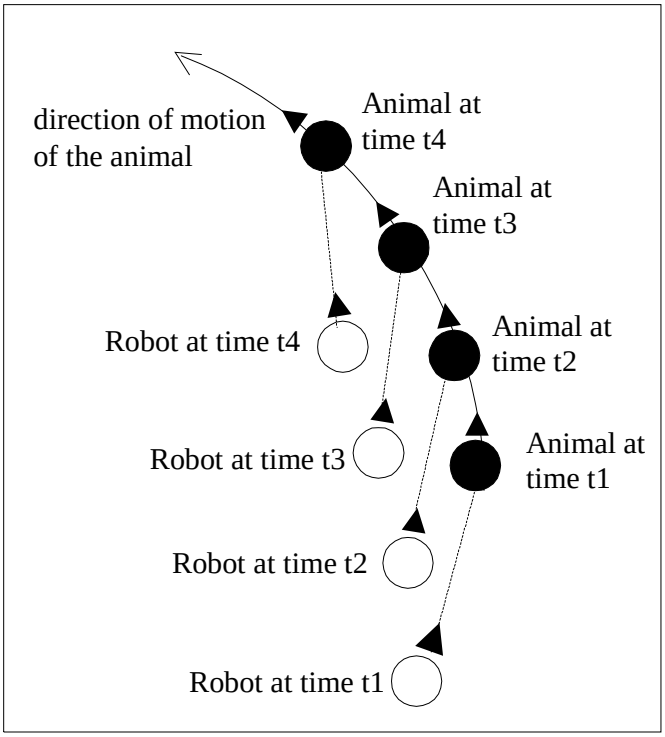


Figure 16. Turning the animal.

Initiate herding and herding forward. The robot first invokes the Initial Position subroutine. Upon successful completion of this subrouting, the robot starts tracking the human. When the distance between the human and animal reduces over time, it means that the human is moving toward the animal, as shown in Figure 17(a). This behavior is interpreted as the human wanting to initiate herding. The robot stops tracking the human and executes the Initiate Movement subroutine, followed by the Herding Forward routine. It resumes tracking the human while herding the animal forward.

Turning the animal. Once the herding commences, if the distance between the animal and the human decreases over time, then this means that the human is moving

towards the robot. The robot interprets this behavior as the human attempting to turn the animal. The robot stops its motion to let the human turn the animal, Figure 17(b). However, if the distance between the animal and the human increases over time, then this means that the human is moving away from the animal. This behavior is interpreted as the human asking the robot to turn the animal, in which case, the robot executes the *Turning the animal* routine, Figure 17(c).

Stopping the animal. To stop the animal, the human stops its motion. When the human is at an angle greater than 85 degrees relative to the robot, Figure 17(d), the robot perceives this as the human wanting to stop the animal, and so the robot executes the *Stopping the animal* routine.

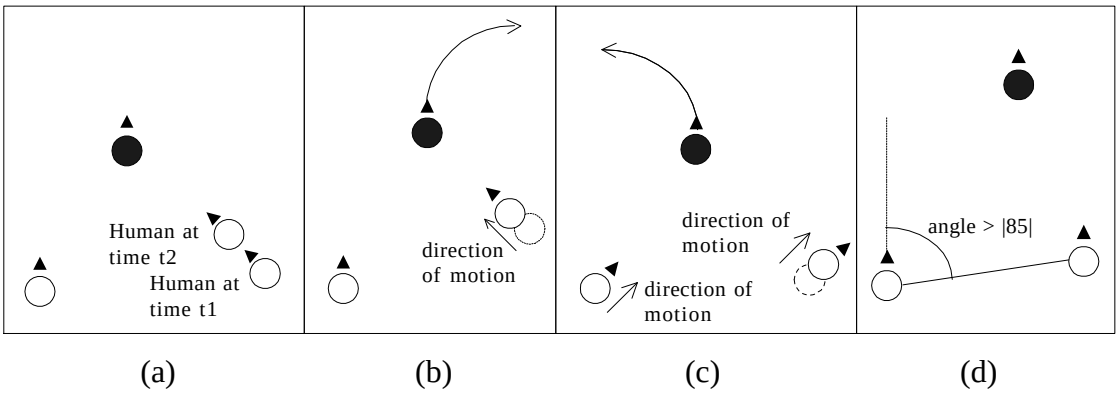


Figure 17. Assisted herding behavior.

CHAPTER 6

Experiments

Hypotheses

The experiments described here are designed to test the following hypotheses:

a. An autonomous mobile robot is able to obtain human instructions and intentions by perceiving and interpreting the movements of a human.
b. An autonomous mobile robot can carry out herding at the direction of, or in the absence of, the human.

We conduct four experiments to test our hypotheses. The first experiment is designed to test the autonomous herding capability of the robot in the absence of the human. It consists of two agents, the robot and the animal. The human is not present. We expect the robot to recognize the absence of the human and to be able to herd the animal autonomously.

The second, third, and fourth experiments consist of all three agents; robot, human, and animal. These experiments are designed to test the assisted herding capability of the robot. We expect the robot to recognize the presence of the human and herd the animal as per the intentions of the human.

In these experiments we have not measured the response time because in low-stress herding techniques, the animal is herded at a pace of the animal's choosing. This pace can vary based on the environmental conditions and is different for different animals.

Description of the Software and Hardware

The robot used in this work is a Pioneer 2-AT robot from Activmedia Robotics (ActivMedia Robotics, 2002; Mobile Robots, Inc., 2009). In addition, two remote-controlled cars, one tagged with an orange stick and the other, with a pink stick, are used. The car with the orange stick represents the animal and the car with the pink stick represents the human. The experiments are conducted in our test lab. Care is taken to ensure that that there are no other orange- or pink-colored objects in the lab besides the orange and pink sticks.

The Pioneer 2-AT robot, as shown in Figure 18, has a client–server architecture and comes equipped with:

a. SICK Laser-Range Finder (LRF)
b. PTZ robotic camera system
c. Front and rear sonar arrays
d. Four-wheel drive with independent motor drive
e. Inflatable pneumatic tires and metal wheels, making it ideal for use on rough terrain
f. 5 rear bumpers
g. ARIA, a C++-based client software that allows a quick interface with the robot and sensors

Figure18. Pioneer 2-AT robot.

SICK laser-range finder. The SICK laser-range finder is used to help determine the angle and distance between the robot and the animal, and between the robot and the human. It is also used for object tracking. It can scan up to 180 degrees in the forward facing direction (-90 to 90 degrees). It directs a narrow laser beam towards an object and measures the time taken by a pulse to be reflected off the object and returned to the sensor.

Even though the robot comes equipped with a sonar array, we use SICK laser-range finder to find and track objects because:

- The laser beam is narrower than the sonar beam, which spreads over a wider arc. The narrow focus of the laser allows for higher accuracy in locating the position of the object.
- The SICK laser-range finder can take 180 readings in 180 degrees (it has a horizontal view of 180 degrees), whereas the sonar can take only 1 reading in 8 degrees.

- The SICK laser-range finder can sense the objects between ten and fifty meters, while the sonar's range is limited to between three and six meters.

A disadvantage of using a laser range finder is that it scans objects in a single plane of less than one degree of vertical elevation. However, this was not an issue in our experiments as the color tags on the objects were positioned to intersect the plane of the robot's laser.

PTZ robotic camera system. The pan-tilt-zoom (PTZ) robotic camera system consists of PTZ software and a Sony D30/31 pan-tilt-zoom color camera, which can pan up to 180 degrees (-90 to 90 degrees).

The image sensor of Sony D30/31 camera is 1/3" Interline Transfer Charge-Coupled Device (IT CCD), the horizontal resolution is 460 TV lines National Television System Committee (NTSC); 450 TV lines Phase Alternating Line (PAL), and the vertical resolution is 350 TV lines NTSC; 400 TV lines PAL.

This system is used to capture the images at various angles. The images are then fed to the image processing system to identify the animal and the human.

Bumper sensors. The robot comes equipped with five rear bumpers. The purpose of the bumpers is to halt the robot when any one of them is pressed, thereby, preventing damage to the robot. The bumpers are also used to give cues to the robot to turn the animal during the experiments.

ARIA (ActivMedia Robotics Interface for Applications). ARIA is an open-source library written in C++ that handles lower-level details, such as interrupts, motor control, and multithreading. It also has an application programming interface (API) that enables

the user to interface the software with the robot and its various sensors, such as cameras, sonar, laser, gyrometer, and gripper arms.

Toy Cars. Two remote-controlled toy cars were used to simulate the behavior of the animal and the human. The cars could be moved forward, in reverse, turned left or right, and stopped by a joystick located on a remote-control box. The car representing the animal had an orange colored stick mounted on top of it and operated at a 27 MHz frequency. The car representing the human in the experiment had a pink colored stick mounted on top of it and operated at a 49 MHz frequency. In future work, we plan to replace color with shape, size, texture, recognition, and tracking capabilities.

Autonomous Herding Behavior

The experiment described in this section tests autonomous herding capabilities of the robot in the absence of the human.

Experiment 1. The robot is placed in such a way that the angle it makes to the rear of the animal is approximately -20 degrees. The task for the robot is to herd the animal forward. After the robot has herded the animal in the forward direction for one minute, the flight zone of the animal is reduced. When this happens the task for the robot is to re-determine the flight zone and begin herding the animal forward. The robot should turn the animal by 90 degrees when one of it bumpers is pressed. In a real-world context, a map of the terrain would be available for the robot to determine when to turn an animal towards the goal location. In this experiment, the task for the robot is to turn the animal 90 degrees. Once the animal is turned, the robot should resume herding the animal in the

forward direction.

Results. The robot scans the environment to locate the animal. Once the animal is located, it turns -90 degrees and moving backwards until the animal is at an angle of 45 degrees with respect to the robot, as shown in Figure 19. Next, the robot scans the environment to locate the human. When it cannot locate the human, it turns to face the animal and slowly moves forward until the animal begins to move forward. The robot then turns in the direction in which the animal is facing and starts following it from the outrider's position.

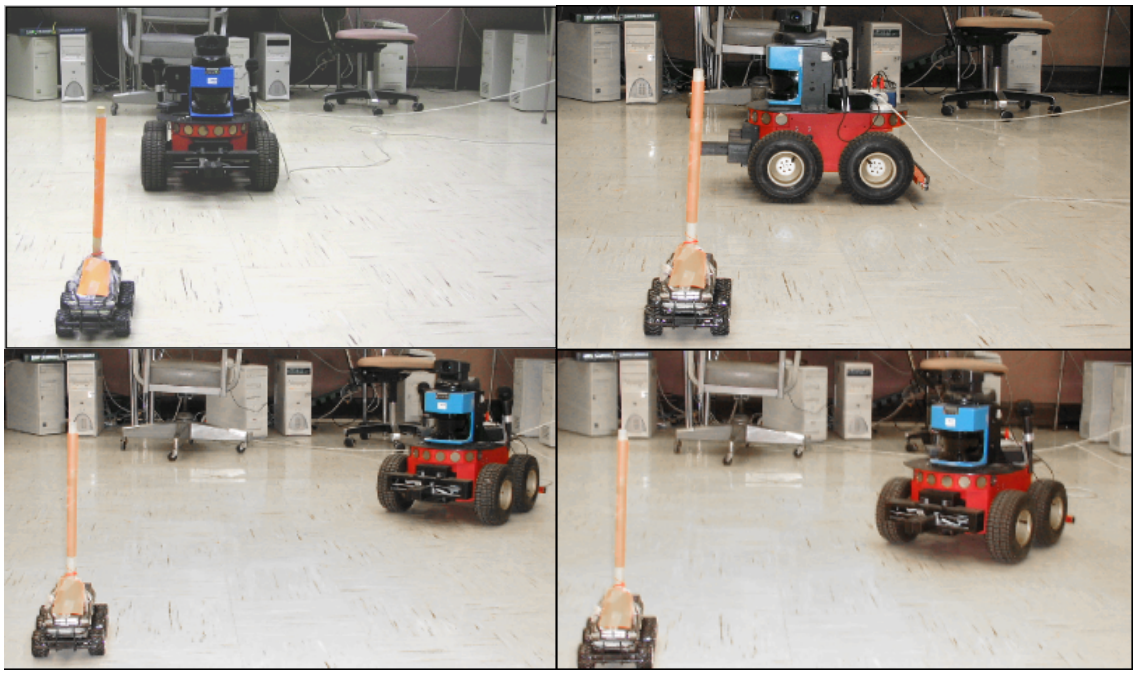


Figure 19. Initiate herding.

After herding the animal forward for one minute, the animal stops and does not move. Perceiving this, the robot approaches the animal until the animal starts moving

forward again. The robot then turns in the direction in which the animal is facing and starts herding the animal in the forward direction.

When one of the bumpers of the robot is pressed to indicate that the animal should be turned, the robot turns and moves towards the animal, as shown in Figure 20. When the animal has turned by an approximately -90 degrees, the robot turns -90 degrees and begins herding the animal in the forward direction.

