SheepLog:
Creating a Prolog framework to herd sheep

Author:
Frank Dorssers (s0824704)
frank.dorssers@gmail.com

Supervisor/assessor:
Dr. ir. Martijn van Otterlo
M.vanOtterlo@donders.ru.nl

May, 2014
Abstract

A predicate based modular Prolog framework for shepherding, called SheepLog, is implemented in this thesis. Input for the framework consists of the current locations of the sheep, the herding dog and the goal. It uses this information to control the behavior of a herding dog. Several existing behaviors have been implemented in SheepLog. The framework works as an extension to an existing Java-based herding simulator which contains a visual representation of the sheep and herding dog, a parameter window, the behavior for the sheep and several herding behaviors. Two types of experiments have been conducted, some dealing with performance (real world time and iterations) and others with ease of (re)use. In terms of iterations the SheepLog implementations perform about the same as the Java implementations depending on the behavior. In terms of real world time the SheepLog implementations are about a factor 30 slower than the Java implementations. The ease of use test shows that it is possible to reuse existing predicates and add new ones to create novel behaviors. Overall this thesis contributes a novel, high-level shepherding language which can be employed for the implementation of generic shepherding behaviors.
## CONTENTS

### 4 Experiments

4.1 Comparing performance .................................................. 37  
4.1.1 Simulator settings ...................................................... 37  
4.1.2 Miki algorithm .......................................................... 38  
4.1.3 Lien algorithm .......................................................... 40  
4.1.4 Comparing the SheepLog implementations .......................... 41  

4.2 Implementing new behaviors .............................................. 42  
4.2.1 Pushing sheep one by one ............................................. 42  
4.2.2 Forming a flock by circling around all sheep ....................... 43  
4.2.3 Gathering sheep by picking the furthest sheep .................... 44

### 5 Conclusion

Appendices

A Overview ................................................................. 55  
B Setters ................................................................. 61  
C Getters ................................................................. 65  
D Checker ................................................................. 81  
E Helper ................................................................. 85  
F Behaviours .............................................................. 95  
G Java Info ............................................................... 101  
H Targets ................................................................. 105  
I Movers ................................................................. 109
Chapter 1

Introduction

In this thesis we will be looking at shepherding. We will focus on how shepherding can be simulated on a computer using a self-created modular framework. We will look at what exactly a modular framework is and on top of that we will discuss how a modular framework can help us in this problem, but first we will continue with a general introduction. There are three elements to shepherding, these are the shepherd, his dog and sheep, meaning it is a multi-agent system. These three can already be simulated using an existing simulator which the framework will extend. When we want to represent this information in the framework we need to know how we are going to model these participants of shepherding and how we are going to represent these participants visually.

Miki and Nakamura [5], Lien et al. [3], Vaughan et al. [9] and Bennet and Trafankowski [1] are researchers who have their own ideas as to how exactly shepherding can be modeled as algorithms. A few of their ideas can be seen in section 1.2.3. In their papers they all have their own way of implementing and describing the behavior, making it hard to immediately compare these behaviors and interchanging implementations.

Interchangeability is important when wanting to compare different algorithms. This is one of the issues that we will try to solve in this thesis. Other issues are understandability and ease-of-use. The goal is to create a herding framework which allows one to easily implement new behaviors.

To achieve this goal the framework that extends the simulator will be created in Prolog and consist of higher level predicates. Having higher level predicates allows you to use these as building blocks when creating new behaviors, this makes it possible to replace one building block with another building block to change and create behaviors.

Using these predicates has advantages as well as disadvantages. One of the disadvantages will be performance. Using high level predicates does not give you the same amount of possibilities when optimizing code without having to rewrite predicates for your specific behavior. While it will most likely negatively impact performance, it will have several advantages, for example the ability to easily create new behaviors by combining existing predicates.

1.1 What is shepherding

People have held sheep for a long time. Often for various goals, for example their meat, milk or wool. These sheep were often held in large packs which can be difficult to control, this is where shepherding comes into play. Shepherding is often not the actual goal, but more of a means to an end. It can be used to move large herds of sheep around, for example from one pasture to
another to prevent overgrazing. There are also shepherding competitions where shepherding is the actual goal. Shepherds go here to measure their skills against other shepherds. An example of a competition is the World Sheep Dog Trials [7].

We now know why shepherding is used, but not yet exactly what it is. There are three key components to shepherding, the shepherd, his dog and the sheep. Figure 1.1 shows visualizations for different situations for shepherding, starting with figure 1.1a which shows what the basic area looks like. Shepherding is not done by the shepherd alone, he or she uses a herding dog to control the sheep.

First we will take a look at the sheep. Sheep are animals that are kept in flocks and their behavior can be described relatively easily. Sheep tend to form flocks while grazing and moving away from any threats, for example dogs and the shepherd. In this implementation the sheep will gravitate towards each other if they are within a certain distance of each other. A representation of a possible situation can be seen in figure 1.1b where the arrows represent the sheep.

Let us introduce a herding dog into this situation as seen in figure 1.1c. At first nothing happens. But when he moves closer, as can be seen in figure 1.1d, you can see that the sheep have reacted and are moving away from the dog. Let us now introduce the shepherd himself as can be seen in figure 1.1e. The sheep now move away from the dog as well as the shepherd. Using this behavior a shepherd can move entire flocks of sheep using his herding dog and himself.

A herding dog does not automatically know what he should do. Herding dogs are trained from the moment that they leave their mother’s side. Many books explain in detail how to raise a herding dog and how to work with them, for example Training the Sheep Dog [4] and Sheepdog Training and Trials [10]. After a herding dog has been raised correctly the shepherd can give it many different commands. Some examples would be ‘come-bye’ which makes the dog circle the flock in a clockwise motion as seen in figure 1.2a. ‘Away to me’ makes the dog circle the flock in a counter-clockwise motion figure 1.2b. ‘Hold’ makes sure that the dog does not let the sheep move. On top of these three commands there are many more that the shepherd has at his disposal.

1.2 Modeling shepherding

The last part of the previous section shows how difficult this problem is in the real world. The shepherd has to judge where the sheep are, where they should be, how they should get there, where the dog is, where the dog should be using known commands to indirectly move the sheep in the right direction and where he himself has to go.

1.2.1 How to represent shepherding

Let us start with how we will be representing this problem. The representation has already been mentioned briefly in the previous section where we referred to figure 1.1. First we need the area that we will be working in. This will be a simple 2D representation using x and y coordinates to determine the locations of objects, animals, et cetera. An empty example of this representation can be seen in figure 1.1a.

The sheep will be represented using small arrows placed on their respective x and y locations. On top of the x and y locations each sheep also has a heading which represents which way a sheep is going. This heading is represented using an arrow where the angle relative to the sheep shows the direction. Next is the herding dog. The herding dog will be represented using a dot. x and y coordinates are again important to represent the position of the dog. This time however there is no heading, so the dog can be represented using a simple dot. Last but not least is the goal. In this thesis the goal is represented as a simple circle, which can be seen in figure 1.4.
1.2. MODELING SHEPHERDING

When all sheep have entered this circle the goal has been completed. We have not described how we are going to represent the shepherd in this situation. This is because we will not be modeling it. The herding dog itself will make the required choices.
CHAPTER 1. INTRODUCTION

1.2.2 Modeling sheep

From the previous section we know what the sheep’s intended behavior is: forming a flock, wandering around, grazing and moving away from any threats, for example dogs and the shepherd. We also know that each sheep is represented using coordinates and a heading. On top of that we know that the new location is calculated using the heading of the sheep. What we still lack is a way to update the heading and actually have the sheep perform a certain behavior. This is what we will be looking at here.

One popular method to model flocks, herds and groups of animals are variations of the boids model which was created by Craig Reynolds [6]. This model relies on each animal having specific information like a location and a heading. The model itself consists of three rules which are used to create emergent behavior. This behavior already resembles the flocking of sheep somewhat, which is why it is used here. The initial rules for boids are the following:

**Cohesion**
Moving towards the average location of flockmates (figure 1.3a)
The sheep first checks what sheep are flockmates by checking whether they are within a certain distance. Next it calculates what the average position of all these sheep is and moves towards this point

**Separation**
Moving away from all other flockmates (figure 1.3b)
The sheep again checks what the flockmates are and moves away from them

**Alignment**
Moving towards the general direction of flockmates (figure 1.3c)
Again the sheep checks what the flockmates are and it will move in the same general direction that they are going

At this point the sheep wants to go in three different directions which is not possible. This is why all these new headings are averaged. We then take the old heading of the sheep and calculate the new heading. However at this point we are still missing several things. These three rules depend on flockmates. So if a sheep is alone its heading would not change. On top of that the animals are not yet afraid of any possible threats, which is why this these rules will be extended with two new rules.
1.2. MODELING SHEPHERDING

Randomness
Move in a random direction (figure 1.3d)
Generate a random heading

Avoidance
Move away from threats (figure 1.3c)
Look if there are any threats near and move away from them

We again take all generated headings (figure 1.3f) and use this to generate our final heading for the sheep (figure 1.3g). By adding these new rules to the previous ones Trafankowsky created a behavior for the sheep where they try to form flocks, move around randomly and move away from threats. This is the behavior which he implemented in his simulator.

1.2.3 Modeling the herding dog

As opposed to sheep there is no commonly used way of implementing a herding dog, which is not a bad thing. It allows for many different behaviors. The shepherd is often not implemented, as mentioned in subsection 1.2.1. This changes the original tasks of the herding dogs. Instead of following orders and forcing the sheep to move, the herding dog will have to make its own choices and can not rely on a shepherd. In this section we will look at two current shepherding algorithms.

Miki

The first algorithm has been created by Miki and Nakamura [5]. There are two parts to this algorithm:

1. Move the sheep into one flock (figure 1.4a)
2. Move the sheep towards the goal (figure 1.4b)

In section 1.2.2 we mentioned that sheep react to dogs by moving away from them in the opposite direction. So if a herding dog is supposed to move a sheep into a specific direction then the dog can move behind the sheep so that the sheep feels threatened and moves in the right direction. This way the sheep can be guided into a single flock or towards the goal.

Miki and Nakamura did not provide a specific implementation of gathering the sheep, so the implementation created by Trafankowsky [8] will be used. This is a straightforward implementation that works by finding the closest flock to the dog, finding the closest sheep to this flock, and then moving this sheep to the flock. You just repeat this until you have a single large flock.

Moving flocks and sheep around is what is done in a way that is specific to Miki. When herding either a flock or a sheep, the dog moves directly behind a flock or a sheep to guide it towards its target and an example of this algorithm can be seen in pseudocode in listing 1.1

```
getTheClosestFlockToTheDog
if (allSheepInTheFlock)
    pushFlockToGoalByMovingBehindFlock
else
    findTheClosestSheepToTheFlock
    moveTheFlockToTheSheepByMovingBehindFlock
```

Listing 1.1: Miki algorithm in pseudocode

The choices that the dog has to make in this case relate to judging which sheep form a flock, which sheep should join the flock and where the flock should move.
CHAPTER 1. INTRODUCTION

Lien

The second algorithm has been created by Lien et al. [3]. This algorithm consists of two parts, which are exactly the same as the behaviour by Miki.

1. Move the sheep into one flock (figure 1.4a)

2. Move the sheep towards the goal (figure 1.4b)

Lien also did not provide a specific implementation of gathering the sheep, so the same implemention as for the Miki algorithm will be used. The significant difference between Miki and Lien is the way that they guide sheep around. Whereas for Miki the dog would simply move straight behind a sheep or flock to push it to its target, for Lien the dog circles behind the sheep or the flock to guide it to its target in a fluid motion. An implementation of this algorithm in pseudocode results in a piece of code that resembles the Miki algorithm. This implementation can be seen in listing 1.2.