Discussion. In this experiment, we demonstrated that the robot is able to perceive the absence of the human and is able to autonomously herd the animal in the desired direction using low-stress herding techniques.

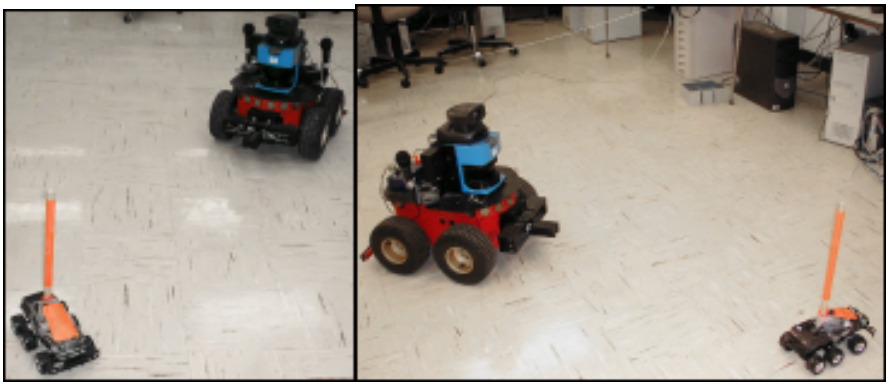


Figure 20. Robot turning the animal.

In practice, environmental conditions such as the size and orientation of shadows, lighting, and other factors can dynamically change an animal's effective flight zone and the herder may unexpectedly find itself outside the flight zone. It is therefore possible that the animal stops while being herded. We tested this situation by reducing the flight

zone when the robot was herding the animal in the forward direction. In this case, the robot stopped herding and invoked the Initiate Herding routine.

Assisted Herding Behavior

In these experiments, we evaluate the robot's ability to herd the animal as per the intentions of the human. The experiments described below consist of all three agents, human, robot, and animal.

Experiment 2. The robot is placed in such a way that the angle it makes to the rear of the animal is approximately 20 degrees. The task for the robot is to assist the human in herding the robot in the forward direction.

*Result*s. The robot scans the environment to locate the animal. When the animal is located, the robot turns 90 degrees and moves backwards until the animal is at -45 degrees with respect to the robot, as shown in Figure 21.

The robot scans the environment to locate the human. When the human is located, it waits for the human to move in the direction of the robot. As soon as the human starts moving towards the animal, the robot detects this and begins to move towards the animal. From hereafter, the robot takes the same actions that it took in Experiment 1, until the animal is herded in the forward direction.

After the animal is herded for a while, the human stops its motion. The robot continues to herd the animal forward until the human is at an angle 85 degrees with respect to the robot. The robot then stops moving to stop herding the animal.

Discussion. In this experiment, we demonstrate the assisted behavior of the robot.

The robot is able to perceive the intentions of the human to commence herding when it sees the human walking towards the animal. It stops herding when it sees the human is no longer herding the animal.

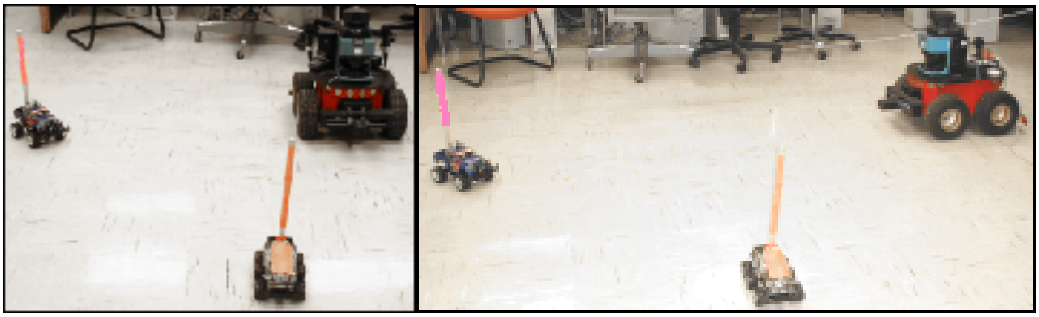


Figure 21. Initial position.

Experiment 3. The robot has to assist the human in turning the animal. The robot is placed in such a way that the angle it makes to the rear of the animal is approximately 40 degrees.

Results. The robot scans the environment to locate the animal. Since the robot is already at the initial herding position, it begins scanning the environment to locate the human. After locating the human, the robot takes the same actions that it took in Experiment 2, until the animal is herded in the forward direction.

After the animal is herded for a while, the human turns and starts moving at an angle of -45 degrees with respect to the animal. Seeing this, the robot invokes the turn routine, as it did in Experiment 1.

Discussion. In this experiment, when the robot sees the human walking away from the animal, it is able to correctly perceive and interpret this action as a command for

it to turn the animal.

Experiment 4. The robot has to assist the human in turning the animal as before. However, in this case, the human performs the turning operation. The robot is placed in such a way that the angle it makes to the rear of the animal is approximately 40 degrees.

Results. Since the robot is already at the initial herding position, it takes the same actions as described in Experiment 3 until the animal is herded in the forward direction. After herding the animal forward for a while, the human turns and moves towards the animal in order to turn it. Seeing this, the robot stops its motion.

Discussion. In this experiment, when the human wants to perform the turning action, the human starts walking towards the animal. The robot is able to perceive this action correctly and stops its motion to let the human turn the animal.

CHAPTER 7

Conclusion and Future Work

In this research, we demonstrate a first robotic system that is able to herd animals using low-stress herding techniques. We also demonstrate a robotic system that is able to assist a human with herding an animal. We show that the system is able to adjust its herding behavior as per the intentions of the human that it perceived, by interpreting the movements of the human. A key factors here is that the assisted herding task is carried out with minimal communication and within un-instrumented environments

The work described here is limited to herding a single cattle-type animal. In the future, we plan to develop methods for scaling our approach to handle multiple animals. Another limitation of the work is the use of color tags to recognize the animal and the human. As part of a future study, we intend to develop more advanced object recognition algorithms that take into account several other attributes such as shape and size.

REFERENCES

ActivMedia Robotics. (2002). *Pioneer 2: PeopleBot Operations Manual. V. 11*. Peterborough, NH: ActivMedia Robotics.

Bruce, A. (2005). *Planning for Human and Robot Interaction: Representing Time and Human Intention.* Thesis, Robotics Institute, Carnegie Mellon University.

Butler, Z. Corke, P., Peterson, R. & Rus, D. (2004). Virtual fences for controlling cows. *Proceedings of the IEEE International Conference of Robotics and Automation*. New Orleans, LA, 4429-4436**.**

Cerqui, D., & Arras, K. O. (2003). Human beings and robots: towards a symbiosis? - a 2000 people survey. *International Conference on Socio Political informatics and Cybernetics (PISTA'03)*. Orlando, FL, 408-413.

Elsasser, T., (2002, January). Detecting stress in animals. *Agricultural Research Magazine,* 50, 10-100.

Fanatico, A., & Morrow, R., (1999, August). Sustainable beef production – Live stock production guide. In *ATTRA National Sustainable Agriculture Information Service,* p.6.

Grandin, T. (1989, December). Behavioural principles of livestock handling. In *American Registry of Professional Animal Scientists*, pp. 1-11.

Jones, J. L., Seiger, B. A., & Flynn, A. M. (1999). *Mobile Robots: Inspiration to Implementation.* Natick, MA: A.K. Peters.

Mobile Robots, Inc. (2009). Research and University Robots, Software and Accessories. Retrieved September 15, 2009 from the Mobile Robots web site: http://robots.activrobots.com

Rahimi, M., & Karwowski, W. (1992). *Human-Robot Interaction.* London: Taylor & Francis.

Reichenbach, J., Bartneck, C., & Carpenter, J. (2006). Well done, robot! - the importance of praise and presence in human-robot collaboration. *Proceedings of RO-MAN 2006: 15th IEEE International Symposium on Robot and Human Interactive Communication*. Hatfield, UK, 86-90.

Sheridan, T. B. (2002). *Humans and Automation: System Design and Research Issues.* New York : Santa Monica, Calif: John Wiley; Published in cooperation with Human Factors and Ergonomics Society.

Siegwart, R., & Nourbakhsh, I. R. (2004). *Introduction to Autonomous Mobile Robots*. Cambridge, MA: MIT Press.

Smith, B. J. (1998). *Moving 'Em: A Guide to Low Stress Animal handling.* Kamuela, HI: Graziers Hui.

Sony Corporation. (n.d.). Aibo: your artifical intelligent companion. Retrieved September 15, 2009 from the Sony web site: http://support.sony-europe.com/aibo/

Steinfeld, A. (2004). Interface lessons for fully and semi-autonomous mobile robots. *Proceedings of the IEEE International Conference on Robotics and Automation.* New Orleans, LA, 2752-2757.

Topp, E.A., Kragic, D., Jensfelt, P., Christensen, H.I., (2004). An interactive interface for service robots. *ICRA '04, IEEE International Conference on Robotics and Automation*. New Orleans, LA, volume 4, pages 3469 – 3474.

Trafton, J. G., Cassimatis, N. L., Bugajska, M. D., Brock, D. P., Mintz, F. E., & Schultz, A. C. (2005). Enabling effective human–robot interaction using perspective-taking in robots. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 35(4), 460-470.

Vaughan, R. T., Sumpter, N., Henderson, J. V., Frost, A., & Cameron, S. (2000). Experiments in automatic flock control. *Robot and Autonomous Systems*, 31, 109-117.

WowWee Robotics. (n.d.). Robosapien, Retrieved September 15, 2009 from the WowWee web site: http://www.wowwee.com/en/products/toys/robots/robotics:robosapien

Yoshizaki, M., Kuno, Y., Nakamura, A., (November 2001). Human-Robot Interface based on the Mutual Assistance between Speech and Vision. *Proceedings of the 2001 workshop on Perceptive user interfaces, ACM International Conference Proceeding Series.* Orlando, FL, volume 15, pages 1 – 4.

ZMP, Inc. (n.d.). Nuvo: the first humanoid robot for home use. Retrieved September 15, 2009 from the ZMP web site: http://nuvo.jp/nuvo_home_e.html

Appendix

Source Code

HDActionHuman.h

```
#ifndef ARACTIONBUMPERS_H
#define ARACTIONBUMPERS_H
#include "ariaTypedefs.h"
#include "ArAction.h"
#include "HDObjectPose.h"
class HDActionHuman: public ArAction
{
public:
        AREXPORT HDActionHuman(const char *name = "TrackObjects", ObjectPose
        *objPose = NULL, ArSick* sick = NULL);
        AREXPORT virtual ~HDActionHuman();
        AREXPORT virtual ArActionDesired *fire(ArActionDesired currentDesired);
        AREXPORT virtual ArActionDesired *getDesired(void) { return &myDesired; }
        AREXPORT void  setObjPose(ObjectPose objPose);
        AREXPORT ObjectPose getObjPose();
        AREXPORT ObjectPose getPrevObjPose();
        ObjectPose objPose;
        ObjectPose prevObjPose;
```

```
protected:
        ArSick sick;
        ArActionDesired myDesired;
        bool valid;
};
#endif // ARACTIONBUMPERS
```

HDAnimal.h

```
#include "Aria.h"
#include "HDObjectPose.h"
#ifndef HDAnimal_H
#define HDAnimal_H
const int UNKNOWN = -1;
const int ANIMAL_MOVING_STRAIGHT       = 0;
const int ANIMAL_NOT_MOVING                    = 1;
const int ANIMAL_MOVING_TOWARDS_ROBOT      = 2;
const int ANIMAL_MOVING_AWAY             = 3;
const int ANIMAL_NOT_FOUND                 = 4;
const int HUMAN_GO_FORWARD                 = 5;
const int HUMAN_GO_TOWARDS_ANIMAL  = 6;
const int HUMAN_STEP_BACK              = 7;
const int HUMAN_GO_AWAY_FROM_ANIMAL = 8;
```

```
const int HUMAN_STOPPED                 = 9;
const int HUMAN_NOT_FOUND               = 10;
const int HERD_STRAIGHT                 = 11;
const int HERD_RIGHT90                  = 12;
const int HERD_LEFT90                = 13;
const int HERD_STOP                     = 14;
const int HERD_GO_TOWARDS_ANIMAL = 15;
const int HERD_Y_INCREASES          = 16;
const int HERD_Y_DECREASES          = 17;
const int HUMAN_EVENT = 18;
const int ANIMAL_EVENT = 19;
const int ROBOT_CANNOT_MOVE = 20;
class HDAnimal
{
        public:
          static int hooktime;
          static int MinFlightDist;
          static int MaxFlightDist;
          static int AngleofApproach ;
          static int MinAngleofApproach ;
          static int MaxAngleofApproach ;
          static int Status;
```

```
        static double FlightDist;
        static double FlightAngle;
        static ObjectPose *animalpose;
        static ObjectPose a1;
};
#endif
```

HDBumpers.h

```
#include "Aria.h"
#ifndef HDBumpers_H
#define HDBumpers_H
class HDBumpers
{
      public:
        HDBumpers();
        ~HDBumpers();
        bool isPressed();
};
#endif
```

HDGoTowardsAnimal.h

```
#include "Aria.h"
```

```
#ifndef HDGoTowardsAnimal_H
#define HDGoTowardsAnimal_H
#include "HDObjectPose.h"
#include "HDUtils.h"
class HDGoTowardsAnimal
{
        public:
          HDGoTowardsAnimal();
          virtual  ~HDGoTowardsAnimal(void);
          bool goTowardsAnimal();
          bool animalMovedStraight(ObjectPose objpose);
        protected:
          HDUtils *utils;
          ObjectPose *animalpose;
          ArRobot* robot;
          ArSick* sick;
          ArSonyPTZ* cam;
          double init_angle;
          char *buf;
          bool isHumanMovingTowardsAnimal(ObjectPose objpose);
          bool  pointToAnimal();
          bool  pointToAnimal(ObjectPose objpose);
```

```
};
#endif
```

HDHerdStraight.h

```
#include "Aria.h"
#ifndef HDHerdStraight_h
#define HDHerdStraight_h
#include "HDObjectPose.h"
#include "HDUtils.h"
#include "HDHumanAction.h"
class HDHerdStraight
{
        public:
          HDHerdStraight();
          ~HDHerdStraight(void);
          void  herdStraight();
          int STATUS;
        protected:
          HDUtils *utils;
          ObjectPose *animalpose;
          ObjectPose *prevanimalpose;
          ArRobot* robot;
```

```
        ArSick* sick;
        ArSonyPTZ* cam;
        int ERROR;
        ObjectPose* humanpose;
        HDHumanAction *humanAction;
        void updatePoses(double angleBeforeTurn);
        char *buf;
        int ANIMAL_NOT_MOVING;
        int ANIMAL_MOVING_TOWARDS_ROBOT;
        int ANIMAL_MOVING_AWAY;
        int ANIMAL_MOVING_STRAIGHT;
        double getFlightAngle();
        bool moveRobot(double FlightX);
        int getAnimalStatus(double y, double FlightX);
        ObjectPose getAnimalPose(double angle);
        ObjectPose getAnimalPose();
};
#endif
```

HDHumanAction.h

```
#include "Aria.h"
#ifndef HDHumanAction_h
```

```
#define HDHumanAction_h
#include "HDObjectPose.h"
#include "HDUtils.h"
class HDHumanAction
{
        public:
          HDHumanAction();
          virtual  ~HDHumanAction(void);
          int getStatus();
          void initHumanParams();
          void updateHumanPose(double angle);
          ObjectPose getHumanPose();
          static int STATUS;
          bool isHumanHerding();
        protected:
          void getPoses();
          void lookforhuman(ObjectPose* p1);
          HDUtils *utils;
          ArRobot* robot;
          int ERROR;
          ObjectPose* animalpose;
          ObjectPose* humanpose;
```

```
        ObjectPose* prevHumanPose;
        ObjectPose* prevGlobalHumanPose;
        ObjectPose* currGlobalHumanPose;
        ArTime start, stop;
        char *buf;
        double disty;
        double angle;
        int prevStatus;
};
#endif
```

HDInitPosition.h

```
#include "Aria.h"
#ifndef HDInitPosition_H
#define HDInitPosition_H
#include "HDObjectPose.h"
#include "HDUtils.h"
class HDInitPosition
{
      public:
        HDInitPosition();
        virtual  ~HDInitPosition(void);
```

```
        void goToInitPostion();
        bool trackAndMoveTill45(ObjectPose *objpose);
        void updateAnimalPose(double angle);
    protected:
        HDUtils *utils;
        int MaxFlightDist;
        FILE *myFile;
        ArRobot* robot;
        ArSick* sick;
        ArSonyPTZ* cam;
        char *buf;
        ObjectPose *animalpose;
};
#endif
```

HDObjectPose.h

```
#include "Aria.h"
#ifndef HDObjectPose_H
#define HDObjectPose_H
class ObjectPose
{
    public:
```

```
        ObjectPose();
        ObjectPose(double x, double y, char *c);
        ObjectPose(double dist, double angle);
        ~ObjectPose(void);
        ObjectPose getGlobalCoordinates();
        void copy(ObjectPose* src, ObjectPose* dest);
        double dist;
        double angle;
        double x;
        double y;
        ArPose robotpose;
};
#endif
```

HDObjects.h

```
#include "Aria.h"
#ifndef HDObjects_H
#define HDObjects_H
#include "HDObjectPose.h"
#include "HDVision.h"
#include "HDBumpers.h"
class HDObjects
```

```
{
        public:
          static ArRobot *robot;
          static ArSick * sick;
          static ArSonyPTZ *cam;
          static HDVision *vision;
          static HDBumpers* bumpers;
          static ObjectPose *animalpose;
          static ObjectPose *human;
          static int ErrorAngle;
          static int ErrorDist;
          static int PINK_MAXHUE;
          static int PINK_MINHUE;
          static int PINK_MAXSAT;
          static int PINK_MINSAT;
          static int PINK_MAXLUM;
          static int PINK_MINLUM;
          static int ORANGE_MAXHUE;
          static int ORANGE_MINHUE;
          static int ORANGE_MAXSAT;
          static int ORANGE_MINSAT;
          static int ORANGE_MAXLUM;
```

```
        static int ORANGE_MINLUM;
        static int ANIMAL_COLOR;
        static int HUMAN_COLOR;
        static int PINK;
        static int ORANGE;
        static bool HUMAN_PRESENT;
        static int HERDING_DIRECTION;
        static int HERDING_STATUS;
        static void createObjects();
        static void destroyObjects();
};
#endif
```

HDUtils.h

```
#include "Aria.h"
#ifndef HDUTILS_H
#define HDUTILS_H
#include "HDObjectPose.h"
#include "HDBumpers.h"
#include "HDVision.h"
class HDUtils
{
```

```
public:
  HDUtils();
  ~HDUtils(void);
  double mod(double no);
  HDVision *vision;
  bool loadWorld(char* wld);
  ObjectPose turnbyDelta(double angle);
  void turn(double angle);
  void turnbyDeltaApproximate(double angle);
  void turnApproximate(double angle);
  void turnLeft(double angle);
  ObjectPose pointToObject(double beginangle, double endangle);
  void turnRight(double angle);
  void turnRight();
  void turnLeft();
  void goForward(double dist);
  ObjectPose getClosetObjectDistance(double beginangle, double endangle);
  ObjectPose getClosetObjectDistance();
  double move(double dist);
  bool trackAndMoveTill45(double angle, double maxDistToMove, ObjectPose
*objPose = NULL);
  bool equals(double x, double y);
```

```
  bool equals(double x, double y, double diff);
  bool isBumperPressed();
  bool cannotMoveForward();
  void stopTransalationMotion();
  void stopRotationalMotion();
  void setTranslationVelocity(double velocity);
  void pointCamera(double angle);
  bool robotCannotMove();
  void display(ObjectPose objpose);
  void log(ObjectPose objpose);
  ObjectPose getAnimalLocation(int begin_angle, int end_angle, bool display =
false );
  ObjectPose getHumanLocation(int begin_angle, int end_angle, bool display =
false );
  ObjectPose getObjectLocation(int begin_angle, int end_angle, double max_hue,
double min_hue, double max_sat, double min_sat, double max_lum, double
min_lum, bool display = false );
  double getNewAngle(double fromangle, double toangle, double objangle);
  double getAngle(ObjectPose prev, ObjectPose curr);
protected:
  ArRobot *robot;
  ArSick *sick;
```

```
        ArSonyPTZ *cam;
        HDBumpers *bumpers;
        int myBumpMask;
        char *buf;
};
#endif
```

HDVision.h

```
#ifndef HDVISION_H
#define HDVISION_H
extern "C"
{
    #include "vislib.h"
    class HSL
    {
    public:
      double hue;
      double sat;
      double luminosity;
    } ;
    class HDVision
    {
```

```
  public:
        HDVision::HDVision();
        ~HDVision();
        bool IsObject(int* column, double max_hue, double min_hue, double
max_sat, double min_sat, double max_lum, double min_lum, bool display1 =
false);
        bool IsObject();
        bool ERROR;
        vlImage *img;
        void setDisplay(bool display1);
        int img_index;
  protected:
        HSL getHSL(int red, int blue, int green);
        bool isWithinRange(int r, int b, int g);
        bool initialize_display;
        double max_h;
        double min_h;
        double max_s;
        double min_s;
        double max_l;
        double min_l;
        bool initialized;
```

```
            vlPoint *point;
            bool display1;
            FILE* visionFile;
        };
}
#endif
```

HDConnection.h

```
#include "Aria.h"
class HDConnection
{
        public:
          HDConnection(void);
          virtual ~HDConnection(void);
          bool connect(bool laser=true, bool sonar=false);
          ArRobot *robot;
          bool CONNECTION_ERROR;
        protected:
          void enableMotors();
          ArSick *sick;
          ArSimpleConnector *connector;
          void resetParams();
```

```
};
```

HDConnection.cpp

```
#include "HDConnection.h"
#include "HDObjects.h"
#include "HDObjectPose.h"
HDConnection::HDConnection()
{
        HDObjects::createObjects();
        robot = HDObjects::robot;
        sick  = HDObjects::sick;
        if(!connect(true, false))
          CONNECTION_ERROR = true;
        else
          CONNECTION_ERROR = false;
}
HDConnection::~HDConnection(void)
{
        resetParams();
        HDObjects::destroyObjects();
        delete connector;
}
```

```
void HDConnection::resetParams()
{
        robot->lock();
        robot->setAbsoluteMaxTransVel(1000.0000);
        robot->setTransVelMax(1000.0000);
        robot->setRotVelMax(200.000000);
        robot->unlock();
        ArUtil::sleep(1000);
}
bool HDConnection::connect(bool laser, bool sonar)
{
        int argc = 0;
        char **argv = NULL;
        connector = new ArSimpleConnector (&argc, argv);
        connector->parseArgs();
        if (!connector->connectRobot(robot))
        {
          printf("Could not connect to robot... exiting\n");
          Aria::shutdown();
          return false;
        }
        robot->runAsync(true);
```

```
        if(!sonar)
          robot->comInt(ArCommands::SONAR, 0);
        if(laser)
        {
          robot->addRangeDevice(sick);
          connector->setupLaser(sick);
          sick->runAsync();
          if (!sick->blockingConnect())
          {
                printf("Could not connect to SICK laser... exiting\n");
                Aria::shutdown();
                return false;
          }
          sick->setConnectionTimeoutTime(0);
        }
        robot->setCycleWarningTime(10000);
        robot->setConnectionTimeoutTime(0);
        enableMotors();
        return true;
}
void HDConnection::enableMotors()
{
```

```
        robot->enableMotors();
        robot->setTransVelMax(100);
        robot->setAbsoluteMaxTransVel(100);
        ArUtil::sleep(100);
        if(robot->areMotorsEnabled())
          printf("\nmotors are enabled");
        else
          printf("\nmotors are disabled");
        ArUtil::sleep(100);
}
```

HDHerd.h

```
#include "Aria.h"
#ifndef HDHerd_H
#define HDHerd_H
#include "HDObjectPose.h"
#include "HDUtils.h"
class HDHerd
{
        public:
          HDHerd();
          ~HDHerd();
```

```
    ObjectPose straight();
    void turn(int angle);
    void stop();
    ObjectPose initPosition();
    ObjectPose hook();
    ObjectPose *animalpose;
    void setAnimalPose(ObjectPose pose);
    HDUtils *utils;
    void goTowardsAnimal();
    bool goStraight();
    double getAngle(ObjectPose prev, ObjectPose curr);
    bool hasObjectMoved(ObjectPose prev_obj, ObjectPose curr_obj);
    ObjectPose pointToAnimal(int begin_angle, int end_angle);
  private:
    char* buf;
    ArRobot* robot;
    int NEXTSTATE;
    ObjectPose initpose;
    bool isWithinAngleOfApproach(double angle);
    double flightangle;
};
#endif
```