Listing 1.2: Lien algorithm in pseudocode

```pseudocode
getTheClosestFlockToTheDog
if (allSheepInTheFlock)
    pushFlockToGoalByCirclingBehindFlock
else
    findTheClosestSheepToTheFlock
    moveTheFlockToTheSheepByCirclingBehindFlock
```

1.3 Goal of this thesis

Taking all the previous information together we get the following goal: Create a herding framework which allows one to easily implement new behaviors.

The framework will work as an extension of an existing herding simulator [8] which already contains several herding behaviors, a parameter frame, the entire behavior for the sheep and a visual representation of the sheep and herding dog. The framework will contain several implementations of existing behaviors in the simulator, completely new behaviors, extra information for visual representation and new parameters for framework behaviors.

All shepherding behaviors in the framework will be a simplification of actual shepherding as these behaviors will resolve around the herding dog. The framework will not contain a shepherd so all decisions will be made by the herding dog.

The framework will be tested in two ways. First the performance is tested by comparing framework behaviors with simulator behaviors. In this case performance refers to time instead of whether the framework implementation moves exactly the same as the simulator implementation. The second test is the ability to implement new behaviors.

Chapter 2 describes the simulator made by Trafankowski [8] that we will be using, the changes that have to be made, the first version of the framework in Prolog and how the framework and simulator will communicate. In chapter 3 we will be taking a closer look at the actual framework itself. What are the key components, how does it work and how do current shepherding behaviors look like in the framework? Chapter 4 will contain several experiments: how easy is it to create new behaviors in the framework and is the performance still acceptable? The last chapter, chapter 5, will contain several conclusions. We will look at whether we have reached our goal of creating an easy to use, understandable framework consisting of predicates that can be easily exchanged between users.
1.3. GOAL OF THIS THESIS

(a) Example of cohesion

(b) Example of separation

(c) Example of alignment

(d) Example of random

(e) Example of avoidance

(f) All headings together

(g) The newly generated heading

Figure 1.3: Boids actions
(a) Gathering the sheep with the Miki behavior

(b) Moving the flock towards the goal with the Miki behavior

(c) Gathering the sheep with the Lien behavior

(d) Moving the flock towards the goal with the Lien behavior

Figure 1.4: Miki and Lien behaviors
Chapter 2

Connecting the simulator and the high-level framework

In this chapter we will first take a look at the simulator which we will be using. We will also look at a first version of the framework and how this Prolog framework will be communicating with the simulator.

2.1 Simulator

By now we know that there are several important factors to modeling shepherding:

- Sheep
- Shepherd
- Herding dog
- Parameters

There is one more thing that has not been mentioned before, but which is also of importance. Once you have the necessary models and implementations you would want to use them. For this you would need something to interact with the models and implementations, an interface. Both for setting parameters in the models as well as receiving data back, which can be in the form of a visual representation. The simulator that will be discussed in this section contains all these important elements as can be seen in figure 2.1. We will come back to this figure in section 2.2. The simulator has been created by Trafankowsky [8] using Java, a class-based and object-oriented programming language, allowing one to easily divide all important aspects of shepherding into different classes. Trafankowsky has made this simulator available to me for use in this Thesis and some aspects of it will be briefly explained in this section, but for a complete overview it would be best to look at his paper [8].

2.1.1 Usage

When starting the simulator two windows appear. One representing the actual (ongoing) simulation while the other one represents the parameters. Parameters can be set and changed in real
CHAPTER 2. CONNECTING THE SIMULATOR AND THE HIGH-LEVEL FRAMEWORK

Figure 2.1: All elements of the simulator

Simulator
  Dog
  Sheep
  Parameters
  Representation

time in the parameter window and the simulator can also be started from here. Once the simulator is running the other window will show what is happening. During the simulation parameters can be changed to alter the behavior.

The simulation works on a frame by frame basis. Each frame all information is taken and used to calculate the new positions for the sheep and the dog. During each frame the simulator also checks whether all sheep are within the goal. If they are within the goal either the next goal is chosen or the simulation ends. So basically a frame is a single iteration for the program. During the entire simulation the frame count is continuously updated and the amount of frames at the end of the simulation represents the effectiveness of the behavior, where a lower frame count indicates a better behavior.

2.1.2 Parameters

The first aspect we will be looking at are the parameters. There are sixteen separate parameters for the sheep and each herding behavior also has a few parameters. Information about specific parameters can be found in the paper by Trafankowsky [8]. All parameters with their default values for the sheep are in the following list:

- Sheep are attracted to all sheep in a circular zone of 400 units
- Sheep are attracted to the nearest sheep in a circular zone of 400 units
- Sheep are attracted to the center of neighboring sheep in a circular zone of 400 units
- Sheep are attracted to a random sheep in a circular zone of 400 units
- Max sheep speed is 2 units per frame
- Number of sheep is 10
- Number of nearest sheep is 1
- Number of neighboring sheep is 2
- Minimum flocking separation is 60 units
- Flocking strength for all sheep is 5
- Flocking strength for nearest sheep is 5
2.1. SIMULATOR

- Flocking strength for center of neighboring sheep is 5
- Flocking strength for random sheep is 1
- Minimum distance for sheep to repel sheep is 20 units
- Sheep to sheep repel strength is 65
- Maximum wander level is 1

Strength refers to how far a sprite will move in a certain direction in each frame before the maximum speed limits the speed of the sprite. The wander level is what the random movement for each sheep is based on. All parameters are put together in a single window, allowing a quick overview of all settings and direct access. Figure 2.2 shows what this window looks like. This window also contains the ability to switch between behaviors and other options.

![Figure 2.2: A screenshot of the parameters frame of the simulator while viewing settings for the sheep and for the algorithm by Brandon](image-url)
CHAPTER 2. CONNECTING THE SIMULATOR AND THE HIGH-LEVEL FRAMEWORK

Changing these parameters has an immediate effect on the simulator, because during simulation the values are directly retrieved from the parameter window.

2.1.3 Representation

Second is the representation of the actual simulation. Without a visual representation one would only have the results of the simulation. The visual representation and the model are independent of each other. In this case 1 unit matches 1 pixel, but this can easily be changed. This representation should show several things:

- Sheep
- Herding dog
- Goals
- Obstacles

All these objects are included and shown. Figure 2.3 shows what the representation looks like.

![Figure 2.3: A screenshot of the animator frame while running a simulation](image)

The circle in the upper left corner represents the goal. The area within the blue circle is where all the sheep should be before either the next goal is chosen or the simulation is over. The white figures, which can be seen in three groups, represent the sheep. The black figure which is the most left object on the map, except for the goal, represents the dog.
2.2. WHY A CONNECTION

Herding dog  The herding dog has its own class within the simulator, including all possible behaviors. Each frame all information is given to this class and it uses this information to calculate the new location for the dog. This new location is then used to draw the visual representation of the dog and will also be used in the calculation for the next frame.

Sheep  The sheep work much the same as the herding dog, except that there are multiple sheep. Each frame all information is given to the sheep and they calculate where they should go based on other sheep and the herding dog. This information is then used to draw the representation and used for the next frame.

Other aspects  There is much more to the simulator of course, for example setting goals, creating objects and batch testing. This section only scratched the surface. For a closer look at the simulator and other possible uses one should look at Trafankowsky’s paper [8].

2.2  Why a connection

Unlike using just a simulator, the framework will be its own entity. It will be a separate program. However we will still include the same elements as before:

- Sheep
- Herding dog
- Parameters
- Representation

Instead of all these elements being in a single program, as shown in figure 2.1, they will be divided over two programs. This new division can be seen in figure 2.4.

The framework will handle all calculations relevant to the dog. We know however that to do these calculations the framework will also need information about the sheep and goal. Because the simulator and framework are written in different languages they can not exchange information like two Java classes can. This is why a way to communicate between the framework and simulator has to be added. A visual representation of this can be seen in figure 2.5.

JPL [2] will be used for communication between Java and Prolog. This is a Java to Prolog interface which comes with the SWI-Prolog suite.
2.3 Creating the Prolog code

Creating good and efficient Prolog code was a challenge. Because of this the first part will go over the initial ideas, which were not ideal. In the next part the improvements to the code will be explained. On top of showing what is possible we will also show what is a good way to create a modular and efficient framework. But first a short explanation of what Prolog is.

2.3.1 What is Prolog

As opposed to many other languages that are imperative or functional, Prolog is a logic based programming language. It consists of atoms (for example `y`, `car`, `Test`), numbers (both integers as well as floats), variables (always start with a capital letter) and compound terms (lists, strings or for example `is_male(pete)`). These objects can be used to create rules and facts. These rules and facts always have the following form:

```
Head :- Body.
```

However in the case of a fact the body is substituted with `true`. Take for example the previous example from the compound terms, if this were written as a fact it would look like this:

```
is_male(pete) :- true.
```

Which would be the same as:

```
is_male(pete).
```

Variables can then be used to find the answers to queries. Let's say we want to know who is male. We could ask the following query:

```
is_male(X).
```

Running this query would result in `X` being substituted with `pete`. A possible rule to add to this subject would be:

```
is_human(X) :- is_male(X).
```

The rule above says that a person is human if he is a male. Prolog also has an AND (,) operator and an OR (;) operator. A human can of course also be female, adding this to the rule would result in the following:

```
is_human(X) :- is_male(X) ; is_female(X).
```

In some cases a predicate might require an atom for a variable. Or it could be the opposite. When describing a certain predicate a + is added in front of the variable if it is required and - if it needs an empty variable.
2.3. CREATING THE PROLOG CODE

2.3.2 Creating the first version of the Prolog code

Figure 2.6 contains a general idea of the framework. First we set all necessary information and add it to the knowledge base. Using the information from the knowledge base the framework can then perform calculations. At the end these results of the calculations can be gotten using getters.

From the first section we know that there are several points of information that are crucial when having to calculate the new position of the dog:

- The location of the sheep
- The location of the dog
- The location of the goal
- The current frame

First this information has to be added to the knowledge base in Prolog. The initial idea was writing all the relevant information to a text file and then adding this information to the knowledge base, but this proved to be rather slow. Finding alternatives led to the Prolog predicates `retract/1` and `assertz/1`. These allow one to manipulate the knowledge base by adding and removing information.

The first step was adding the position of the dog. For this you obviously need the position of the dog, but also the current frame. The predicate in listing 2.1 shows how this information is added and removed after five frames. The predicate receives a frame and a position and immediately adds the position at the frame to the knowledge base. It also removes an older position, keeping the knowledge base clean.

```
addPosDog(Frame, Pos) :-
    assertz(posDog(Frame, Pos)),
    OldFrame is Frame - 2,
    retractall(posDog(OldFrame, _)).
```

Adding the goal information to the knowledge base is a bit simpler, because the frame does not matter here. The predicate to add the goal position can be seen in listing 2.2.
CHAPTER 2. CONNECTING THE SIMULATOR AND THE HIGH-LEVEL FRAMEWORK

Listing 2.2: Adding the location of the goal

addGoal(Pos) :-
    retractall(posGoal(_)),
    assertz(posGoal(Pos)).

There are also several other values that can be added that do not depend on a frame number,
much like addGoal, but those all have the same structure.

Putting the information to use

Now that all the relevant information has been added to the knowledge base, it can be put to
use. For example having the following rule in the knowledge base:

posSheep(3,(6,7),5)

Now if you would like to know at what position sheep 5 was in frame 3, you can run the following
query:

posSheep(3,Pos,5)

This will substitute Pos with the actual location of the sheep, which is (6,7) in this case.