HDHerd.cpp

```
#include "Aria.h"
#include "HDHerd.h"
#include "HDInitPosition.h"
#include "HDUtils.h"
#include "HDObjectPose.h"
#include "HDAnimal.h"
#include "HDObjects.h"
#include "HDGoTowardsAnimal.h"
#include "HDHerdStraight.h"
int INIT_POSITION = 0;
int GO_TOWARDS_ANIMAL = 1;
int GO_STRAIGHT = 2;
int TURN = 3;
HDHerd::HDHerd()
{
        animalpose = HDAnimal::animalpose;
        utils = new HDUtils();
        buf = new char[1000];
        robot = HDObjects::robot;
}
HDHerd::~HDHerd()
```

```
{
        delete utils;
}
ObjectPose HDHerd::straight()
{
        ArPose pose;
        ObjectPose global_animalpose = animalpose->getGlobalCoordinates();
        return global_animalpose;
}
void HDHerd::stop()
{
        utils->move(-300);
}
ObjectPose HDHerd::initPosition()
{
        printf("\n****goToInitPostion****");
        HDInitPosition initPose;
        initPose.goToInitPostion();
        if(animalpose->dist != 0)
          utils->display(*animalpose);
        NEXTSTATE = GO_TOWARDS_ANIMAL;
        return *animalpose;
```

```
}
void HDHerd::setAnimalPose(ObjectPose pose)
{
        *animalpose = pose;
}
void HDHerd::goTowardsAnimal()
{
        HDGoTowardsAnimal approachAnimal;
        approachAnimal.goTowardsAnimal();
        return;
}
bool HDHerd::isWithinAngleOfApproach(double angle)
{
        if( utils->mod(angle) > HDAnimal::MinAngleofApproach && utils->mod(angle)
        < HDAnimal::MaxAngleofApproach)
          return true;
        return false;
}

ObjectPose HDHerd::hook()
{
        NEXTSTATE = 1;
```

```
        goTowardsAnimal();
        return *animalpose;
}
bool HDHerd::goStraight()
{
        HDHerdStraight* hdstraight = new HDHerdStraight();
        hdstraight->herdStraight();
        delete hdstraight;
        return true;;
}
void HDHerd::turn(int angle)
{
        utils->turnbyDelta(animalpose->angle);
        ArUtil::sleep(1000);
        ObjectPose* p1 = new ObjectPose();
        *p1 = *animalpose;
        *animalpose = utils->getClosetObjectDistance(-5, 5);
        if(!utils->equals(animalpose->dist, p1->dist, 300))
        {
          *animalpose = utils->getAnimalLocation(0, 0, false);
          if(animalpose->dist == 0)
          {
```

```
        *animalpose = utils->getAnimalLocation(-90, 90, false);
        if(animalpose->dist == 0)
            return;
  }
}
ArTime start;
ObjectPose* prevPose = new ObjectPose();
ObjectPose globalpose;
ObjectPose globalPrevPose;
*prevPose = *animalpose;
*p1 = *animalpose;
double angle;
int angle2turn;
if(animalpose->angle < 0)
  angle2turn = 90 -10;
else
  angle2turn = -90 +10;
time_t t1;
start.setToNow();
while(1)
{
  if(animalpose->angle > 2 ||  animalpose->angle < -2)
```

```
        robot->setDeltaHeading(animalpose->angle);
  robot->move(animalpose->dist);
  *animalpose =  utils->getClosetObjectDistance(animalpose->angle-3,
animalpose->angle+3);
  if(!utils->equals(animalpose->dist, p1->dist, 300))
  {
        robot->setDeltaHeading(0);
        robot->move(0);
        time_t t1 = start.getMSec();
        *animalpose = utils->getAnimalLocation(p1->angle, p1->angle, false);
        if(animalpose->dist == 0)
              *animalpose = utils->getAnimalLocation(p1->angle - 15, p1->angle -
15, false);
        if(animalpose->dist == 0)
              *animalpose = utils->getAnimalLocation(p1->angle + 15, p1->angle +
15, false);
        if(animalpose->dist == 0)
              *animalpose = utils->getAnimalLocation(-90, 90, false);
        if(animalpose->dist == 0)
              break;
        start.setMSec(t1);
  }
```

```
globalpose    = animalpose->getGlobalCoordinates();
if(start.mSecSince() > 1000)
{
      globalPrevPose = prevPose->getGlobalCoordinates();
      if(hasObjectMoved(globalPrevPose, globalpose) )
      {
            ArLog::logPlain(ArLog::Terse, buf);
            angle  = utils->getAngle(globalpose, globalPrevPose);
            ArLog::logPlain(ArLog::Terse,buf);
            if( (angle2turn < 0 && angle < angle2turn)
                  || (angle2turn > 0 && angle > angle2turn) )
                  break;
            *prevPose = *animalpose;
            start.setToNow();
      }
}
if(utils->robotCannotMove())
{
      printf("\n robot cannot move");
      break;
}
*p1 = *animalpose;
```

```
        ArUtil::sleep(100);
      }
      robot->setDeltaHeading(0);
      robot->move(0);
      delete prevPose;
      delete p1;
}
double HDHerd::getAngle(ObjectPose prev, ObjectPose curr)
{
      double x = (prev.x - curr.x);
      double y = (prev.y - curr.y);
      double angle = ArMath::radToDeg(atan(y/x));
      return angle;
}
bool HDHerd::hasObjectMoved(ObjectPose prev_obj, ObjectPose curr_obj)
{
      if(utils->mod(prev_obj.x - curr_obj.x) < 50 && utils->mod(prev_obj.y -
      curr_obj.y) < 50)
        return false;
      return true;
}
```

HDHerding.cpp

```
#include <math.h>
#include "Aria.h"
#include "HDObjects.h"
#include "HDConnection.h"
#include "HDAnimal.h"
#include "HDUtils.h"
#include "HDInitPosition.h"
#include "HDBumpers.h"
#include "HDHerd.h"
#include "HDHumanAction.h"
#include "HDActionHuman.h"
class HDHerding : public HDConnection
{
        public:
          HDHerding();
          virtual  ~HDHerding(void);
          void startHerding(void);
          bool initSystem(void);
          bool isHumanHerding(void);
          bool hasObjectMoved(ObjectPose prev_obj, ObjectPose curr_obj);
          void findHuman();
```

```
    void findAnimal();
    void turn();
  protected:
        HDUtils         *utils;
        HDHerd          *herd;
        HDHumanAction*humanAction;
        ObjectPose* human;
        ObjectPose *animalpose;
        char* buf;
};
HDHerding::HDHerding()
        :HDConnection()
{
  buf = new char[1000];
  utils = new HDUtils();
  herd =  new HDHerd();
  humanAction = new HDHumanAction();
  animalpose = HDAnimal::animalpose;
  human = HDObjects::human;
  ArLog::init(ArLog::File, ArLog::Terse, "herdstraight.xml");
  ArLog::logPlain(ArLog::Terse,"<start>");
}
```

```
HDHerding::~HDHerding(void)
{
        ArLog::logPlain(ArLog::Terse,"</start>");
        delete utils ;
        delete [] buf;
        delete herd;
        delete animalpose;
        delete human;
        delete humanAction;
}
bool HDHerding::initSystem()
{
        if(CONNECTION_ERROR)
          return false;
        if(robot->getRotVelMax() < 200.000000)
        {
          robot->setRotVelMax(200.000000) ;
          ArUtil::sleep(1000);
          if(robot->getRotVelMax() < 200.000000)
          {
                printf("\nerror: RotVelMax is %lf < 200.00", robot->getRotVelMax() );
                return false;
```

```
  }
}
if(robot->getAbsoluteMaxRotVel() < 360.000000)
{
  printf("\nerror: AbsoluteMaxRotVel is %lf < 360.00", robot-
>getAbsoluteMaxRotVel() );
  robot->setAbsoluteMaxRotVel(360.000000);
  ArUtil::sleep(1000);
  if(robot->getAbsoluteMaxRotVel() < 360.000000)
  {
        printf("\nerror: AbsoluteMaxRotVel is %lf < 360.00", robot-
>getAbsoluteMaxRotVel() );
        return false;
  }
}
if(robot->getTransVelMax() < 100.000000 )
{
  printf("\nerror: TransVelMax is %lf < 100.0", robot->getTransVelMax() );
  robot->setTransVelMax(100.000000 );
  ArUtil::sleep(1000);
  if(robot->getTransVelMax() < 100.000000 )
  {
```

```
            printf("\nerror: TransVelMax is %lf < 100.0", robot->getTransVelMax() );
            return false;
          }
        }
        if(utils->isBumperPressed())
        {
          printf("\nbumpers are pressed!");
          return false;
        }
        ArUtil::sleep(100);
        return true;
}
bool HDHerding::isHumanHerding()
{
        if(!HDObjects::HUMAN_PRESENT)
          return true;
        bool retvalue;
        ArLog::logPlain(ArLog::Terse,"***isHumanHerding***");
        ObjectPose p1, p2, p3, a2, a3, h1, h2, h3;
        ArTime start;
        double anglen, anglei;
        retvalue = false;
```

```
h1 = *human ;
if(animalpose->dist == 0)
{
  //looking for animal
  *animalpose = utils->getAnimalLocation(-90, 0, false);
}
if(animalpose->dist == 0)
{
  printf("animal not found!!");
  return false;
}
if(h1.dist == 0)
{
  //looking for human
  h1 = utils->getHumanLocation(-90, 90, false);
}
if(h1.dist == 0)
  return true;
utils->pointCamera(h1.angle);
sprintf(buf,"\nh1.dist=%lf, h1.th=%lf h1.x=%lf, h1.y=%lf",  h1.dist, h1.angle,
h1.x, h1.y);
ArLog::logPlain(ArLog::Terse,buf);
```

```
sprintf(buf,"\na1.dist=%lf, animalpose->th=%lf animalpose->x=%lf, animalpose-
>y=%lf",  animalpose->dist, animalpose->angle, animalpose->x, animalpose->y);
ArLog::logPlain(ArLog::Terse,buf);
double slopei =  (h1.y - animalpose->y) /(h1.x - animalpose->x);
double xt = h1.x - animalpose->x;
double yt = h1.y - animalpose->y;
double disti = sqrt( pow(xt, 2) + pow(yt,2) );
anglei = utils->getAngle(*animalpose, h1);
double slopen;
double distn;
sprintf(buf,"\ndist=%lf anglei = %lf", disti, anglei);
ArLog::logPlain(ArLog::Terse,buf);
robot->setTransVelMax(100.000000 );
start.setToNow();
while(true)
{
  ArUtil::sleep(500);
  h2 = utils->getHumanLocation(h1.angle - 15, h1.angle + 15, false);
  utils->pointCamera(h2.angle);
  if(h2.dist == 0)
  {
        ArLog::logPlain(ArLog::Terse,"since the distance is 0, we will scan -90 to
```

```
90");
        h2 = utils->getHumanLocation(-90, 90, false);
  }
  if(h2.dist == 0)
  {
        ArLog::logPlain(ArLog::Terse,"human not found");
        retvalue = false;
        break;
  }
  slopen =  (h2.y - animalpose->y) /(h2.x - animalpose->x);
  xt = h2.x - animalpose->x;
  yt = h2.y - animalpose->y;
  distn = sqrt( pow(xt, 2) + pow(yt,2) );
  anglen = utils->getAngle(*animalpose, h2);
  if(((distn + 75) < disti) && utils->equals(anglei, anglen, 10))
  {
        //human is herding the animal
        retvalue =  true;
        break;
  }
  h1 = h2;
  slopei = slopen;
```

```
            disti = distn;
            anglei = anglen;
            if(utils->robotCannotMove())
                break;
        }
        *human = h2;
        return retvalue;
}
bool HDHerding::hasObjectMoved(ObjectPose prev_obj, ObjectPose curr_obj)
{
        if(utils->mod(prev_obj.x - curr_obj.x) < 50 && utils->mod(prev_obj.y -
        curr_obj.y) < 50)
          return false;
        return true;
}
void HDHerding::turn()
{
        *animalpose = utils->getAnimalLocation(-60, 0, false);
        utils->turnbyDelta(animalpose->angle);
        ArUtil::sleep(1000);
        *animalpose = utils->getAnimalLocation(0, 0, false);
        if(animalpose->dist == 0)
```

```
{
  *animalpose = utils->getAnimalLocation(-90, 90, false);
  if(animalpose->dist == 0)
        return;
}
ArTime start;
ObjectPose* prevPose = new ObjectPose();
ObjectPose* p1 = new ObjectPose();
ObjectPose globalpose;
ObjectPose globalPrevPose;
*prevPose = *animalpose;
*p1 = *animalpose;
double angle;
int angle2turn;
if(animalpose->angle < 0)
  angle2turn = 90 -10;
else
  angle2turn = -90 +10;
time_t t1;
start.setToNow();
while(1)
{
```

```
  if(animalpose->angle > 2 ||  animalpose->angle < -2)
        robot->setDeltaHeading(animalpose->angle);
  robot->move(animalpose->dist);
  *animalpose =  utils->getClosetObjectDistance(animalpose->angle-3,
animalpose->angle+3);
  if(!utils->equals(animalpose->dist, p1->dist, 300))
  {
        robot->setDeltaHeading(0);
        robot->move(0);
        time_t t1 = start.getMSec();
        *animalpose = utils->getAnimalLocation(p1->angle, p1->angle, false);
        if(animalpose->dist == 0)
              *animalpose = utils->getAnimalLocation(p1->angle - 15, p1->angle -
15, false);
        if(animalpose->dist == 0)
              *animalpose = utils->getAnimalLocation(p1->angle + 15, p1->angle +
15, false);
        if(animalpose->dist == 0)
              *animalpose = utils->getAnimalLocation(-90, 90, false);
        if(animalpose->dist == 0)
              break;
        start.setMSec(t1);
```

```
}
globalpose    = animalpose->getGlobalCoordinates();
if(start.mSecSince() > 1000)
{
      globalPrevPose = prevPose->getGlobalCoordinates();
      if(hasObjectMoved(globalPrevPose, globalpose) )
      {
            angle  = utils->getAngle(globalpose, globalPrevPose);
            if( (angle2turn < 0 && angle < angle2turn)
                  || (angle2turn > 0 && angle > angle2turn) )
                  break;
            *prevPose = *animalpose;
            start.setToNow();
      }
}
if(utils->robotCannotMove())
{
      printf("\n robot cannot move");
      break;
}
*p1 = *animalpose;
ArUtil::sleep(100);
```

```
        }
        robot->setDeltaHeading(0);
        robot->move(0);
        delete prevPose;
        delete p1;
}
void HDHerding::startHerding(void)
{
        ArUtil::sleep(5000);
        int i;
        *animalpose = herd->initPosition();
        if(animalpose->dist != 0 && humanAction->isHumanHerding())
        {
          herd->goTowardsAnimal();
          ArUtil::sleep(2000);
          herd->goStraight();
          herd->turn(90);
        }
        return;
        if(HDObjects::HERDING_STATUS == HERD_STRAIGHT)
        {
          if(HDAnimal::Status != ANIMAL_MOVING_STRAIGHT)
```

```
{
    switch(HDAnimal::Status)
    {
        case ANIMAL_NOT_MOVING:
            herd->goTowardsAnimal();
            break;
        case ANIMAL_MOVING_AWAY:
            utils->turnbyDelta(animalpose->angle);
            break;
    }
}
else if(HDHumanAction::STATUS != HUMAN_GO_FORWARD)
{
    switch(HDHumanAction::STATUS)
    {
        case HUMAN_GO_TOWARDS_ANIMAL:
            if(animalpose->angle > 0 )
                utils->turn(-45);
                //herd->turn(90);
            else
                utils->turn(45);
                //herd->turn(-90);
```

```
                break;
            case HUMAN_STEP_BACK:
            case HUMAN_STOPPED:
                herd->stop();
                break;

            case HUMAN_GO_AWAY_FROM_ANIMAL:
                utils->turnbyDelta(animalpose->angle);
                break;
        }
  }
}
return;
printf("\n going to init position...");
*animalpose = herd->initPosition();
if(humanAction->isHumanHerding())
{
  herd->goTowardsAnimal();
}
return;
*animalpose = utils->getAnimalLocation(-45, 30, false);
printf("\npan=%d", (HDObjects::cam)->getPan());
```

```
return;
printf("\npan=%d", HDObjects::cam->getPan());
int prevpan = HDObjects::cam->getPan() ;
int angle = -90 ;
while(angle <= 90)
{
  utils->pointCamera(angle);
  if(prevpan == angle);
  if(utils->mod(angle - prevpan) > 30)
        ArUtil::sleep(5000);
  else
        ArUtil::sleep(1500);
  if(HDObjects::vision->IsObject(&i, HDObjects::PINK_MAXHUE,
                                 HDObjects::PINK_MINHUE,
                                 HDObjects::PINK_MAXSAT,
                                 HDObjects::PINK_MINSAT,
                                 HDObjects::PINK_MAXLUM,
                                 HDObjects::PINK_MINLUM, false))
  break;
  prevpan = angle;
  angle = angle + 15;
}
```

```
        return ;
        *animalpose = utils->getAnimalLocation(-90, 0, false);
}
int main(int argc, char **argv)
{
        HDHerding* test =  new HDHerding();
        if(!test->CONNECTION_ERROR)
        {
          if(test->initSystem())
                test->startHerding();
        }
        delete test;
        return 0;
}
void HDHerding::findAnimal()
{
        bool flg =  false;
        *animalpose = utils->getAnimalLocation(-90, -90, flg );
        *animalpose = utils->getAnimalLocation(-75, -75, flg );
        *animalpose = utils->getAnimalLocation(-60, -60, flg );
        *animalpose = utils->getAnimalLocation(-45, -45, flg );
        *animalpose = utils->getAnimalLocation(-30, -30, flg );
```

```
        *animalpose = utils->getAnimalLocation(-15, -15, flg );
        *animalpose = utils->getAnimalLocation(0, 0, flg );
        *animalpose = utils->getAnimalLocation(15, 15, flg );
        *animalpose = utils->getAnimalLocation(30, 30, flg );
        *animalpose = utils->getAnimalLocation(45, 45, flg );
        *animalpose = utils->getAnimalLocation(60, 60, flg );
        *animalpose = utils->getAnimalLocation(75, 75, flg );
        *animalpose = utils->getAnimalLocation(90, 90, flg );
}
void HDHerding::findHuman()
{
        bool flg =  false;
        *animalpose = utils->getHumanLocation(-90, -90, flg );
        *animalpose = utils->getHumanLocation(-75, -75, flg );
        *animalpose = utils->getHumanLocation(-60, -60, flg );
        *animalpose = utils->getHumanLocation(-45, -45, flg );
        *animalpose = utils->getHumanLocation(-30, -30, flg );
        *animalpose = utils->getHumanLocation(-15, -15, flg );
        *animalpose = utils->getHumanLocation(0, 0, flg );
        *animalpose = utils->getHumanLocation(15, 15, flg );
        *animalpose = utils->getHumanLocation(30, 30, flg );
        *animalpose = utils->getHumanLocation(45, 45, flg );
```

```
        *animalpose = utils->getHumanLocation(60, 60, flg );
        *animalpose = utils->getHumanLocation(75, 75, flg );
        *animalpose = utils->getHumanLocation(90, 90, flg );
}
```

HDActionHuman.cpp

```
#include "ArExport.h"
#include "ariaOSDef.h"
#include "HDActionHuman.h"
#include "ArRobot.h"
/**
   @param name name of the action
   @param backOffSpeed speed at which to back away (mm/sec)
   @param backOffTime number of msec to back up for (msec)
   @param turnTime number of msec to alow for turn (msec)
*/
AREXPORT HDActionHuman::HDActionHuman(const char *name, ObjectPose
        *objPose, ArSick* sick1)
                        :ArAction(name, "Track Human")
{
        sick  = *sick1;
        prevObjPose = *objPose;
```

```
        valid =  true;
}
AREXPORT HDActionHuman::~HDActionHuman()
{
}
AREXPORT void HDActionHuman::setObjPose(ObjectPose objPose)
{
        prevObjPose = objPose;
        valid =  true;
}
AREXPORT ObjectPose HDActionHuman::getPrevObjPose()
{
        return prevObjPose;
}
AREXPORT ArActionDesired *HDActionHuman::fire(ArActionDesired currentDesired)
{
        if(valid)
        {
          myRobot->lock();
          sick.lockDevice();
          objPose.dist = sick.currentReadingPolar(prevObjPose.angle - 3,
        prevObjPose.angle + 3, &objPose.angle);
```

```
            objPose.robotpose.setPose(myRobot->getPose());
            sick.unlockDevice();
            myRobot->unlock();
            if(objPose.dist - prevObjPose.dist > 300 || objPose.dist - prevObjPose.dist < -
          300)
            {
                  objPose.dist = 0;
                  valid = false;
            }
            else
                  prevObjPose = objPose;
            return NULL;
          }
}
```

HDGoTowardsAnimal.cpp

```
#include "Aria.h"
#include "HDUtils.h"
#include "HDObjects.h"
#include "HDAnimal.h"
#include "HDGoTowardsAnimal.h"
HDGoTowardsAnimal::HDGoTowardsAnimal( )
```

```
{
        robot = HDObjects::robot;
        sick   = HDObjects::sick;
        cam         = HDObjects::cam;
        buf = new char[1000];
        utils = new HDUtils();
        animalpose =  HDAnimal::animalpose;
}
HDGoTowardsAnimal::~HDGoTowardsAnimal(void)
{
        delete []buf;
        delete utils;
}
bool HDGoTowardsAnimal::pointToAnimal()
{
        ArUtil::sleep(100);
        if(animalpose->dist == 0)
          *animalpose = utils->getAnimalLocation(-90 ,90);
        if(animalpose->dist == 0)
          return false;
        ArPose pose1 = robot->getPose();
        utils->turnbyDelta(animalpose->angle);
```

```
ArUtil::sleep(100);
*animalpose = utils->getClosetObjectDistance(-10, +10);
sprintf(buf, "\nanimalpose->dist= %lf animalpose->angle = %lf, animalpose->x=
%lf and animalpose->y = %lf", animalpose->dist, animalpose->angle,
animalpose->x, animalpose->y);
ArLog::logPlain(ArLog::Terse, buf);
if(utils->mod(animalpose->angle) > 2 )
{
  utils->turnbyDelta(animalpose->angle);
  ArUtil::sleep(100);
  *animalpose = utils->getClosetObjectDistance(-5, 5);
  if(utils->mod(animalpose->angle) > 2 )
  {
        utils->turnbyDelta(animalpose->angle);
        ArUtil::sleep(100);
        *animalpose = utils->getClosetObjectDistance(-5, 5);
        ArPose pose2 = robot->getPose();
  }
}
ArLog::logPlain(ArLog::Terse, "**********end-
pointToAnimal**********");
return true;
```

```
}
bool HDGoTowardsAnimal::animalMovedStraight(ObjectPose objpose)
{
        if(utils->mod(init_angle - objpose.angle) >  3)
          return true;
        if (objpose.robotpose.getTh() > 0 && objpose.angle < -3)
        {
          return true;
        }
        if (objpose.robotpose.getTh() < 0 && objpose.angle > 3 )
        {
          return true;
        }
        return false;
}
bool HDGoTowardsAnimal::pointToAnimal(ObjectPose objpose)
{
        utils->stopTransalationMotion();
        objpose = utils->turnbyDelta(objpose.angle);
        utils->setTranslationVelocity(100);
        return true;
}
```

```
bool HDGoTowardsAnimal::isHumanMovingTowardsAnimal(ObjectPose objpose)
{
        return false;
}
bool HDGoTowardsAnimal::goTowardsAnimal()
{
        ObjectPose objpose, prevpose;
        ArPose pose;
        double prevdist;
        if(!pointToAnimal())
        {
          HDAnimal::Status= UNKNOWN;
          return false;
        }
        utils->pointCamera(0);
        objpose = *animalpose;
        prevpose = objpose;
        HDAnimal::FlightDist = objpose.dist;
        HDAnimal::FlightAngle = objpose.angle + objpose.robotpose.getTh();
        init_angle = objpose.angle;
        utils->setTranslationVelocity(100);
        while(1)
```

```
{
  objpose = utils->getClosetObjectDistance(-5, 5);
  if(!utils->equals(prevpose.dist, objpose.dist, 700))
        objpose = utils->getClosetObjectDistance(prevpose.angle - 15,
prevpose.dist + 15);
  if(objpose.dist == 0)
        objpose = utils->getClosetObjectDistance(-90, 90);
  if(objpose.dist == 0)
        break;
  if(utils->robotCannotMove())
        break;
  if(animalMovedStraight(objpose))
        break;
  if(utils->mod(objpose.angle) > 2 )
        pointToAnimal(objpose);
  HDAnimal::FlightDist = objpose.dist ;
  HDAnimal::FlightAngle = objpose.angle + objpose.robotpose.getTh();
  prevpose = objpose;
  ArUtil::sleep(100);
}
utils->stopTransalationMotion();
*animalpose = objpose;
```

```
        return true;
}
```