Let us start with something simple, for example calculating the average location of a flock.
Two parameters are needed here, firstly the frame and secondly the flock, represented as a list
of integers. The code for getting the average position of a flock can be seen in listing 2.3

Listing 2.3: Getting the average position of a flock

averagePosFlock(Frame, Flock , Result) :-
    maplist(sumCoor(Frame) ,Flock ,SumCoords) ,
    maplist(avgCoor ,SumCoords, Flock ,AvgCoords) ,
    maplist(combinePosAndFlock , Flock ,AvgCoords , Result) .

There are three steps to calculating the average position of a flock. The first step is calculating
the total sum of the positions of the sheep in the flock. This results in the parameter SumCoords,
which consists of a tuple representing the sum of the coordinates. The second step is calculating
the average position, resulting in the parameter AvgCoords consisting of a tuple representing the
average coordinates. Now there is a separate flock and an average position. But due to the way
the framework is made now these variables have to be combined together in a tuple, resulting in
the parameter Result which has the form of (Flock,AvgCoords).

2.3.3 Improving the Prolog code

The initial version was very strict concerning which order queries had to be executed. An
example can be seen in listing 2.4. The variable Flocks would be a list of lists of integers, each
representing the IDs of the sheep in a flock. This would then be used in averagePosFlock.
The resulting variable FlocksAvg would then be a list of tuples containing a list of integers
and the average location. These two would have to be executed before you could even properly
call findClosestFlock. This was a very awkward way of working. Because every little bit of
information had to be added to the tuple. Every time something new came along everything had
to be rewritten to fit the new layout.

Listing 2.4: Getting the closest flock in the first version of the code

generateFlocks(Frame, Flocks) ,
averagePosFlock(Frame, Flocks , FlocksAvg) ,
findClosestFlock (Frame, FlocksAvg , ClosestFlock)
The entire codebase had to be rewritten. The resulting code was very modular. You could call almost anything without having to meet any prerequisites. This can be seen in listing 2.5. This single line is now all that is needed to get the closest flock.

Listing 2.5: Getting the closest flock in the latest version of the code

```prolog
getClosestFlock(Frame, ClosestFlock)
```

The actual code of that predicate can be seen in listing 2.6. What happens is that first all flocks are requested and then they are compared to each other in calculateClosestFlockToDog.

Listing 2.6: The code to calculate the closest dog to a flock

```prolog
getClosestFlockToDog(Frame, ClosestFlock) :-
getFlocks(Frame, Flocks),
calculateClosestFlockToDog(Frame, Flocks, ClosestFlock),
).
```

In listing 2.7 the closest dog is calculated. This is done by requesting the distance from the flock to the dog and looking which one is smallest.

Listing 2.7: Calculating the closest flock to the dog

```prolog
calculateClosestFlockToDog(_Frame,[ ClosestFlock ] , ClosestFlock) :- !.
calculateClosestFlockToDog(Frame ,[ Flock1 , Flock2 | Tail ] , ClosestFlock) :-
getFlockDistanceToDog(Frame, Flock1 , Dis1),
getFlockDistanceToDog(Frame, Flock2 , Dis2),
( Dis1 > Dis2 -->
calculateClosestFlockToDog(Frame ,[ Flock2 | Tail ] , ClosestFlock)
; calculateClosestFlockToDog(Frame ,[ Flock1 | Tail ] , ClosestFlock)
).
```

Every piece of information can be gotten using get predicates, which we will refer to as getters. These are all modular and work so that you do not have to worry about any prerequisites. However this has one obvious disadvantage. Each time you use a getter everything has to be calculated. This would be less efficient than the first method for which everything had to be calculated just once. The solution to this was tabling.

Tabling

Tabling allows the framework to store the information that is created by a getter, so that if it needs it again it can retrieve the information straight from the knowledge base. The predicate getClosestFlockToDog with tabling can be seen in listing 2.8

Listing 2.8: Getting the closest flock to the dog with tabling

```prolog
getClosestFlockToDog(Frame, ClosestFlock) :-
( closestFlockToDog(Frame, ClosestFlock),! )
; ( getFlocks(Frame, Flocks),
calculateClosestFlockToDog(Frame, Flocks, ClosestFlock),
assertz(closestFlockToDog(Frame, ClosestFlock)),
OldFrame is Frame - 2,
retractall(closestFlockToDog(OldFrame,_,_)),!)
```
The first step is checking whether the predicate `closestFlockToDog` already exists for a certain frame. If it does the variable `ClosestFlock` will be substituted with the flock that is closest to the dog. If it does not exist, the closest flock is calculated like in listing 2.6. This is then added to the knowledge base using `assertz` and all old instances are removed to keep the knowledge base from cluttering.

Modularity and tabling together allows the code to be much more user friendly than the first version while probably not being much less efficient.

### 2.4 Changing the simulator

The initial simulator, while having a vast array of options, does not completely suit our needs. Several changes had to be made to create the connection to Prolog and to add several features.

#### 2.4.1 Establishing a connection to Prolog

The connection between Java and Prolog is established using JPL [2], an interface that ships with the SWIPL distribution of Prolog. This allows the user to directly call Prolog predicates from Java. A specific class was created for this task. This separated it from the rest of the program and made it easier to maintain. The class that handles the communication is called `PrologCommunication`. This class is initialized exactly once at the start of the program, making sure that the Prolog code is also loaded exactly once.

#### 2.4.2 Sending information to Prolog

The first step to interacting with Prolog is done by loading the data. Opening a Prolog file is done by using the `consult/1` predicate from Prolog, as can be seen in listing 2.9. The next step is creating a string. In this case that will be `consult(flockBehaviours.pl)`. You then create a query from this string for which you can then check whether it has a solution. If it has, the file has been loaded.

**Listing 2.9: Consulting a file using JPL**

```java
private void openPrologFile(String name){
    String t1 = "consult ( " + name + " )";
    Query q1 = new Query(t1);
    System.out.println ( t1 + "/uni2423" + (q1.hasSolution() ? "succeeded" : "failed") );
}
```

The next step is giving all relevant information to the Prolog knowledge base. The predicates that are used for adding the information to the knowledge base can be seen in section 2.3.

When adding the dog the first step is to create a string that represents a valid Prolog command, for example `addPosDog(1,(5,4))`, which can then be converted into a query and run through JPL as can be seen in 2.10. Checking whether the query has a solution results in it actually being executed.

**Listing 2.10: Adding the location of the dog through JPL**

```java
public void addDog(int frame, float x, float y){
    String q1s = "addPosDog(" + frame + ",(" + x + "," + y + ")\n);";
    Query q1 = new Query(q1s);
```
2.4. CHANGING THE SIMULATOR

Adding the sheep takes a bit more work. Because there can be any number of sheep we get a list of sheep. We then have to create a valid Prolog command which also contains a list with elements that represent the actual sheep. This is done using a string builder and iterating through the list. The resulting String is then run through JPL as can be seen in listing 2.11

Listing 2.11: Adding the location of all sheep through JPL

```java
public void addSheep(ArrayList<SheepSprite> sheepList, int frame){
    StringBuilder posList = new StringBuilder("addPosSheep(*);
    posList.append(frame);
    posList.append(" ,[");
    for(int i = 0 ; i < sheepList.size() ; i++){
        posList.append("((");
        posList.append(sheepList.get(i).getXcoord());
        posList.append(" ,") ;
        posList.append(sheepList.get(i).getYcoord());
        posList.append(" ) ,") ;
        posList.append(" ,") ;
    }
    posList.deleteCharAt(posList.length()-1);
    posList.append("])");
    Query sheepq = new Query(posList.toString());
    sheepq.hasSolution();
}
```

The positions for the dog and the sheep have to be added every frame. Which is different for objects like a goal. A goal only has to be added if the location has changed. This is also to reduce the amount of queries that have to be sent through JPL and increase the speed of the program.

The first step when updating the goal is to check whether the location has changed. If it has not changed, we do nothing. If it has, we create a valid query and run it through JPL to add the position to the knowledge base. Lastly we change the previous x and y coordinates, to reflect the change in the knowledge base. The function that has been described can be seen in listing 2.12

Listing 2.12: Adding the location of the goal through JPL

```java
public void updateGoal(float x, float y){
    if (x != prevGoalX || y != prevGoalY){
        Query sgoal = new Query("addGoal(" + x + "," + y + ")") ;
        sgoal.hasSolution();
        Query sgoaltest = new Query("posGoal((X,Y))") ;
        if (sgoaltest.hasSolution()){ 
            prevGoalX = x ;
            prevGoalY = y ;
        }
    }
}
```

Because all these functions add to the behavior of the dog, all these functions are called in SheepDogSprite.java. As mentioned before this class simulates the dog and contains all relevant information which are needed for these functions.
2.4.3 Receiving information from Prolog

Receiving information from Prolog is just as important as sending information to Prolog. At the moment there are two types of information that can be received from Prolog. The first, and most important of the two, is the new location of the dog. For every frame in the simulator the information of the current state of the world is sent to Prolog and a response is asked. In this case what the new position of the dog should be. Retrieving this new information is done with a single predicate in Prolog, thus it is also short in Java. The code for retrieving the new position using the Miki algorithm can be seen in listing 2.13.

Listing 2.13: Retrieving the new dog location using Miki

```java
public void runMiki(int frame) {
    int newFrame = frame + 1;
    String q2s = "miki("+newFrame+",(X,Y))";
    Query q2 = new Query(q2s);
    if (q2.hasSolution()) {
        newX = new Float(q2.oneSolution().get("X").toString());
        newY = new Float(q2.oneSolution().get("Y").toString());
    } else {
        System.out.println("No solution found in frame "+ frame);
    }
}
```

The new locations are set in two local variables, `newX` and `newY` which can then be gotten using getters. The second kind of information that can be requested from Prolog is information that can be used to generate the (live) representation of the information. An example of retrieving the information for the Miki algorithm can be seen in listing 2.14. If this were for the Lien algorithm we would also be requesting the circle radius of the dog.

Listing 2.14: Retrieving information for the live representation of the Miki algorithm

```java
public void drawMiki(int frame) {
    ArrayList infoList = new ArrayList();
    infoList.add(getTargetLocation(frame));
    infoList.addAll(getFlocks(frame));
    this.infoArray = infoList;
}
```

The information that has been requested is parsed and put in `infoArray` which can then be gotten from the Prolog communication class, but this will be described in more detail in section 2.4.5.

2.4.4 Adding extra options to the parameters

We know by know that from the previous chapters that parameters are important for these simulations. The framework itself also has multiple parameters that can be set. For example minimum distance from the flock, speed of the dog, et cetera. On top of that there are multiple algorithms that can be used. Being able to change the parameters and the choice of algorithm on the fly is a great addition, because the changes would be immediately seen. To make this possible a new tab has been added to the parameters window that has been shown before in figure 2.2. This new tab has several options as can be seen in figure 2.7. These options are briefly described below:

Algorithm

A dropdown menu which selects the algorithm that should be run.
2.4. CHANGING THE SIMULATOR

Active
Whether the framework is active or that one of the other algorithms in the simulator should be used.

Live info
If this is selected the live information for the current selected algorithm will be shown while running the simulator.

Follow sheep
Used for one of the simpler algorithms which is simply following a sheep. Using this slider you can select which one should be followed.

Max distance moved
The maximum distance that the dog can move during a single frame.

Cluster distance
Distance that is used when generating the flocks in Prolog and deciding which sheep belongs to a flock and which does not.

Minimum distance between dog and sheep
Used to tell the framework how far the dog should stay away from the sheep. If this is set to 0 the dog could end up on the same location as the sheep.

Circle limit
Used for algorithms where the dog circles behind the sheep, this tells the framework how many degrees maximum it can move.

Using these options you can directly influence the behavior of the dog and test different settings.

![Figure 2.7: A screenshot of new options that have been added to the parameters](image)

2.4.5 Live representation of the data

The live representation of the framework specific information is a significant addition to the default representation of the simulator. Receiving the information has been briefly described in section 2.4.3. The data that can shown is the data from the knowledge base of the framework. This allows the user to better understand what is happening. Several things can be shown:

- Flocks
- Average positions of flocks
• Radius of flocks
• Target location of the dog
• The angle that the dog will be circling behind a flock

After being parsed the resulting information is sent to the Animator where the representations are drawn on the frame. One thing to note here is that the information for the live representation is combined into large predicates in Prolog, instead of calling each get predicate from Java. This allows one to use less calls to Prolog to represent the same data, improving performance of the program when showing the live representation. An example of the live representation can be seen in figure 2.8 and a description of relevant objects and colors is explained below:

**Red dot (beneath the dog)**
- The position the dog is moving towards

**Blue circle (inner circle around the sheep)**
- The radius of the flock

**Red circle (outer circle around the sheep)**
- The radius of the flock plus the minimum distance

**Yellow line (near the red dot, over the red line)**
- The angle which the dog will circle behind a flock

![Figure 2.8: A screenshot of the live representation of information](image-url)
Chapter 3

SheepLog

In this chapter several things will be explained, starting with a closer look at the predicates describing different predicate levels and types. Relevant predicates will also be described. In the last part examples of the behaviors will be given and how they were implemented using SheepLog.

3.1 What exactly is SheepLog

By now we have explained what shepherding itself is and what possible algorithms are in chapter 1 and what the simulator looks like and how the simulator will be communicating with SheepLog in chapter 2. SheepLog will consist of dozens of rules, or predicates, that can be combined to create new behaviors. SheepLog will act much as a policy. It will be given the relevant data from the simulator and it will then try to find the action that best matches this data. Almost every change in data will result in a new optimal action.

As a small reminder SheepLog is the right side of the image in figure 2.5 and will take care of the herding dog.

3.2 A closer look at predicates

There are two important facets regarding the predicates, levels and types. In the first section predicate levels will be explained. What they are and how you can find out what level a certain predicate has. In the second section the different types of predicates will be explained: why there are different types, what the advantages are and what they are used for. Each type will also be briefly described.

3.2.1 Predicate levels

You can divide the get-predicates in SheepLog into different levels. This is based on the level of abstraction. Starting at the lowest level, level 0, all up to level $n$. A predicate is put in a certain level based on what other get-predicates it uses and what level they are. This means that there is no maximum level for predicates. What information is needed for a predicate does not influence its level. For example a predicate that returns the location of the first member of a flock would be of a lower level than the predicate that is required to create a flock.
Level 0

It starts at level 0, the lowest level. Predicates in this level return raw data. Some examples are:

- Position of the sheep
- Position of the dog
- Several variables set by the simulator like the minimum distance between the sheep and the dog

Level 1

The predicates in this level use the raw data from level 0 predicates. Now you can perform calculations on this information or aggregate the data. Some examples are:

- Getting the average position of a flock (Only first level because a group of sheep is given, this predicate does not have to generate flocks)
- Getting distances between objects, for example sheep and the herding dog
- Getting angles between objects, for example sheep and the herding dog

Level 2

This adds another layer of abstraction. By now we have access to locations, distances, angles, averages and all sheep from the previous level, which can be used for the following example predicates:

- Generating all flocks
- Calculating angles between flocks and other flocks, sheep or the shepherd
- Calculating the distances between flocks and other flocks, sheep or the shepherd

Level 3

Predicates in this level obviously use angles and distances so for example finding the furthest flock. Figure 3.2 shows a graph for the Miki behavior which connects all predicates needed to generate the behavior.

Visual example

In this section a short visual example of what is needed to generate flocks is given. We start on the lowest level, level 0. For flocks we need two things, the position of the sheep and the maximum distance between sheep before they can be considered a flock. Figure 3.1a shows this information for six sheep. The location is shown in coordinates and the maximum distance is shown by the circle around each sheep. The positions of the sheep can then be used to calculate the distances between sheep as seen in figure 3.1b. Now that we have all distances for the sheep we can compare them to the maximum flock distance that was represented in level 0 as a circle. When sheep fall within this distance they form a flock. For this example four flocks are generated as can be seen in figure 3.1c.
3.2. A CLOSER LOOK AT PREDICATES

3.2.2 Predicate types

In SheepLog there are different types of predicates that have different functionalities. In the following two sections it will be briefly explained what types of predicates there are and why these exist.

Brief description of the types

The following list contains different types of predicates including a short explanations for each of them:

Setters
Set predicates are mostly used for communication from Java to Prolog, for example to add the positions of the sheep and the dog to the knowledge base in Prolog. It is also used for the variables that can be changed in the simulator. If one of these values is changed a set predicate is called and the new value is added to the knowledge base in Prolog and the old values are removed.

Getters
Getters are predicates that retrieve information. These can be used to get information
about the dog, sheep, goal, but also to get information about the variables that can be set in the simulator. Getters use tabling where necessary.

**Checkers**
Checkers can be compared to boolean functions. These predicates only receive information, they do not fill any parameter. They only evaluate to true or false.

**Helpers**
Helpers are predicates that calculate information. Using helpers prevents you from having to write the same code multiple times and allows you to move the calculations out of the get predicates.

**Behaviors**
Behaviors are the predicates that correspond to a specific behavior. For example Lien or Miki. These always have the same layout. When calling one of these predicates you have two parameters. One contains the current frame and the other will be filled with the new coordinates for the dog.

**Java Info**
These predicates are mostly used to get information from Prolog to Java. For example for the live preview. The default predicates are not used because you would add about 4 extra calls to Prolog. Using specific predicates allows you to combine all this information and turn it into a single call, speeding up the program.

**Targets**
These calculate target locations, basically where the dog should be given certain information.

**Movers**
These predicates take a frame, a target location and return the new location for the dog. Based on what you would say is realistic movement for a dog.

**Why use different types**
By now it is hopefully clear what these types do and you might have also figured out why they exist. There are several reasons why there are different types of predicates, Some of these will be briefly described below:

**Reusability**
Using different types of predicates and dividing the code allows you to reuse certain parts. For example almost all get predicates have their own calculate predicate under the helper predicates. The helper predicates also contain several predicates that do some general calculations. For example calculating the angle between two positions or reducing an angle so that it is between 0 and \( 2\pi \).

**Readability**
Using these different types also increases readability. Like mentioned in the previous description, almost each get predicate has its own calculate predicate. This reduces most get predicates to a few simple steps.

1. Check if the information exists
2. If it does not: calculate the information and store it
This way you do not get these huge predicates and you can get a quick overview of what exactly is happening.

3.2.3 Other properties

Another small but important aspect of SheepLog is the naming convention. By looking at a name one should immediately understand what it does. Sometimes this can result in short names:

`getFlocks(+Frame, -Flocks)`

But sometimes it can also result in longer names:

`getFurthestSheepFromFlockInFlock(+Frame, +Flock, -Furthest)`

It would be possible to use abbreviations, but when predicates get more complex having a collection of abbreviations might not be the best solution, so names will be written in full and explain the purpose of the predicate.

3.3 SheepLog predicates

The framework already consists of dozens upon dozens of predicates. Too many to explain here one by one. All predicates, including descriptions, can be found in the appendix.

3.4 Herding behaviors in SheepLog

In this section some of the sheep herding behaviors will be shown in SheepLog. Many predicates that are used were initially specifically made for these behaviors. But now that they have been implemented they can also be used for other behaviors.

3.4.1 Miki

First is the algorithm by Miki which has been previously explained in chapter 1.2.3. In short it has the following two steps:

- Gather all the sheep
- Move the sheep to the goal

As mentioned before, gathering the sheep is done by pushing the closest flock to the closest sheep and all pushing by the dog is done by moving directly behind the sheep and moving towards the goal. Exploring these options more gives the following steps:

- Find the closest flock
- Are all sheep in the flock?
  
  *All sheep are in the flock:*
  
  - Move behind the flock, aimed at the goal
  
  *Not all sheep are in the flock:*

CHAPTER 3. SHEEPLOG

- Find the closest sheep
- Move behind the flock, aimed at the sheep

This seems to be the most compact way to describe this algorithm, so translating this to SheepLog is the next step. Some actions, for example finding the closest flock, were already implemented, moving behind a sheep facing a certain target was not. The following predicates are relevant to this behavior:

- getClosestFlockToDog(+Frame, -ClosestFlock)
- checkAnySheepNotInFlock(+Frame, +Flock)
- getClosestSheepToFlockNotInFlock(+Frame, +Flock, -Closest)
- moveBehindFlockFacingTarget(+Frame, +Flock, +ID, -TargetLocation)
- moveBehindFlockFacingGoal(+Frame, +Flock, -TargetLocation)
- moveToLocation(+Frame, +TargetLocation, -NewPosition)

The first five of these predicates represent the first five lines from the description, and the predicates names and requirements should specify exactly what they do and what they need. One small change has been made, instead of immediately moving behind the flock, a target location is set for the dog, which he then moves to using a separate algorithm. Putting this separate allows the movement behavior to be easily changed, for example moving faster the further away the dog is. Putting all these building blocks together results in the predicate shown in listing 3.1 which describes the behavior given by Miki.

Listing 3.1: Predicate describing Miki behavior

miki(FrameNew,Pos) :-
    getClosestFlockToDog(Frame, ClosestFlock),
    ( checkAnySheepNotInFlock(Frame, ClosestFlock) ->
        getClosestSheepToFlockNotInFlock(Frame, ClosestFlock, Sheep),
        moveBehindFlockFacingTarget(Frame, ClosestFlock, Sheep, Target) ;
        moveBehindFlockFacingGoal(Frame, ClosestFlock, Target)
    ),
    moveToLocation(Frame, Target, Pos).

The predicate shown above is almost a direct match to the description and reading the predicates should give everyone a good idea of what the behavior does and how it works.

The Miki predicate itself is a level 7 predicate, meaning there are six layers between the final predicate and the raw data. A graphical representation of all get and check predicates and how they are related can be seen in figure 3.2.
Figure 3.2: A graph representing the connections between predicates for the Miki behavior.
3.4.2 Lien

As described in the earlier section 1.2.3, the behavior designed by Lien is much like the behavior designed by Miki. It has the two steps consisting of gathering the sheep and moving the sheep towards the goal. The main difference is how the sheep are guided. Instead of moving directly behind the flock and pushing them, the herding dog circles behind the flock, from left to right and vice versa to guide them towards the goal. Combining all these elements gives the following description:

- Find the closest flock
- Are all sheep in the flock?
  
  All sheep are in the flock:
  
  - Circle behind the flock, aimed at the goal
  
  Not all sheep are in the flock:
  
  - Find the closest sheep
  - Circle behind the flock, aimed at the sheep

And as one can see, the only difference between this algorithm and the algorithm by Miki is that the herding dog circles behind the flock. The next step was converting this to SheepLog. Due to the similarity between the two behaviors and the fact that several predicates have been added, we can reuse previous predicates. This only leaves two new predicates:

- `circleBehindFlockFacingGoal(+Frame, +Flock, -TargetLocation)`
- `circleBehindFlockFacingSheep(+Frame, +Flock, +ID, -TargetLocation)`

Putting the old and new predicates together gives the predicate that can be seen in listing 3.2 which describes the behavior by Lien.

Listing 3.2: Predicate describing Lien behavior

```prolog
lien(_FRAMENew, Pos) :-
  getClosestFlockToDog(_Frame, ClosestFlock),
  ( checkAnySheepNotInFlock(_Frame, ClosestFlock)
    ->
      getClosestSheepToFlockNotInFlock(_Frame, ClosestFlock, Sheep),
      circleBehindFlockFacingSheep(_Frame, ClosestFlock, Sheep, Target)
    ;
      circleBehindFlockFacingGoal(_Frame, ClosestFlock, Target)
  ),
  moveToLocation(_Frame, Target, Pos).
```

3.4.3 Miki and Lien combined

By implementing the behaviors from Miki en Lien there are now multiple useful predicates which can form behaviors. So far only two have been described, but with the predicates that the framework now contains more can be built.