HDHerdStraight.cpp

```
#include "Aria.h"
#include "HDUtils.h"
#include "HDObjects.h"
#include "HDAnimal.h"
#include "HDHerdStraight.h"
HDHerdStraight::HDHerdStraight()
{
        robot = HDObjects::robot;
        sick   = HDObjects::sick;
        cam          = HDObjects::cam;
        buf = new char[1000];
        utils = new HDUtils();
        animalpose =  HDAnimal::animalpose;
        prevanimalpose = new ObjectPose();
        humanpose = HDObjects::human;
        humanAction = new HDHumanAction();
        ANIMAL_MOVING_STRAIGHT = 0;
        ANIMAL_NOT_MOVING = 1;
```

```
        ANIMAL_MOVING_TOWARDS_ROBOT = 2;
        ANIMAL_MOVING_AWAY = 3;
        ERROR = -1;
}
HDHerdStraight::~HDHerdStraight(void)
{
        delete []buf;
        delete utils;
        delete humanAction;
        delete prevanimalpose;
}
double HDHerdStraight::getFlightAngle()
{
        if(HDAnimal::FlightAngle == 0)
        {
          if(animalpose->angle < 0)
                HDAnimal::FlightAngle =  animalpose->angle - 4;
          else
                HDAnimal::FlightAngle = animalpose->angle + 4;
        }
        if(HDAnimal::FlightDist == 0)
          HDAnimal::FlightDist = animalpose->dist;
```

```
        return HDAnimal::FlightDist * ArMath::cos(HDAnimal::FlightAngle) - 500; ;
}
ObjectPose HDHerdStraight::getAnimalPose(double angle)
{
        ObjectPose p1;
        if(animalpose->dist != 0)
        {
          angle = utils->getNewAngle(angle, 0, animalpose->angle);
          p1 = *animalpose;
          *animalpose = utils->getClosetObjectDistance(angle-3, angle+3);
          if(!utils->equals(animalpose->dist, p1.dist, 300))
                *animalpose = utils->getAnimalLocation(angle - 15, angle + 15, false);
        }
        if(animalpose->dist == 0)
          *animalpose = utils->getAnimalLocation(-90, 90);
        return *animalpose;
}
bool HDHerdStraight::moveRobot(double FlightX)
{
        if(animalpose->x - FlightX <= 0)
        {
          ArUtil::sleep(100);
```

```
    *animalpose = getAnimalPose();
    if(animalpose->x - FlightX <=  0)
        return false;
  }
  if(animalpose->x - FlightX > 0)
  {
    robot->lock();
    robot->move(animalpose->x - FlightX);
    robot->unlock();
  }
  return true;
}
int HDHerdStraight::getAnimalStatus(double y, double FlightX)
{
  int state = ANIMAL_MOVING_STRAIGHT;
  ArUtil::sleep(100);
  if( (HDAnimal::FlightAngle < 0) && (animalpose->angle <
  (HDAnimal::FlightAngle - HDObjects::ErrorAngle) )
      || (HDAnimal::FlightAngle >= 0) &&  (animalpose->angle >
  (HDAnimal::FlightAngle + HDObjects::ErrorAngle)) )
  {
    ArUtil::sleep(1000);
```

```
  *animalpose = utils->getClosetObjectDistance(animalpose->angle-3,
animalpose->angle+3);
  if( (HDAnimal::FlightAngle < 0) && (animalpose->angle <
(HDAnimal::FlightAngle - HDObjects::ErrorAngle) )
  || (HDAnimal::FlightAngle >= 0) &&  (animalpose->angle >
(HDAnimal::FlightAngle + HDObjects::ErrorAngle)) )
  {
        state = ANIMAL_NOT_MOVING;
  }
}
if(utils->mod(animalpose->y) >  y + 1000)
{
  state = ANIMAL_MOVING_AWAY;
}
if(utils->mod(animalpose->y) <  y - 1000)
{
  state = ANIMAL_MOVING_TOWARDS_ROBOT;
}
return state;
}
void HDHerdStraight::updatePoses(double angleBeforeTurn)
{
```

```
        *animalpose = getAnimalPose(angleBeforeTurn);
        humanAction->updateHumanPose(angleBeforeTurn);
}
ObjectPose HDHerdStraight::getAnimalPose()
{
        *animalpose = utils->getClosetObjectDistance(prevanimalpose->angle-3,
        prevanimalpose->angle+3);
        if(!utils->equals(animalpose->dist, prevanimalpose->dist, 300))
        {
          robot->lock();
          robot->move(0);
          robot->unlock();
          *animalpose = utils->getAnimalLocation(prevanimalpose->angle,
        prevanimalpose->angle, false);
          if(animalpose->dist == 0)
                *animalpose = utils->getAnimalLocation(prevanimalpose->angle - 15,
        prevanimalpose->angle - 15, false);
          if(animalpose->dist == 0)
                *animalpose = utils->getAnimalLocation(prevanimalpose->angle + 15,
        prevanimalpose->angle + 15, false);
          if(animalpose->dist == 0)
                *animalpose = utils->getAnimalLocation(-90, 90, false);
```

```
        }
        return *animalpose;
}
void HDHerdStraight::herdStraight()
{
        HDObjects::HERDING_STATUS = HERD_STRAIGHT;
        HDAnimal::Status = ANIMAL_MOVING_STRAIGHT;
        HDHumanAction::STATUS = HUMAN_GO_FORWARD;
        double angleBeforeTurn = robot->getPose().getTh();
        utils->turn(HDObjects::HERDING_DIRECTION);
        updatePoses(angleBeforeTurn);
        if(animalpose->dist == 0)
          return;
        double FlightX  = getFlightAngle();
        double disty = utils->mod(animalpose->y);
        humanAction->initHumanParams();
        int direction;
        *prevanimalpose = *animalpose;
        while(1)
        {
          *animalpose = getAnimalPose();
          utils->pointCamera(animalpose->angle);
```

```
if(animalpose->dist == 0)
{
	HDAnimal::Status = ANIMAL_NOT_FOUND;
	HDObjects::HERDING_STATUS = UNKNOWN;
	break;
}
if(!moveRobot(FlightX))
{
	HDAnimal::Status = ANIMAL_NOT_MOVING;
	break;
}
HDAnimal::Status = getAnimalStatus(disty, FlightX) ;
if(HDAnimal::Status != ANIMAL_MOVING_STRAIGHT)
	break;
if(HDObjects::HUMAN_PRESENT)
{
	HDHumanAction::STATUS = humanAction->getStatus();
	if(HDHumanAction::STATUS != HUMAN_GO_FORWARD)
		break;
}
if(utils->robotCannotMove())
{
```

```
                HDObjects::HERDING_STATUS = ROBOT_CANNOT_MOVE;
                break;
            }
            ArUtil::sleep(100);
            *prevanimalpose = *animalpose;
          }
          utils->stopTransalationMotion();
}
```

HDHumanAction.cpp

```
#include "Aria.h"
#include "HDUtils.h"
#include "HDObjects.h"
#include "HDAnimal.h"
#include "HDHumanAction.h"
HDHumanAction::HDHumanAction()
{
          robot = HDObjects::robot;
          buf = new char[1000];
          utils = new HDUtils();
          animalpose =  HDAnimal::animalpose;
          humanpose = HDObjects::human;
```

```
        prevHumanPose = new ObjectPose();
        prevGlobalHumanPose = new ObjectPose();
        currGlobalHumanPose = new ObjectPose();
        prevStatus = 0;
        STATUS = 0;
        ERROR = -1;
        start.setToNow();
}
HDHumanAction::~HDHumanAction(void)
{
        ArLog::logPlain(ArLog::Terse, "delete HDHumanAction" );
        delete []buf;
        delete utils;
        delete prevHumanPose;
        delete prevGlobalHumanPose;
        delete currGlobalHumanPose;
}
int HDHumanAction::STATUS = UNKNOWN;
void HDHumanAction::updateHumanPose(double angle)
{
        if(!HDObjects::HUMAN_PRESENT)
          return;
```

```
        if(humanpose->dist == 0
          || humanpose->robotpose.getX() != robot->getPose().getX()
                || humanpose->robotpose.getY() != robot->getPose().getY()
                      || humanpose->robotpose.getTh() != robot->getPose().getTh())
          *humanpose = utils->getHumanLocation(-90, 90);
        else if(humanpose->dist != 0)
        {
          angle = utils->getNewAngle(angle, 0, humanpose->angle);
          *humanpose = utils->getHumanLocation(angle, angle, false);
          if(humanpose->dist == 0)
                *humanpose = utils->getHumanLocation(angle-15, angle+15, false);
          if(humanpose->dist == 0)
                *humanpose = utils->getHumanLocation(angle + 15, angle + 15, false);
        }
        if(humanpose->dist == 0)
          *humanpose = utils->getHumanLocation(-90, 90);
}
void HDHumanAction::initHumanParams()
{
        if(HDObjects::HUMAN_PRESENT)
        {
          *prevHumanPose = *humanpose;
```

```
            disty = humanpose->y;
        }
}
ObjectPose HDHumanAction::getHumanPose()
{
        *humanpose = utils->getClosetObjectDistance(prevHumanPose->angle-3,
        prevHumanPose->angle+3);
        if(!utils->equals(humanpose->dist, prevHumanPose->dist, 300) || humanpose-
        >dist == 0)
        {
          robot->lock();
          robot->move(0);
          robot->unlock();
          *humanpose = utils->getHumanLocation(prevHumanPose->angle,
        prevHumanPose->angle);
          if(humanpose->dist == 0)
                *humanpose = utils->getHumanLocation(prevHumanPose->angle-15,
        prevHumanPose->angle-15);
          if(humanpose->dist == 0)
                *humanpose = utils->getHumanLocation(prevHumanPose->angle+15,
        prevHumanPose->angle+15);
          if(humanpose->dist == 0)
```

```
            *humanpose = utils->getHumanLocation(-90, 90);
        }
        return *humanpose;
}
void HDHumanAction::getPoses()
{
        if(animalpose->dist == 0)
        {
          printf("\n looking for animal");
          *animalpose = utils->getAnimalLocation(-90, 90, false);
        }
        if(humanpose->dist == 0 ||  humanpose->robotpose.getX() != robot-
        >getPose().getX()
                                    || humanpose->robotpose.getY() != robot-
        >getPose().getY()
                                    || humanpose->robotpose.getTh() != robot-
        >getPose().getTh() )
        {
          *humanpose = utils->getHumanLocation(-90, 90, false);
        }
}
bool HDHumanAction::isHumanHerding()
```

```
{
        if(!HDObjects::HUMAN_PRESENT)
          return true;
        bool retvalue = false;
        time_t t1;
        ArTime start;
        double init_angle, curr_angle;
        ObjectPose p1;
        getPoses();
        if(animalpose->dist == 0 || humanpose->dist == 0)
          return false;
        utils->pointCamera(humanpose->angle);
        *prevHumanPose = *humanpose;
        init_angle = utils->getAngle(*humanpose, *animalpose);
        double xt = humanpose->x - animalpose->x;
        double yt = humanpose->y - animalpose->y;
        double init_dist = sqrt( pow(xt, 2) + pow(yt,2) );
        double curr_dist;
        start.setToNow();
        while(1)
        {
          *humanpose = utils->getClosetObjectDistance(prevHumanPose->angle-3,
```

```
prevHumanPose->angle+3);
  if(!utils->equals(humanpose->dist, prevHumanPose->dist, 300))
  {
        time_t t1 = start.getMSec();
        lookforhuman(prevHumanPose);
        if(humanpose->dist == 0)
              break;
        start.setMSec(t1);
  }
  if(start.mSecSince() > 5000)
  {
        xt = humanpose->x - animalpose->x;
        yt = humanpose->y - animalpose->y;
        curr_dist = sqrt( pow(xt, 2) + pow(yt,2) );
        curr_angle = utils->getAngle(*humanpose, *animalpose);
        if(curr_dist < (init_dist - 75) && utils->equals(curr_angle, init_angle, 10))
        {
              retvalue = true;
              break;
        }
        if(curr_dist > (init_dist - 75))
        {
```

```
                init_angle = curr_angle;
                init_dist = curr_dist;
            }
            start.setToNow();
        }
        if(utils->robotCannotMove())
        {
            break;
        }
        *prevHumanPose = *humanpose;
        ArUtil::sleep(100);
      }
}
void HDHumanAction::lookforhuman(ObjectPose* p1)
{
      *humanpose = utils->getHumanLocation(p1->angle, p1->angle, false);
      if(humanpose->dist == 0)
        *humanpose = utils->getHumanLocation(p1->angle - 15, p1->angle - 15, false);
      if(humanpose->dist == 0)
        *humanpose = utils->getHumanLocation(p1->angle + 15, p1->angle + 15,
      false);
      if(humanpose->dist == 0)
```

```
            *humanpose = utils->getHumanLocation(-90, 90, false);
}
int HDHumanAction::getStatus()
{
          if(!HDObjects::HUMAN_PRESENT)
            return 0;
          *humanpose = getHumanPose();
          STATUS = HUMAN_GO_FORWARD;
          if(humanpose->dist == 0 )
          {
            STATUS = HUMAN_NOT_FOUND;
          }
          else if(utils->mod(humanpose->y) - utils->mod(disty) > 300)
          {
            STATUS = HUMAN_GO_AWAY_FROM_ANIMAL;
          }
          else if(utils->mod(disty) - utils->mod(humanpose->y) > 300)
          {
            STATUS = HUMAN_GO_TOWARDS_ANIMAL;
          }
          else if(utils->mod(humanpose->angle) >  85)
          {
```

```
            STATUS = HUMAN_STOPPED;
        }
        *prevHumanPose =  *humanpose;
        return STATUS;
}
```