One possible downside of the algorithm by Miki is that the dog always moves directly behind a flock. By standing in the same relative spot and moving towards the flock the sheep on the
3.4. HERDING BEHAVIORS IN SHEEPLOG

edges might be pushed to either the left or right, splitting from the group. This results in having to gather those sheep up again. In the case of the algorithm by Lien one downside could be that the dog always circles behind the sheep. In the case of an entire flock this is useful because it helps preventing sheep from splitting off. In the case of a single sheep however it might be inefficient. So how about combining these two behaviors. It appears it might be inefficient to circle behind a single sheep or pushing into a flock. So discard these parts and take the best of both. The combination of Miki and Lien can be seen in listing 3.3

Listing 3.3: Combination of Miki and Lien behaviors

\[
\text{mien}(\text{FrameNew}, \text{Pos}) : - \\
\text{getClosestFlockToDog}(\text{Frame}, \text{ClosestFlock}) , \text{getClosestSheepToFlockNotInFlock}(\text{Frame}, \text{ClosestFlock}, \text{Sheep}) , \text{moveBehindFlockFacingTarget}(\text{Frame}, \text{ClosestFlock}, \text{Sheep}, \text{Target}) ; \text{moveBehindFlockFacingGoal}(\text{Frame}, \text{ClosestFlock}, \text{Target}) , \text{moveToLocation}(\text{Frame}, \text{Target}, \text{Pos}).
\]

This new behavior, called ‘mien’, shows how the behaviors from Miki and Lien can be easily combined to solve possible disadvantages of the original behaviors. In this case using the straight push from Miki (figure 1.4a) and the circling push from Lien (figure 1.4d). Of course this is not the only way these can be combined. It could also be the other way around and removing the possible advantages, by circling behind single sheep and pushing into flocks.

Other behaviors

Adding new predicates to the framework is still a possibility. One of the other examples is modifying the behavior by Lien by making the circle angle of the herding dog scale by the amount of sheep that are in the closest flock. This allows the dog to directly push a single sheep and start circling as soon as a flock forms to keep it together. This behavior can be seen in listing 3.4.

Listing 3.4: Lien algorithm with scaling circling

\[
\text{lienScale}(\text{FrameNew}, \text{Pos}) : - \\
\text{getClosestFlockToDog}(\text{Frame}, \text{ClosestFlock}) , \text{getClosestSheepToFlockNotInFlock}(\text{Frame}, \text{ClosestFlock}, \text{Sheep}) , \text{circleBehindFlockFacingSheepScale}(\text{Frame}, \text{ClosestFlock}, \text{Sheep}, \text{Target}) ; \text{circleBehindFlockFacingGoalScale}(\text{Frame}, \text{ClosestFlock}, \text{Target}) , \text{moveToLocation}(\text{Frame}, \text{Target}, \text{Pos}).
\]

Figure 3.3a shows how much the dog would circle behind a single sheep while pushing it towards the flock and figure 3.3b shows how the angle has increased by the time the dog is pushing the entire flock towards the goal.
CHAPTER 3. SHEEPLOG

Figure 3.3: Representation of a scaling Lien behavior

(a) Small angle while herding a single sheep  
(b) Larger angle while herding an entire flock
Chapter 4

Experiments

Two types of experiments, or tests, will be done with SheepLog. The first test is measuring performance of the algorithms of Miki and Lien in SheepLog versus these algorithms in Java which come with the simulator. The second type of experiment will test how easy it is to implement new ideas.

4.1 Comparing performance

In this section the performance of the Java and SheepLog versions of the Miki and Lien algorithms will be compared. Performance is always an important part when running simulations. Preferably it should be as fast as possible so that many simulations can be run. However in this case there is a trade-off to make. The trade-off for the framework is between speed and ease of creating new behaviors. The goal for this framework was to make it easy to create new herding behaviors, but it should still be usable.

When comparing the performance of the Miki and Lien algorithms in SheepLog with those that are in Java and come with the simulator, there are multiple types of performance that can be measured. In this case we will only look at the following two:

- Performance relative to the world in the simulator
- Performance relative to real world time

In this case the performance relative to the simulator is measured in frames. In previous chapters it was explained that new locations are calculated for each frame so this will be used as measurement. The final value will be saved when the sheep have reached the last goal. More information on this is in the next section. The performance is measurement is not whether the framework implementation lets the dog move the same as the Java implementation. Considering it would be hard to get a quantitative measurement for this with all the randomness. However visual observation has shown that in essence the behaviors are the same. Fortunately the Miki and Lien behaviors have always achieved their goals during testing.

4.1.1 Simulator settings

For all tests the exact same settings for the simulator were used, to ensure that any difference in performance of the algorithms is due to the algorithms themselves. Explanations of each setting can be found in the original paper of the simulator [8]. Though there still remain some random
factors in the simulator, which is why each test consists of 1000 samples. The random factors are the following:

- Initial location of the sheep
- A random direction variable inherent to the sheep behavior
- Slightly moving towards a random sheep in the flock behavior

For these tests we used the default settings except for attraction to the nearest sheep and a random sheep which have both been changed to 90, because otherwise sheep would flock together no matter where on the field they are. During the tests three goals would appear, always in the same order and always in the same location. First in the upper left corner, followed by the upper right corner and ending in the lower left part of the frame. An overview of these goals can be seen in figure 4.1.

![Figure 4.1: The three goals for the tests](image)

### 4.1.2 Miki algorithm

In this section the performance difference between the Miki algorithm implementations will be discussed based on a test containing 100 trials.
4.1. COMPARING PERFORMANCE

Settings

For the Miki algorithm in Java the following settings were used:

- Max dog speed is 7
- Dog repelling distance is 50
- Dog repelling strength is 20
- Distance behind sheep is 60
- Distance of flocked sheep is 150

For the Miki algorithm in SheepLog the following settings were used:

- Max distance moved is 10
- Cluster distance is 75
- Minimum distance between dog and sheep is 5

Results

These were the results obtained after each algorithm had been run a 1000 times:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Java Mean (Std. Dev)</th>
<th>SheepLog Mean (Std. Dev)</th>
</tr>
</thead>
<tbody>
<tr>
<td># frames</td>
<td>7520.93 (1064.65)</td>
<td>11047.26 (3957.55)</td>
</tr>
</tbody>
</table>

The results here show that there is quite a performance difference between implementations. The SheepLog implementation is worse in terms of time taken when looking at the mean. While the best case scenario does not differ that much between implementations, the worst case scenario does, being more than five times as bad. The standard deviation also shows that the results for the SheepLog implementation vary more wildly than the Java implementation, being almost four times as big.

Keep in mind that these results were gotten with the default parameters for both implementations. It might be possible to change the parameters so that both implementations improve. However there is a significant amount combinations possible and these will not all be covered.

The results for the amount of actual time passed can be seen in the table below. Figure 4.2 shows the scatter plots for frames versus seconds for both implementations. This test contains 100 trials.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Java Mean (Std. Dev)</th>
<th>SheepLog Mean (Std. Dev)</th>
</tr>
</thead>
<tbody>
<tr>
<td># seconds</td>
<td>0.1362 (0.0510)</td>
<td>4.7835 (1.4683)</td>
</tr>
</tbody>
</table>
At first glance it is immediately obvious that there is a significant difference, even when keeping the differences in terms of frames in mind. On average the difference is a factor of 35. This sounds like a lot, but when the starting point is 0.14 seconds the end result is not that bad. It is still usable in many cases.

4.1.3 Lien algorithm

In this section the performance difference between the Lien algorithm implementations will be discussed based on a test containing 100 trials.

Settings

For the Lien algorithm in Java the following settings were used:

- Max dog speed is 4
- Dog repelling distance is 50
- Dog repelling strength is 20
- Distance behind sheep is 60
- Distance of flocked sheep is 150
- Distance from point behind sheep is 10
- Distance to move to side is 45
- Degree move to side is 60

For the SheepLog implementation the following settings were used:

- Max distance moved is 10
4.1. COMPARING PERFORMANCE

- Cluster distance is 75
- Minimum distance between dog and sheep is 5
- Circle limit is 78

Results

These were the results obtained after each implementation had run 1000 times with regard to the amount of frames needed to finish the simulation:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td># frames in Java</td>
<td>8278.12</td>
<td>2208.73</td>
<td>4358</td>
<td>23715</td>
</tr>
<tr>
<td># frames in SheepLog</td>
<td>8836.37</td>
<td>1405.98</td>
<td>5531</td>
<td>22442</td>
</tr>
</tbody>
</table>

Solely looking at the mean would suggest that the SheepLog implementation functions worse than the Java implementation, however there are some important differences. The standard deviation for the SheepLog implementation is lower than the Java implementation, meaning that performance is more consistent than the Java implementation. While in the best case scenario the SheepLog implementation performs worse than the Java implementation, in the worst case scenario it functions better.

The results for the amount of actual time passed can be seen in the table below. Figure 4.3 shows the scatter plots for frames versus seconds for both implementations:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td># seconds in Java</td>
<td>0.1567</td>
<td>0.0809</td>
<td>0.0851</td>
<td>0.8345</td>
</tr>
<tr>
<td># seconds in SheepLog</td>
<td>4.45223</td>
<td>0.9958</td>
<td>3.1630</td>
<td>11.9553</td>
</tr>
</tbody>
</table>

Just like with the Miki algorithm there is a big difference between the Java and SheepLog implementation. For the Lien algorithm the difference is a factor of 28. The starting point is again low, meaning that the SheepLog implementation will still be usable. However the SheepLog implementation does again show that the amount of real world time linearly increases together with the amount of frames needed. This means that even when possible new behaviors in SheepLog become very extensive and require more frames the time does not suddenly exponentially increase.

4.1.4 Comparing the SheepLog implementations

Judging by the results, the bad performance of the Miki implementation is not due to SheepLog itself. Most likely due to some wrong parameters. The Lien algorithm shows that in terms of frames SheepLog can compete with the Java implementations. At the moment you also have less control over specific settings of the implementations in the framework, but this can be resolved in later versions.
4.2 Implementing new behaviors

This section will review how implementing new behaviors works. Whether it is easy to implement ideas, what predicates can be reused and what has to be added.

4.2.1 Pushing sheep one by one

The first new behavior approaches the herding in a different way. Instead of looking at flocks it looks at single sheep. The idea is that the dog moves the furthest sheep from the goal into the goal, then moving on to the sheep that is then furthest away from the goal and moving that one to the goal. So step by step it would look like this:

- Are all sheep in the goal?
  - Not all sheep are in the goal:
    - Find the furthest sheep from the goal
    - Push this sheep to the goal

The first problem that we encounter here has to do with the predicate \texttt{getFurthestSheepFromGoal}. You get a new sheep as soon as the current sheep that is being guide to the goal is closer to the goal than another sheep. A way is needed to focus on a certain sheep, drop this focus and find a new sheep. For this the predicate \texttt{getFocusSheepFurthest} was created.

The predicate \texttt{getFocusSheepFurthest(+Frame,-ID)} does multiple things. First it checks if a focused sheep already exists for the frame (tabling). If it does not exist the predicate checks whether there was a focused sheep in the previous frame. If there was not one the furthest sheep from the goal is found and returned. If there was a previous sheep then the predicate checks whether this sheep is already within the bounds of the goal. If the sheep is within the goal a new furthest sheep is found and returned. If it was not within the bounds of the goal the previous focused sheep is returned.
4.2. IMPLEMENTING NEW BEHAVIORS

Now that the above predicate has been added you will always have access to the ID of the furthest sheep, even when it might be closer than some other sheep. The next step is getting a target location, in this case circleBehindFlockFacingGoal will be used. The ID that is returned by getFocusSheepFurthest is put in a list and given to the function so the target location is returned. Last not but least the dog has to move towards the location, using moveToLocation.