HDInitPosition.cpp

```
#include "Aria.h"
#include "HDUtils.h"
#include "HDObjects.h"
#include "HDInitPosition.h"
#include "HDAnimal.h"
HDInitPosition::HDInitPosition( )
{
        robot = HDObjects::robot;
        sick   = HDObjects::sick;
        cam          = HDObjects::cam;
        animalpose = HDAnimal::animalpose;
        buf = new char[1000];
        utils = new HDUtils();
}
HDInitPosition::~HDInitPosition(void)
```

```
{
        delete []buf;
        delete utils;
}
void HDInitPosition::updateAnimalPose(double angle)
{
        ObjectPose prevpose = *animalpose;
        *animalpose = utils->getClosetObjectDistance(angle-3, angle+3);
        if(!utils->equals(animalpose->dist, prevpose.dist , 300))
          *animalpose = utils->getAnimalLocation(angle, angle , false);
        if(animalpose->dist == 0)
          *animalpose = utils->getAnimalLocation(angle-15, angle-15 , false);
        if(animalpose->dist == 0)
          *animalpose = utils->getAnimalLocation(angle+15, angle+15 , false);
        if(animalpose->dist == 0)
          *animalpose = utils->getAnimalLocation(-90, 90 , false);
}
void HDInitPosition::goToInitPostion()
{
        if(animalpose->dist == 0)
          *animalpose = utils->getAnimalLocation(-90, 90, false);
        if(animalpose->dist == 0)
```

```
  return ;
ArPose pose = robot->getPose();
double mod_angle = utils->mod(animalpose->angle );
if( mod_angle > HDAnimal::MinAngleofApproach && mod_angle <
HDAnimal::MaxAngleofApproach)
  return;
double max_x = HDAnimal::MaxFlightDist*
ArMath::sin(HDAnimal::AngleofApproach);
double max_y = HDAnimal::MaxFlightDist*
ArMath::sin(HDAnimal::AngleofApproach);
if(utils->mod(animalpose->x) > max_x || utils->mod(animalpose->y) > max_y )
{
  double x = animalpose->x - max_x;
  double y, th;
  if(animalpose->y > 0)
        y = animalpose->y - max_y;
  else
        y = animalpose->y + max_y;
  th = ArMath::radToDeg(atan(y/x));
  if( animalpose->y > max_y && animalpose->x < max_x )
  {
        if(animalpose->y > 0)
```

```
                th = 90 - th;
            else
                th = -90 - th;
          }
          utils->turnbyDelta(th);
          utils->goForward(sqrt(x*x + y*y) + robot->getRobotRadius());
        }
        else
        {
          double angle, angletoturn;
          if(animalpose->angle < 0 )
                angletoturn = -90;
          else
                angletoturn = 90;
          utils->turn(angletoturn);
          angle = animalpose->angle - angletoturn;
          updateAnimalPose(angle);
          if(animalpose->dist != 0)
                trackAndMoveTill45(animalpose);
        }
}
bool HDInitPosition::trackAndMoveTill45(ObjectPose *opose)
```

```
{
        int direction = 1;
        double prevdist;
        double angle = opose->angle;
        bool ret_val = false;
        if(utils->mod(angle) > HDAnimal::AngleofApproach)
          direction = -1 ;
        utils->setTranslationVelocity(direction * 100);
        prevdist = opose->dist;
        ObjectPose objpose;
        while (1)
        {
          objpose = utils->getClosetObjectDistance(angle - 3, angle + 3 );
          if(utils->mod(prevdist - objpose.dist ) > 1000 )
          {
                utils->setTranslationVelocity(0);
                objpose = utils->getAnimalLocation(angle, angle);
                if(objpose.dist == 0)
                      objpose = utils->getAnimalLocation(-90, 90);
                if(objpose.dist == 0)
                      break;
                utils->setTranslationVelocity(direction * 100);
```

```
        }
        if( utils->mod(objpose.angle) < 47 && utils->mod(objpose.angle) > 43)
        {
            ret_val = true;
            break;
        }
        angle = objpose.angle;
        utils->pointCamera(angle);
        if(utils->robotCannotMove())
            break;
        ArUtil::sleep(100);
    }
    utils->stopTransalationMotion();
    opose->x = objpose.x;
    opose->y = objpose.y;
    opose->angle = objpose.angle;
    opose->dist = objpose.dist;
    return ret_val;
}
```

HDAnimal.cpp

```
#include "HDAnimal.h"
```

```
int HDAnimal::MinFlightDist = 900;

int HDAnimal::MaxFlightDist = 10000;

int HDAnimal::AngleofApproach = 45;

int HDAnimal::MinAngleofApproach = 30;

int HDAnimal::MaxAngleofApproach = 60;

ObjectPose *HDAnimal::animalpose = new ObjectPose();

double HDAnimal::FlightDist;

double HDAnimal::FlightAngle;

int HDAnimal::hooktime = 5000;

int HDAnimal::Status = UNKNOWN;
```

HDBumpers.cpp

```
#include "Aria.h"

#include "HDObjects.h"

#include "HDBumpers.h"

HDBumpers::HDBumpers()

{

}

HDBumpers::~HDBumpers()

{

}

bool HDBumpers::isPressed()

{
```

```
        int myBumpMask = (ArUtil::BIT1 | ArUtil::BIT2 | ArUtil::BIT3 | ArUtil::BIT4 |
        ArUtil::BIT5);
        return (HDObjects::robot->getStallValue() & 0xff) & myBumpMask;
}
```

HDObjects.cpp

```
#include "HDObjects.h"
#include "HDAnimal.h"
ArRobot*     HDObjects::robot= NULL;
ArSick*      HDObjects::sick       = NULL;
ArSonyPTZ*HDObjects::cam       = NULL;
HDVision*   HDObjects::vision     = NULL;
ObjectPose*  HDObjects::human    = NULL;
HDBumpers*     HDObjects::bumpers  = NULL;
int HDObjects::ErrorAngle      = 3;
int HDObjects::ErrorDist   = 100;
int HDObjects::PINK_MAXHUE = 242;
int HDObjects::PINK_MINHUE = 222;
int HDObjects::PINK_MAXSAT = 250;
int HDObjects::PINK_MINSAT = 225;
int HDObjects::PINK_MAXLUM = 200;
int HDObjects::PINK_MINLUM = 175;
```

```
int HDObjects::ORANGE_MAXHUE = 15;
int HDObjects::ORANGE_MINHUE = 10;
int HDObjects::ORANGE_MAXSAT = 220;
int HDObjects::ORANGE_MINSAT = 115;
int HDObjects::ORANGE_MAXLUM = 155;
int HDObjects::ORANGE_MINLUM = 125;
int HDObjects::PINK = 0;
int HDObjects::ORANGE = 1;
int HDObjects::ANIMAL_COLOR = ORANGE;
int HDObjects::HUMAN_COLOR = PINK;
bool HDObjects::HUMAN_PRESENT = true;
int HDObjects::HERDING_DIRECTION = 0;
int HDObjects::HERDING_STATUS = HERD_STRAIGHT;
void HDObjects::destroyObjects()
{
        delete robot;
        delete vision;
        delete human;
        delete bumpers;
}
void HDObjects::createObjects()
{
```

```
        robot = new ArRobot();
        sick   = new ArSick();
        cam        = new ArSonyPTZ(robot);
        vision     = new HDVision();
        human      = new ObjectPose();
        bumpers   = new HDBumpers();
}
```

HDUtils.cpp

```
#include "HDUtils.h"
#include "HDObjects.h"
#include "HDAnimal.h"
#include "HDObjects.h"
#include "HDBumpers.h"
HDUtils::HDUtils()
{
        robot = HDObjects::robot;
        sick   = HDObjects::sick;
        cam        = HDObjects::cam;
        buf = new char[1000];
        bumpers = HDObjects::bumpers;
        vision = HDObjects::vision;
```

```
}
HDUtils::~HDUtils()
{
        delete []buf;
}
void HDUtils::pointCamera(double angle)
{
        if(cam != NULL)
          cam->pan(-1.0* angle);
}
void HDUtils::stopTransalationMotion()
{
        robot->lock();
        robot->setVel(0);
        robot->unlock();
}
void HDUtils::stopRotationalMotion()
{
        robot->lock();
        robot->setRotVel(0);
        robot->setRotVelMax(200.000000);
        robot->unlock();
```

```
}
bool HDUtils::cannotMoveForward()
{
        ObjectPose pose = getClosetObjectDistance(-45, 45);
        if(pose.dist <= HDAnimal::MinFlightDist )
          return true;
        return false;
}
void HDUtils::setTranslationVelocity(double velocity)
{
        robot->lock();
        robot->setVel(velocity);
        robot->unlock();
}
bool HDUtils::isBumperPressed()
{
        return bumpers->isPressed();
}
ObjectPose HDUtils::getClosetObjectDistance()
{
        ObjectPose objPose;
        ArPose pose;
```

```
        while(1)
        {
          robot->lock();
          sick->lockDevice();
          objPose.dist = sick->currentReadingPolar(-90, 90, &objPose.angle);
          pose = robot->getPose();
          objPose.x = ArMath::cos(objPose.angle) * objPose.dist ;
          objPose.y = ArMath::sin(objPose.angle) * objPose.dist ;
          objPose.robotpose.setPose(pose);
          sick->unlockDevice();
          robot->unlock();
          if(objPose.dist < sick->getMaxRange() &&  objPose.dist > sick-
        >getMinRange())
                break;
        }
        return objPose;
}
ObjectPose HDUtils::getClosetObjectDistance(double beginangle, double endangle)
{
        ObjectPose objPose;
        ArPose pose;
        if(beginangle > endangle)
```

```
{
  double temp = endangle;
  endangle =    beginangle;
  beginangle = temp;
}
if(beginangle < - 90 )
  beginangle = -90;
if(endangle >  90 )
  endangle = 90;
while(1)
{
  robot->lock();
  sick->lockDevice();
  objPose.dist = sick->currentReadingPolar(beginangle, endangle,
&objPose.angle);
  pose.setPose(robot->getPose());
  sick->unlockDevice();
  robot->unlock();
  objPose.x = ArMath::cos(objPose.angle) * objPose.dist ;
  objPose.y = ArMath::sin(objPose.angle) * objPose.dist ;
  objPose.robotpose.setPose(pose);
  if(objPose.dist < sick->getMaxRange() &&  objPose.dist > sick-
```

```
        >getMinRange())
                break;
        }
        return objPose;
}
void HDUtils::goForward(double dist)
{
        robot->lock();
        robot->move(dist);
        robot->unlock();
        while (1)
        {
          robot->lock();
          if (robot->isMoveDone())
          {
                robot->unlock();
                break;
          }
          robot->unlock();
          ArUtil::sleep(100);
        }
}
```

```
//returns the distance moved
double HDUtils::move(double dist)
{
        ArPose initpose  = robot->getPose();;
        if(dist > 0)
        {
          double distancetoGo, objectDist;
          double keepdistance = robot->getRobotRadius() + 100;
          if(dist >= keepdistance)
          {
                objectDist = sick->currentReadingPolar(-90, 90);
                if(dist <= objectDist )
                      distancetoGo = dist;
                else
                      distancetoGo = objectDist;
                robot->move(distancetoGo);
                while (objectDist > keepdistance )
                {
                      objectDist = sick->currentReadingPolar(-90, 90);
                      robot->lock();
                      if(robot->isMoveDone())
                      {
```

```
                    robot->unlock();
                    break;
                }
                robot->unlock();
                ArUtil::sleep(100);
            }
        }
      }
      else //move backwards
      {
        int rearBump ;
        robot->move(dist);
        while(!robot->isMoveDone())
        {
            if(isBumperPressed())
                break;
        }
      }
      ArPose currpose = robot->getPose();
      return initpose.findDistanceTo(currpose);
}
void HDUtils::turnLeft()
```

```
{
        turn(90);
}
void HDUtils::turnRight()
{
        turn(-90);
}
void HDUtils::turnLeft(double angle)
{
        turn(angle);
}
void HDUtils::turnRight(double angle)
{
        if(angle < 0)
          angle = angle * -1;
        turn(angle);
}
void HDUtils::turnApproximate(double angle)
{
        robot->lock();
        robot->setRotVelMax(200.000000);
        robot->setHeading(angle);
```

```
        robot->unlock();
        while (1)
        {
          robot->lock();
          if (robot->isHeadingDone(0))
          {
                robot->unlock();
                break;
          }
          robot->unlock();
          ArUtil::sleep(100);
        }
}
double HDUtils::mod(double no)
{
        if(no < 0)
          no = -1 * no;
        return no;
}
bool HDUtils::equals(double x, double y)
{
        if ( (mod(x) > mod(y) - HDObjects::ErrorDist) &&  (mod(x) < mod(y) +
```

```
        HDObjects::ErrorDist) )
          return true;
        return false;
}
bool HDUtils::equals(double x, double y, double diff)
{
        if ( mod(x) > mod(y) - diff &&  mod(x) < mod(y) + diff)
          return true;
        return false;
}
void HDUtils::turnbyDeltaApproximate(double angle)
{
        robot->lock();
        robot->setDeltaHeading(angle);
        robot->unlock();
        while (1)
        {
          robot->lock();
          if (robot->isHeadingDone(0))
          {
                robot->unlock();
                break;
```

```
            }
            robot->unlock();
            ArUtil::sleep(100);
          }
}
bool HDUtils::loadWorld(char * wld)
{
        return robot->comStr(ArCommands::LOADWORLD, wld) ;
}
ObjectPose HDUtils::pointToObject(double beginangle, double  endangle)
{
        ObjectPose objPose = getClosetObjectDistance(beginangle, endangle);
        objPose = turnbyDelta(objPose.angle);
        return objPose;
}
bool HDUtils::trackAndMoveTill45(double angle, double distToMove, ObjectPose *
        newObjPose)
{
        printf("\n ***trackAndMoveTill45***");
        ObjectPose objpose;
        objpose = getClosetObjectDistance(angle - 20, angle + 20);
        angle = objpose.angle;
```

```
bool ret_val;
if(angle < 0 && angle > -45 && distToMove < 0)
  distToMove = -1 * distToMove;
if(angle < -45 && distToMove > 0)
  distToMove = -1 * distToMove;
if(angle > 45 && distToMove > 0)
  distToMove = -1 * distToMove;
if(angle > 0 && angle < 45 && distToMove < 0)
  distToMove = -1 * distToMove;
robot->lock();
robot->move(distToMove);
robot->unlock();
pointCamera(angle);
while (1)
{
  robot->lock();
  if (robot->isMoveDone())
  {
        robot->unlock();
        ret_val = false;
        break;
  }
```

```
  robot->unlock();
  objpose = getClosetObjectDistance(angle - 3, angle + 3 );
  if( mod(objpose.angle) < 47 && mod(objpose.angle) > 43)
  {
        ret_val = true;
        break;
  }
  angle = objpose.angle;
  pointCamera(angle);
  if(isBumperPressed())
  {
        ret_val = true;
        break;
  }
  ArUtil::sleep(100);
}
stopTransalationMotion();
if(newObjPose != NULL)
{
  newObjPose->angle = objpose.angle;
  newObjPose->dist = objpose.dist;
  newObjPose->x = objpose.x;
```

```
    newObjPose->y = objpose.y;
  }
  return ret_val;
}
void HDUtils::turn(double angle)
{
  ArRobot robot1;
  double currPose  = robot->getPose().getTh();
  double finalPose = angle;
  if(currPose > finalPose -1 && currPose < finalPose + 1)
    return;
  ArPose pose;
  turnApproximate(finalPose);
  pose = robot->getPose();
  if(pose.getTh() < finalPose  - 1 || pose.getTh() > finalPose  + 1)
  {
    double temp;
    if(pose.getTh() > finalPose )
        temp = finalPose-5;
    else
        temp = finalPose+5;
    robot->lock();
```

```
robot->setRotVelMax(2);
robot->setHeading(temp);
robot->unlock();
currPose  = robot->getPose().getTh();
while(1)
{
	robot->lock();
	pose = robot->getPose();
	robot->unlock();
	robot->lock();
	if (robot->isHeadingDone(0))
	{
		robot->unlock();
		break;
	}
	robot->unlock();
	if( (currPose > finalPose && pose.getTh() - finalPose < 1)
		||  (currPose < finalPose && pose.getTh() - finalPose > -1) )
	{
		stopRotationalMotion();
		break;
	}
```

```
            if(isBumperPressed())
            {
                stopRotationalMotion();
                break;
            }
            ArUtil::sleep(100);
        }
    }
    robot->lock();
    robot->setRotVelMax(200.000000);
    robot->unlock();
}

ObjectPose HDUtils::turnbyDelta(double angle)
{
    double finalPose = robot->getPose().getTh() + angle;
    turn(finalPose);
    ArUtil::sleep(1000);
    return getClosetObjectDistance(-3, 3);
}
bool HDUtils::robotCannotMove(/*double dist*/)
{
```

```
        if(isBumperPressed())
        {
          stopTransalationMotion();
          return true;
        }
        if(cannotMoveForward())
        {
          stopTransalationMotion();
          return true;
        }
        return false;
}
void HDUtils::display(ObjectPose objpose)
{
        robot->lock();
        ArPose pose = robot->getPose();
        robot->unlock();
}
void HDUtils::log(ObjectPose objpose)
{
        robot->lock();
        ArPose pose = robot->getPose();
```

```
        robot->unlock();
}

ObjectPose HDUtils::getAnimalLocation(int begin_angle, int end_angle,bool display )
{
        ArLog::logPlain(ArLog::Terse, "\n looking for animal");
        if (HDObjects::ANIMAL_COLOR == HDObjects::PINK)
        {
          return getObjectLocation(begin_angle, end_angle,
                                   HDObjects::PINK_MAXHUE,
                                   HDObjects::PINK_MINHUE,
                                   HDObjects::PINK_MAXSAT,
                                   HDObjects::PINK_MINSAT,
                                   HDObjects::PINK_MAXLUM,
                                   HDObjects::PINK_MINLUM, display );
        }
        else
        {
          return getObjectLocation(begin_angle, end_angle,
                                   HDObjects::ORANGE_MAXHUE,
                                   HDObjects::ORANGE_MINHUE,
                                   HDObjects::ORANGE_MAXSAT,
```

```
                                HDObjects::ORANGE_MINSAT,
                                HDObjects::ORANGE_MAXLUM,
                                HDObjects::ORANGE_MINLUM, display );
        }
}
ObjectPose HDUtils::getHumanLocation(int begin_angle, int end_angle, bool display )
{
        ArLog::logPlain(ArLog::Terse, "\n looking for human");
        if (HDObjects::HUMAN_COLOR == HDObjects::ORANGE)
        {
          return getObjectLocation(begin_angle, end_angle,
                                HDObjects::ORANGE_MAXHUE,
                                HDObjects::ORANGE_MINHUE,
                                HDObjects::ORANGE_MAXSAT,
                                HDObjects::ORANGE_MINSAT,
                                HDObjects::ORANGE_MAXLUM,
                                HDObjects::ORANGE_MINLUM, display );
        }
        else
        {
          return getObjectLocation(begin_angle, end_angle,
                                HDObjects::PINK_MAXHUE,
```

```
                                HDObjects::PINK_MINHUE,
                                HDObjects::PINK_MAXSAT,
                                HDObjects::PINK_MINSAT,
                                HDObjects::PINK_MAXLUM,
                                HDObjects::PINK_MINLUM, display);
        }
}
ObjectPose HDUtils::getObjectLocation(int begin_angle, int end_angle, double
        max_hue, double min_hue, double max_sat, double min_sat, double max_lum,
        double min_lum, bool display )
{
        int col = -9999;
        ObjectPose objpose;
        bool b1;
        if(begin_angle > end_angle)
        {
          int temp_angle = end_angle;
          end_angle = begin_angle;
          begin_angle = temp_angle;
        }
        if(begin_angle < -90)
          begin_angle = -90;
```

```
if(end_angle > 90)
  end_angle = 90;
int prevpan = cam->getPan() ;
int angle = begin_angle ;
while(angle <= end_angle)
{
  pointCamera(angle);
  if(prevpan == angle);
  else if(mod(prevpan - angle) > 30 )
        ArUtil::sleep(5000);
  else
        ArUtil::sleep(2000);
  b1 = vision->IsObject(&col,max_hue, min_hue, max_sat, min_sat, max_lum,
min_lum,display);
  if(b1 == true)
        break;
  prevpan = angle;
  angle = angle + 15;
}
if(col != -9999)
{
  angle = angle + 20 - (40*col/320);
```

```
    objpose = getClosetObjectDistance(angle - 10, angle + 10);
  }
  else
  {
    objpose.dist = 0;
    ArLog::logPlain(ArLog::Terse, "object not found");
  }
  return objpose;
}
double HDUtils::getNewAngle(double fromangle, double toangle, double objangle)
{
  return fromangle + objangle - toangle;
}
double HDUtils::getAngle(ObjectPose prev, ObjectPose curr)
{
  double x = (prev.x - curr.x);
  double y = (prev.y - curr.y);
  if(x == 0)
    return 0;
  double angle = ArMath::radToDeg(atan(y/x));
  return angle;
}
```

HDVision.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "HDVision.h"
HDVision::HDVision()
{
        img_index = 0;
        visionFile = fopen("vision.txt", "w");
}
extern "C" HDVision::~HDVision()
{
        if (visionFile != NULL)
          fclose(visionFile);
}
extern "C" bool HDVision::IsObject()
{
        if(ERROR)
          return false;
        if(img == NULL)
          printf("\n img == null");
```

```
        else
          printf("\n img != null");
        vlGrabImage (img);
        if (0 > vlImageSave (img, "img3.ppm"))
          printf ("unable to save image\n");
        return false;
}
extern "C" bool HDVision::IsObject(int* column, double max_hue, double min_hue,
        double max_sat, double min_sat, double max_lum, double min_lum, bool
        display1)
{
        img = VL_IMAGE_CREATE();
        point = VL_POINT_CREATE();
        if (0 > vlGrabInit (NULL, 0, 0, 0, 1))
        {
          VL_ERROR ("vlInit: error: unable to initialize framegrabber\n");
        }
        int index;
        int r, g, b;
        max_h = max_hue;
        min_h = min_hue;
        max_s = max_sat;
```

```
min_s = min_sat;
max_l = max_lum;
min_l = min_lum;
vlGrabImage (img);
if(display1 ==  true)
{
  if (0 > vlDisplayInit (VL_COLS_DEFAULT, VL_ROWS_DEFAULT,
FALSE))
  {
        VL_ERROR ("vlInit: error: unable to setup display\n");
        return (-1);
  }
  while (1)
  {
        vlGrabImage (img);
        vlShow (img, 0);
        if (vlEvent() && vlButtonPress())
          break;
  }
  vlGetPoint (point);
  printf("\n point clicked on: x:%d, y:%d", point->x, point->y);
  index = (point->y * img->width *3) + (point->x *3);
```

```
  r = img->pixel[index++];
  g = img->pixel[index++];
  b = img->pixel[index++];
  HSL hsl = getHSL(r,g,b);
}
else
  vlGrabImage (img);
char buf[100];
if (0 > vlImageSave (img, buf))
  printf ("unable to save image\n");
int i, j;
index = 0;
int COUNT = 0;
for (i=0; i < img->height; i++)
{
  for (j=0; j < img->width; j++)
  {
        r = img->pixel[index];
        g = img->pixel[index+1];
        b = img->pixel[index+2];
        index = index +3 ;
        if(isWithinRange(r,g, b))
```

```
            {
                fprintf(visionFile,"\ni = %d, j = %d", i, j);
                ++COUNT;
                if(COUNT == 5)
                    *column = j;
                if(COUNT == 10 )
                    break;
            }
          }
          if(COUNT == 10 )
              break;
        }
        vlImageDestroy(img);
        vlPointDestroy(point);
        vlShutdown();
        if(COUNT > 0)
          fprintf(visionFile, "\n count = %d", COUNT);
        if(COUNT >=10)
          return true;
        return false;
}
extern "C" bool HDVision::isWithinRange(int r, int g, int b)
```

```
{
        HSL hsl = getHSL(r,g,b);
        int count = 0;
        if(hsl.hue <= max_h && hsl.hue >= min_h)
        {
          if(hsl.sat <= max_s && hsl.sat >= min_s)
          {
                if(hsl.luminosity <= max_l && hsl.luminosity >= min_l)
                {
                        fprintf(visionFile, "\n r = %d, g = %d, b = %d", r, g, b);
                        fprintf(visionFile, "\n hsl.hue=%lf hsl.sat=%lf hsl.luminosity= %lf",
        hsl.hue,hsl.sat, hsl.luminosity);
                        return true;
                }
          }
        }
        return false;
}
extern "C" HSL HDVision::getHSL(int red, int green, int blue)
{
        double r = red/255.0;
        double g = green/255.0;
```

```
double b = blue/255.0;
double min = r;
double max = r;
if(b > r && b > g)
  max=b;
if(g > r && g >= b)
  max=g;
if(b < r && b <= g)
  min = b;
if(g < r && g < b)
  min=g;
double L = (min + max)/2.0;
double S = 0;
double H = 0;
if(L < 0.5)
  S = (max-min)/(max+min);
else
  S = (max-min)/(2.0-max-min);
double h1 = 0.0;
if(max == min)
  h1=160;
else
```

```
        {
          if(r == max)
                H = (g-b)/(max-min);
          else if(g == max)
                H = 2.0 + (b-r)/(max-min);
          else if(b == max)
                H = 4.0 + (r-g)/(max-min);
          h1 = (H*239)/6;
          if(h1 < 0)
                h1 = h1 + 240;
        }
        double hue = h1;
        double sat =S * 239;
        double lum = L * 239;
        HSL hsl;
        hsl.hue = hue;
        hsl.sat = sat;
        hsl.luminosity = lum;
        return hsl;
}
extern "C" void HDVision::setDisplay(bool display_1)
{
```

```
        if(display1 != display_1)
          display1 = display_1;
        if(!initialize_display && display_1)
        {
          if (0 > vlDisplayInit (VL_COLS_DEFAULT, VL_ROWS_DEFAULT,
        FALSE))
          {
                VL_ERROR ("vlInit: error: unable to setup display\n");
                ERROR = true;
                return;
          }
          initialize_display =  true;
          vlGrabImage (img);
          vlShow (img, 0);
        }
}
```