Putting all these predicates together results in the behavior that can be seen in listing 4.1.

Listing 4.1: Implementation of the behavior that pushes sheep one by one

```
pushFurthestSheep(NewFrame, Pos) :-
   Frame is NewFrame - 2,
   getFocusSheepFurthest(Frame, ID),
   circleBehindFlockFacingGoal(Frame, [ID], Target),
   moveToLocation(Frame, Target, Pos).
```

Figures 4.4a and 4.4b show a visual representation of this behavior. First the herding dog guides one sheep to the goal, followed by the next.

![Visual representation of the behavior that pushes sheep one by one](image)

(a) Pushing the first sheep towards the goal  
(b) Pushing the second sheep towards the goal

Figure 4.4: Representation of the new behavior that pushes the furthest sheep towards the goal

4.2.2 Forming a flock by circling around all sheep

The second idea was changing how the flock is brought together. Instead of the dog pushing the closest flock to another closest flock, it circles around all sheep, slowly pushing them together. Taking the previous into consideration it meant that there had to be a differentiation between when the dog should be circling the sheep and when it should be pushing them. In this case the difference is when there is either one flock or multiple flocks. Step by step this would look like this:

- **Are all sheep in a single flock?**
  
  *All sheep are in a single flock:*
  
  - Circle behind the sheep facing the goal

- **Not all sheep are in a single flock:**
  
  - Circle around all sheep
CHAPTER 4. EXPERIMENTS

For checking whether there is only a single flock a new predicate is created. Initially a predicate like checkSingleFlock might seem like a good idea, but this does not help with reusability. A better idea is checkAmountOfFlocks(+Frame,+Amount). This predicate will work for all amounts of flocks.

After checking what should be done there are two options. First is circling around all sheep. The framework already contains a predicate called circleFlock(+Frame,+Flock,+Direction,-Target). In this case the dog should circle around all sheep, so circleFlock is combined with getAllSheep. This results in the wanted behavior. The second option is used when there is a single flock and this is pushing the flock to the goal, which can be done using either moveBehindFlockFacingTarget or circleBehindFlockFacingTarget. Last but not least is moving towards the target location that is returned by circleFlock or circleBehindFlockFacingTarget using moveToLocation. Piecing all this together results in the behavior that can be seen in listing 4.2.

Listing 4.2: Implementation of the behavior that circles around sheep and then pushes them to the goal

```
pushToGoalWithCircleGathering(FrameNew,Pos) :-
    Frame is FrameNew - 1,
    checkAmountOfFlocks(Frame,1)
    ->
    getClosestFlockToDog(Frame,Flock),
    circleBehindFlockFacingGoal(Frame,Flock,Target)
    ;
    getAllSheep(Frame,AllSheep),
    circleFlock(Frame,AllSheep,clockwise,Target)
    ),
    moveToLocation(Frame,Target,Pos).
```

Figures 4.5a and 4.5b show a visual representation of this behavior. First the herding dog guides one sheep to the goal, followed by the next.

![Figure 4.5a](image1)
![Figure 4.5b](image2)

Figure 4.5: Representation of the new behavior that circles around all sheep to gather them and then pushes them to the goal

4.2.3 Gathering sheep by picking the furthest sheep

The initial predicates had a big advantage, which was that they always looked for the closest flock and sheep. Due to this flocks were automatically formed and kept together. This third new
behavior will be different. Instead of picking the closest sheep to the flock the suggestion was to pick the furthest sheep from the flock. The issue that became apparent was that a new structure was needed, one that had not yet been implemented in the framework. A small and different example can already be seen in the first new behavior of this section, where the dog focuses on a single sheep, ignoring others. The first step consisted of creating a general idea what is needed and what has to be added. The new behavior was put into pseudocode to facilitate this process. This behavior can be seen below:

- **Are all sheep in a single flock?**
  
  *All sheep are in a single flock:*
  
  - Circle behind the sheep facing the goal
  
  *There are multiple flocks:*
  
  - **How many primary flocks are there**
    
    *There is one primary flock:*
    
    - Find the sheep that is furthest away from the primary flock and add a primary predicate
    - Move behind the primary flock facing the target
    
    *There are no primary flocks:*
    
    - Find the closest sheep and add a primary predicate
    - Find the furthest sheep and add a primary predicate
    - Move behind the closest sheep facing the furthest
    
    *There are multiple primary flocks:*
    
    - Find the closest primary flock
    - Find the closest primary sheep to the flock
    - Move behind the flock facing the target

One issue was how the knowledge base should know whether a sheep belongs to the primary set. There would be multiple solutions, creating a list with all relevant IDs and creating a new predicate which contains all information for each sheep for example. In this case a new predicate was chosen which contains only the ID of the sheep, `primSheep(+ID)`. This extra predicate was easy to integrate with other predicates like `generateFlocks` by adding the new predicate to each `posSheep` check. Many other predicates were needed for this issue. The new `get`, `set` and `check` predicates can be seen in the list below, new `help` predicates have been omitted:

- `getClosestSheepToDog(+Frame, -ID)`
- `getSheepDistanceToDog(+Frame, +Flock, -Dis)`
- `checkAmountOfPrimaryFlocks(+Frame, +Amount)`
- `getClosestPrimarySheepToFlockNotInFlock(+Frame, +Flock, -Closest)`
• `getAllPrimarySheep(+Frame, +AllSheep)`
• `getFurthestSheepFromFlockNotInFlock(+Frame, +Flock, -Furthest)`
• `getClosestPrimaryFlockToDog(+Frame, -ClosestFlock)`
• `getPrimaryFlocks(+Frame, -Flocks)`
• `setPrimaryFlockSheep(+ID)`

Many predicates from the list above were already in the framework one way or another. They just had to be modified slightly to accommodate the new primary predicate. Putting all these new predicates together creates the new behavior. Due to the many checks that are required it is a bit longer than previous behaviors. The new behavior can be seen in listing 4.3 and figures 4.6a, 4.6b and 4.6c show how the flock is created by choosing the furthest sheep.

Listing 4.3: Implementation of the behavior that picks the furthest sheep from the flock

```prolog
furthestSheepFromFlock(FrameNew, Pos) :-
    Frame is FrameNew - 1,
    (checkAmountOfFlocks(Frame, 1) ->
        getClosestFlockToDog(Frame, Flock),
        circleBehindFlockFacingGoal(Frame, Flock, Target)
    ;
    (checkAmountOfPrimaryFlocks(Frame, 1) ->
        getPrimaryFlocks(Frame, PrimaryFlocks),
        first(PrimaryFlocks, FirstFlock),
        getFurthestSheepFromFlockNotInFlock(Frame, FirstFlock, Furthest),
        setPrimaryFlockSheep(Furthest),
        moveBehindFlockFacingTarget(Frame, FirstFlock, Furthest, Target)
    ;
    (checkAmountOfPrimaryFlocks(Frame, 0) ->
        getClosestSheepToDog(Frame, ClosestSheep),
        setPrimaryFlockSheep(ClosestSheep),
        getFurthestSheepFromFlockNotInFlock(Frame, [ClosestSheep], FurthestSheep),
        setPrimaryFlockSheep(FurthestSheep),
        moveBehindFlockFacingTarget(Frame, [ClosestSheep], FurthestSheep, Target)
    ;
        getClosestPrimaryFlockToDog(Frame, ClosestPrimaryFlock),
        getClosestPrimarySheepToFlockNotInFlock(Frame, ClosestPrimaryFlock, Closest),
        moveBehindFlockFacingTarget(Frame, ClosestPrimaryFlock, Closest, Target)
    )
    ),
    moveToLocation(Frame, Target, Pos).
```

For this behavior a new system was implemented, a ‘primary flock’. This flock has priority when herding. For the behavior the furthest sheep was added to the flock but this can be replaced
with alternative predicates. For example the first sheep that is clockwise relative to the flock and dog.

(a) Moving the first two sheep together
(b) Add a third sheep to the flock group
(c) Add a fourth sheep to the flock group

Figure 4.6: Representation of the new behavior that uses a primary flock
Chapter 5

Conclusion

The goal of this thesis was to create a herding framework which allows one to easily implement new behaviors. Over the past four chapters we have shown multiple things, starting with how shepherding with a single dog can be modeled and which current algorithms exist for shepherding in chapter one. In chapter two we looked at what a possible simulator for shepherding looks like, and most importantly, how a framework for shepherding has been created.

In chapter three we have shown in detail how the framework works and how it can be used to implement existing behaviors. In the fourth chapter we took an ever closer look and tested whether new behaviors can be easily implemented and whether the framework itself is actually usable in terms of performance.

These tests have shown that it is relatively easy to implement new behaviors due to the modularity of the framework which allows predicates to be reused. Whether a behavior was easy to implement depends on what predicates are available. If many predicates already exist creating a new behavior is easier compared to when many predicates have to be made from scratch. Due to the modularity these new predicates can be distributed to other people so that they can be reused, improving the framework for everyone and giving everyone more options. So while at first implementing new behaviors can be a bit of a chore, this would become increasingly easier as more predicates are made for the framework.

The tests have also shown that there is one significant disadvantage. Performance is significantly worse compared to native code contained within the simulator. Although it is worse, it is still usable, though this will depend on what it will be used for. This framework will not be suited for applications that are time sensitive. However it will be ideal for situations where time is not much of an issue or where one is interested in prototyping different behaviors. The framework allows you to quickly try out different ideas that might pop up in your head, run a few tests and see if it does what you want. Once it is clear that it works you could then create the same behavior in the native code of the simulator.

Now that the goal for this thesis has been met there are several aspects that can be extended and improved in the future. Of course one could create a new simulator, add an extra dimension to the framework so that it works in a three dimensional environment, add new behaviors for the dog or add support for multiple dogs. The framework is aimed at a very specific task with specific inputs and outputs and changing this task is interesting. The current foundation that has been lain can be applied to other problems. A possible problem is simulating animals in an environment, using the actual senses as inputs and possible actions as outputs. This can also include interacting with the environment, something which at the moment is not being done. Another interesting subject to look at are humans. Humans are much more complicated than
animals and this can be represented in the framework. Instead of only having a direction and speed one could add a mental state. Actions and reactions can then be influenced based on whether someone is happy, tired, scared or something else.
Bibliography


Appendices
Appendix A

Overview

Setters

- setPrimaryFlockSheep(+ID)
- setDirection(+Direction)
- setTargetLocation(+Frame,+Target)
- setCircleLimit(+CircleLimit)
- addGoal(+Position)
- setClusterDistance(+ClusterDistance)
- setMaxDist(+MaxDist)
- setMinDist(+MinDist)
- addPosDog(+Frame, +Position)
- addPosSheep(+Frame, +ListOfSheep)
- addPosSheep(+Frame, +Sheep)

Getters

- getPrimaryFlocks(+Frame, -Flocks)
- getClosestPrimaryFlockToDog(+Frame, -ClosestFlock)
- getFurthestSheepFromFlockNotInFlock(+Frame, +Flock, -Furthest)
- getAllPrimarySheep(+Frame, -AllSheep)
- getClosestPrimarySheepToFlockNotInFlock(+Frame, +Flock, -Closest)
- getSheepDistanceToDog(+Frame, +Flock, -Distance)
- getClosestSheepToDog(+Frame, -ClosestSheep)
- getFocusSheepFurthest(+Frame, -ID)
• getFurthestSheepFromGoal(+Frame, -ID)
• getSheepToGoalDistance(+Frame, +ID, -Distance)
• getAllFlocksAndGoalRadius(+Frame, -Radius)
• getFurthestFlockFromAllFlocksAndGoal(+Frame, -Furthest)
• getGoalToAllFlocksAndGoalDistance(+Frame, -Distance)
• getAllFlocksAndGoalDistance(+Frame, +Flock, -Distance)
• getAllFlocksAndGoalAveragePosition(+Frame, +Flocks, -Pos)
• getAllSheep(+Frame, -AllSheep)
• getTargetLocation(+Frame, -Pos)
• getClosestSheepToFlockNotInFlock(+Frame, +Flock, -Closest)
• getFlockToGoalAngle(+Frame, +Flock, -Angle)
• getFlockToSheepAngle(+Frame, +Flock, +ID, -Angle)
• getFlockToDogAngle(+Frame, +Flock, -Angle)
• getFlockRadiusWithMinDistance(+Frame, +Flock, -Radius)
• getFlockRadius(+Frame, +Flock, -Radius)
• getFurthestSheepFromFlockInFlock(+Frame, +Flock, -Furthest)
• getSheepToFlockDistance(+Frame, +Flock, +ID, -Distance)
• getAllSheepNotInFlock(+Frame, +Flock, -SheepNotInFlock)
• getClosestFlockToDog(+Frame, -ClosestFlock)
• getFlockDistanceToDog(+Frame, +Flock, -Dis)
• getFlockAveragePosition(+Frame,+Flock,-AvgPos)
• getFlocks(+Frame, -Flocks)
• getCircleLimit(-CircleLimit)
• getClusterDistance(-ClusterDistance)
• getMinDist(-MinDist)
Checkers

- checkAmountOfPrimaryFlocks(+Frame, +Amount)
- checkSheepInGoal(+Frame, +ID)
- checkAmountOfFlocks(+Frame, +Amount)
- closerToLeftGoal(+Frame, +Flock)
- closerToLeftSheep(+Frame, +Flock, +ID)
- lessFlockMembersThan(+Flock, +Amount)
- checkDogWithinGoalLimitScale(+Frame, +Flock)
- checkDogWithinGoalLimit(+Frame, +Flock)
- checkDogWithinSheepLimitScale(+Frame, +Flock, +ID)
- checkDogWithinSheepLimit(+Frame, +Flock, +ID)
- checkAnySheepNotInFlock(+Frame, +Flock)
- checkDistance(+Frame, +ID, +ID, +Distance)
- largerThanAbs(+XValue, +YValue)

Helpers

- first(+List, -FirstElement)
- calculateClosestSheepToDog(+Frame, +Sheep, -Closest)
- generatePrimaryFlocks(+Frame, -Flocks)
- calculateDistance(+Pos, +Pos, -Distance)
- calculateFurthestSheepFromGoal(+Frame, +IDList, -Furthest)
- calculateFurthestFlockFromAllFlocksAndGoal(+Frame, +List, -Furthest)
- calculateClosestSheepToFlockBasedOnList(+Frame, +Flock, +SheepList, -ClosestSheep)
- calculateFurthestSheepFromFlockInFlock(+Frame, +Flock, +Flock, -Furthest)
- closestGroup(+Frame, +FlockList, -ClosestFlock)
- averagePositions(+PosList, -PosAvg)
- sumPositions(+PosList, -PosSum)
- closerToLeft(+Angle1, +Angle2)
- checkWithinLimit(+LeftLimit, +RightLimit, +Angle)
- reducedAngle(+Angle, -NewAngle)
- angle(+Pos1, +Pos2, -Angle)
APPENDIX A. OVERVIEW

- calculateFlockRadius(+Frame, +Flock, -Radius)
- calculateSheepToFlockDistance(+Frame, +Flock, +ID, -Distance)
- getAllSheepNIF(+Frame, +Cluster, +ID, -IDs)
- avgPos(+Cluster, +Pos, -AvgPos)
- sumPosSheep(+Frame, +Cluster, -SumPos)
- generateFlocks(+Frame, -Flocks)
- fixedValue(+Value, -FixedValue, +MaxValue)
- numberOne(+Value, -ValueOne)

Behaviors

- furthestSheepFromFlock(+Frame, -Pos)
- pushFurthestSheep(+Frame,-Pos)
- pushToGoalWithCircleGathering(+Frame, -Pos)
- lienScale(+Frame, -Position)
- lienv2(+Frame, -Position)
- lien(+Frame, -Position)
- miki(+Frame, -Position)
- circleAllSheep(+Frame, -Pos)

Java info

- getAllAngleInfoScale(+Frame, -Info)
- getAllAngleInfo(+Frame, -Info)
- mergeAllAngleInfo(+FlockInfo, +DogAngle, +TargetAngle, +CircleLimit, -Info)
- getAllSheepFlockInfo(+Frame,- FlockInfo)
- getAllFlockInfo(+Frame, -FlockInfo)
- mergeAllFlockInfo(+Flock, +Avg, +Radius, -Info)
Flock targets

- `circleBehindFlockFacingGoalScale(+Frame, +Flock, -Target)`
- `circleBehindFlockFacingGoal(+Frame, +Flock, -Target)`
- `circleBehindFlockFacingSheepScale(+Frame, +Flock, +Sheep, -Target)`
- `circleBehindFlockFacingSheep(+Frame, +Flock, +Sheep, -Target)`
- `circleBehindFlock(+Frame, +Flock, +Limit, +Left, -Target)`
- `circleFlock(+Frame, +Flock, -Pos)`
- `getDogTarget(+FlockPosition, +GoalPosition, +Angle, +Radius, -TargetPosition)`
- `moveBehindFlockFacingTarget(+Frame, +Flock, +ID, -Target)`
- `moveBehindFlockFacingGoal(+Frame, +Flock, -Target)`

Movers

- `moveToLocation(+Frame, +Target, -NewPosition)`
Appendix B

Setters

:- dynamic primSheep/1.
% Adds a sheep to the primary flock
% setPrimaryFlockSheep(+ID)

setPrimaryFlockSheep(ID) :-
  assertz(primSheep(ID)).

:- dynamic direction/1.
% Sets the direction that the dog will circle
% setDirection(+Direction)

setDirection(Direction) :-
  retractall(direction(_)),
  assertz(direction(Direction)).

:- dynamic targetLocation/2.
% Sets the target location
% setTargetLocation(+Frame,+Target)

setTargetLocation(Frame,Pos) :-
  assertz(targetLocation(Frame,Pos)),
  OldFrame is Frame - 2,
  retractall(targetLocation(OldFrame,_)).

:- dynamic circleLimit/1.
APPENDIX B. SETTERS

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sets the circle limit and removes all old values
% setCircleLimit(+CircleLimit)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
setCircleLimit(CircleLimit) :-
    retractall(circleLimit(_)),
    assertz(circleLimit(CircleLimit)).

:- dynamic posGoal/1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Adds the position of a goal and removes all other goals
% addGoal(+Position)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
addGoal(Pos) :-
    retractall(posGoal(_)),
    assertz(posGoal(Pos)).

:- dynamic clusterDistance/1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sets the distance used to generate clusters. Each time it
% is changed all previous versions are removed
% setClusterDistance(+ClusterDistance)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
setClusterDistance(ClusterDistance) :-
    retractall(clusterDistance(_)),
    assertz(clusterDistance(ClusterDistance)).

:- dynamic maxDist/1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sets the maximum distance the sheep can move. Each time it
% is changed all previous versions are removed
% setMaxDist(+MaxDist)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
setMaxDist(MaxDist) :-
    retractall(maxDist(_)),
    assertz(maxDist(MaxDist)).

:- dynamic minDist/1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sets the minimum distance between the dog and a flock.
% This value is used if the dog is following a single sheep
and is added to the radius of a flock

setMinDist(+MinDist)

setMinDist(MinDist) :-
  retractall(minDist(_)),
  assertz(minDist(MinDist)).

:- dynamic posDog/2.

addPosDog(+Frame, +Position)

addPosDog(Frame,Pos) :-
  assertz(posDog(Frame,Pos)),
  OldFrame is Frame - 5,
  retractall(posDog(OldFrame,_)).

addPosSheep(+Frame, +ListofSheep)

addPosSheep(Frame,PosSheepList) :-
  maplist(addPosSheep(Frame),PosSheepList),!.

:- dynamic posSheep/3.

addPosSheep(+Frame, +Sheep)

addPosSheep(Frame,(Pos,ID)) :-
  OldFrame is Frame - 10,
  (  
posSheep(OldFrame,Pos,ID)
->
  retract(posSheep(OldFrame,_Pos,ID)),
  assertz(posSheep(Frame,Pos,ID))
  ;
  assertz(posSheep(Frame,Pos,ID))
)
Appendix C

Getters

:- dynamic primaryFlocks/2.
% Returns the flocks created from sheep that have a primSheep predicate
% getPrimaryFlocks(+Frame, -Flocks)
%
getPrimaryFlocks(Frame,Flocks) :-
(   primaryFlocks(Frame,Flocks),!
);
(   generatePrimaryFlocks(Frame,Flocks),
    assertz(primaryFlocks(Frame,Flocks)),
    OldFrame is Frame - 2,
    retractall(primaryFlocks(OldFrame,_,)),!
).

:- dynamic closestPrimaryFlockToDog/2.
% Gets the closest primary flock to the dog
% getClosestPrimaryFlockToDog(+Frame, -ClosestFlock)
%
getClosestPrimaryFlockToDog(Frame,ClosestFlock) :-
(   closestPrimaryFlockToDog(Frame,ClosestFlock),!
);
(   getPrimaryFlocks(Frame,Flocks),

65
calculateClosestFlockToDog(Frame,Flocks,ClosestFlock),
assertz(closestPrimaryFlockToDog(Frame,ClosestFlock)),
OldFrame is Frame - 2,
retractall(closestPrimaryFlockToDog(OldFrame,_,_)),!)
).

:- dynamic furthestSheepFromFlockNotInFlock/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gets the furthest sheep from the flock that is not in the
% flock
%
% getFurthestSheepFromFlockNotInFlock(+Frame, +Flock,
%     -Furthest)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFurthestSheepFromFlockNotInFlock(Frame,Flock,Furthest) :-
    (furthestSheepFromFlockNotInFlock(Frame,Flock,Furthest),!)
    ;
    (getAllSheep(Frame,AllSheep),
     subtract(AllSheep,Flock,NewFlock),
     calculateFurthestSheepFromFlockInFlock(Frame,Flock,NewFlock,Furthest),
     assertz(furthestSheepFromFlockNotInFlock(Frame,Flock,Furthest)),
     OldFrame is Frame - 2,
     retractall(furthestSheepFromFlockNotInFlock(OldFrame,_,_)),!)
).

:- dynamic allPrimarySheep/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns all sheep that have a primsheep predicate
%
% getAllPrimarySheep(+Frame, -AllSheep)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllPrimarySheep(Frame,AllSheep) :-
    (allPrimarySheep(Frame,AllSheep))
    ;
    (getPrimaryFlocks(Frame,Flocks),
     flatten(Flocks,AllSheep),
     OldFrame is Frame - 1,
     retractall(allPrimarySheep(OldFrame,_)),
     assertz(allPrimarySheep(Frame,AllSheep))
).
:- dynamic closestPrimarySheepToFlockNotInFlock/3.
% Returns the closest sheep to a certain flock that are not in that flock and stores this as a predicate
%
% getClosestPrimarySheepToFlockNotInFlock(+Frame, +Flock, -Closest)
%
getClosestPrimarySheepToFlockNotInFlock(Frame,Flock,Closest) :-
  (
    closestPrimarySheepToFlockNotInFlock(Frame,Flock,Closest),!
  )
  ;
  (getAllPrimarySheep(Frame,AllSheep),
   subtract(AllSheep,Flock,SheepNotInFlock),
   calculateClosestSheepToFlockBasedOnList(Frame,Flock,SheepNotInFlock,Closest),
   assertz(closestPrimarySheepToFlockNotInFlock(Frame,Flock,Closest)),!
  ).

:- dynamic sheepDistanceToDog/3.
% Returns the distance between a sheep and the dog
%
% getSheepDistanceToDog(+Frame, +Flock, -Distance)
%
getSheepDistanceToDog(Frame,ID,Dis) :-
  (sheepDistanceToDog(Frame,ID,Dis),!)
  ;
  (posSheep(Frame,PosSheep,ID),
   posDog(Frame,PosDog),
   calculateDistance(PosDog,PosSheep,Dis),
   assertz(sheepDistanceToDog(Frame,ID,Dis)),
   OldFrame is Frame - 2,
   retractall(sheepDistanceToDog(OldFrame,_,_)),!
  ).

:- dynamic closestSheepToDog/2.
% Gets the closest sheep to the dog
%
% getClosestSheepToDog(+Frame, -ClosestSheep)
%
getClosestSheepToDog(Frame,ID) :-
APPENDIX C. GETTERS

(  
closestSheepToDog(Frame,ID),!
)
;
(
getAllSheep(Frame,Sheep),
calculateClosestSheepToDog(Frame,Sheep,ID),
assertz(closestSheepToDog(Frame,ID)),
OldFrame is Frame - 1,
retractall(closestSheepToDog(OldFrame,_)),!
).

:- dynamic focusSheep/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns either the furthest sheep or the sheep that is the
% current focus
%
% getFocusSheepFurthest(+Frame, -ID)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFocusSheepFurthest(Frame,ID) :-
(  
focusSheep(Frame,ID),!
)
;
(
OldFrame is Frame - 1,
focusSheep(OldFrame,IDFocus)
->
  (  
    retract(focusSheep(OldFrame,IDFocus)),
    checkSheepInGoal(Frame,IDFocus)
    ->
      getFurthestSheepFromGoal(Frame,ID),
      assertz(focusSheep(Frame,ID)),!
    ;
    ID is IDFocus,
    assertz(focusSheep(Frame,ID)),!
  );
  getFurthestSheepFromGoal(Frame,ID),
  assertz(focusSheep(Frame,ID)),!
).

:- dynamic furthestSheepFromGoal/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates the furthest sheep from the goal
%
% getFurthestSheepFromGoal(+Frame, -ID)
%
getFurthestSheepFromGoal(Frame, ID) :-
    (furthestSheepFromGoal(Frame, ID), !)
    ;
    (getAllSheep(Frame, AllSheep),
calculateFurthestSheepFromGoal(Frame, AllSheep, ID),
assertz(furthestSheepFromGoal(Frame, ID)),
OldFrame is Frame - 1,
retractall(furthestSheepFromGoal(OldFrame, _)), !).

:- dynamic sheepToGoalDistance/3.

getSheepToGoalDistance(Frame, ID, Distance) :-
    (sheepToGoalDistance(Frame, ID, Distance), !)
    ;
    (posSheep(Frame, PosSheep, ID),
posGoal(PosGoal),
calculateDistance(PosSheep, PosGoal, Distance),
assertz(sheepToGoalDistance(Frame, ID, Distance)),
OldFrame is Frame - 2,
retractall(sheepToGoalDistance(OldFrame, ID, _))
), !.

:- dynamic allFlocksAndGoalRadius/2.

getAllFlocksAndGoalRadius(Frame, Radius) :-
    (allFlocksAndGoalRadius(Frame, Radius)
    ;

:- dynamic sheepToGoalDistance/3.

getSheepToGoalDistance(Frame, ID, Distance) :-
    (sheepToGoalDistance(Frame, ID, Distance), !)
    ;
    (posSheep(Frame, PosSheep, ID),
posGoal(PosGoal),
calculateDistance(PosSheep, PosGoal, Distance),
assertz(sheepToGoalDistance(Frame, ID, Distance)),
OldFrame is Frame - 2,
retractall(sheepToGoalDistance(OldFrame, ID, _))
), !.

:- dynamic allFlocksAndGoalRadius/2.

getAllFlocksAndGoalRadius(Frame, Radius) :-
    (allFlocksAndGoalRadius(Frame, Radius)
    ;

APPENDIX C. GETTERS

{
    getFurthestFlockFromAllFlocksAndGoal(Frame,Furthest),
    getFlockToAllFlocksAndGoalDistance(Frame,Furthest,FlockDistance),
    getGoalToAllFlocksAndGoalDistance(Frame,GoalDistance),
    (FlockDistance > GoalDistance
        -> Radius is FlockDistance
        ; Radius is GoalDistance)
}.

:- dynamic furthestFlockFromAllFlocksAndGoal/2.

% Returns which flock is the furthest from the average when
% looking at all flocks and the goal
%
% getFurthestFlockFromAllFlocksAndGoal(+Frame, -Furthest)
%
%getFurthestFlockFromAllFlocksAndGoal(Frame,Furthest) :-
%    (furthestFlockFromAllFlocksAndGoal(Frame,Furthest),!)
%; (getFlocks(Frame,Flocks),
%     calculateFurthestFlockFromAllFlocksAndGoal(Frame,Flocks,Furthest),
%     assertz(furthestFlockFromAllFlocksAndGoal(Frame,Furthest)),
%     OldFrame is Frame - 2,
%     retractall(furthestFlockFromAllFlocksAndGoal(OldFrame,_)),!)
).

:- dynamic goalToAllFlocksAndGoalDistance/2.

% Returns what the distance is between the goal and the
% average of all flocks and the goal
%
% getGoalToAllFlocksAndGoalDistance(+Frame, -Distance)
%
%getGoalToAllFlocksAndGoalDistance(Frame,Distance) :-
%    (goalToAllFlocksAndGoalDistance(Frame,Distance),!)
%; (posGoal(PosGoal),
getAllFlocksAndGoalAveragePosition(Frame, PosAvg),
calculateDistance(PosGoal, PosAvg, Distance),
% Distance is sqrt((XAvg - XGoal)^2 + (YAvg - YGoal)^2),
assertz(goalToAllFlocksAndGoalDistance(Frame, Distance)),
OldFrame is Frame - 2,
retractall(goalToAllFlocksAndGoalDistance(OldFrame, _)), !
).

:- dynamic flockToAllFlocksAndGoalDistance/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the distance between a flock and the average
% position of all flocks and the goal
% %
% getFlockToAllFlocksAndGoalDistance(+Frame, +Flock, %
% -Distance)
% %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFlockToAllFlocksAndGoalDistance(Frame, Flock, Distance) :-
(  flockToAllFlocksAndGoalDistance(Frame, Flock, Distance), !
) ;
(  getAllFlocksAndGoalAveragePosition(Frame, PosAvg),
      flockToAllFlocksAndGoalDistance(Frame, Flock, Distance),
      calculateDistance(PosAvg, PosFlock, Distance),
      % Distance is sqrt((XAvg - XFlock)^2 + (YAvg - YFlock)^2),
      assertz(flockToAllFlocksAndGoalDistance(Frame, Flock, Distance)),
      OldFrame is Frame - 2,
      retractall(flockToAllFlocksAndGoalDistance(OldFrame, _, _)), !
  )
).

:- dynamic allFlocksAndGoalAveragePosition/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates the average position between all flocks and the
% goal. Does not weigh the flocks
% %
% getAllFlocksAndGoalAveragePosition(+Frame, +Flocks, -Pos)
% %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllFlocksAndGoalAveragePosition(Frame, Pos) :-
(  allFlocksAndGoalAveragePosition(Frame, Pos), !
) ;
(  getFlocks(Frame, Flocks),
    maplist(getFlockAveragePosition(Frame), Flocks, FlockPos),
    ...
posGoal(GoalPos),
append(FlockPos,[GoalPos],PosList),
averagePositions(PosList,Pos).

:- dynamic allSheep/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generates a list of all sheep and stores this in a
% predicate
% % getAllSheep(+Frame, -AllSheep)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllSheep(Frame,AllSheep) :-
  (  
    allSheep(Frame,AllSheep),!
  ) ;
  (  
    getAllSheepNotInFlock(Frame,[],AllSheep),
    assertz(allSheep(Frame,AllSheep)),
    OldFrame is Frame - 2,
    retractall(allSheep(OldFrame,_,)),!
  ).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the target location
% % % getTargetLocation(+Frame,-Pos)
% % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getTargetLocation(Frame,Pos) :-
  targetLocation(Frame,Pos).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the closest sheep to a certain flock that are not
% in that flock and stores this as a predicate
% % % getClosestSheepToFlockNotInFlock(+Frame, +Flock, -Closest)
% % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getClosestSheepToFlockNotInFlock(Frame,Flock,Closest) :-
  (  
    closestSheepToFlockNotInFlock(Frame,Flock,Closest),!
  ) ;
  (  
    getAllSheepNotInFlock(Frame,Flock,SheepNotInFlock),
    ...
calculateClosestSheepToFlockBasedOnList(Frame,Flock,SheepNotInFlock,Closest),
assertz(closestSheepToFlockNotInFlock(Frame,Flock,Closest)).

:- dynamic flockToGoalAngle/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the angle between the flock and the goal in a certain frame and stores this value in a predicate
% getFlockToGoalAngle(+Frame, +Flock, -Angle)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFlockToGoalAngle(Frame,Flock,Angle) :-
(  flockToGoalAngle(Frame,Flock,Angle),! )
; (  getFlockAveragePosition(Frame,Flock,FlockPos),
    posGoal(PosGoal),
    angle(FlockPos,PosGoal,Angle),
    assertz(flockToGoalAngle(Frame,Flock,Angle)),
    OldFrame is Frame - 2,
    retractall(flockToGoalAngle(OldFrame,_,_),_),! ).

:- dynamic flockToSheepAngle/4.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the angle between the flock and the sheep in a certain frame and stores this value in a predicate
% getFlockToSheepAngle(+Frame, +Flock, +ID, -Angle)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFlockToSheepAngle(Frame,Flock,ID,Angle) :-
(  flockToSheepAngle(Frame,Flock,ID,Angle),! )
; (  getFlockAveragePosition(Frame,Flock,FlockPos),
    posSheep(Frame,SheepPos,ID),
    angle(FlockPos,SheepPos,Angle),
    assertz(flockToSheepAngle(Frame,Flock,ID,Angle)),
    OldFrame is Frame - 2,
    retractall(flockToSheepAngle(OldFrame,_,_,_),_),! ).

:- dynamic flockToDogAngle/3.
APPENDIX C. GETTERS

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the angle between the flock and the dog and stores
% this value in a predicate
%
% getFlockToDogAngle(+Frame, +Flock, -Angle)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFlockToDogAngle(Frame,Flock,Angle) :-
  (
    flockToDogAngle(Frame,Flock,Angle),!
  );
  (
    flockAveragePosition(Frame,Flock,FlockPos),
    posDog(Frame,DogPos),
    angle(FlockPos,DogPos,Angle),
    assertz(flockToDogAngle(Frame,Flock,Angle)),
    OldFrame is Frame - 2,
    retractall(flockToDogAngle(OldFrame,_,_)),!
  ).

:- dynamic flockRadiusWithMinDistance/3.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the radius of a flock together with the minimum
% distance and stores this value in a predicate
%
% getFlockRadiusWithMinDistance(+Frame, +Flock, -Radius)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getFlockRadiusWithMinDistance(Frame,Flock,RadiusMinDist) :-
  (
    flockRadiusWithMinDistance(Frame,Flock,RadiusMinDist),!
  );
  (
    flockRadius(Frame,Flock,Radius),
    minDist(MinDist),
    RadiusMinDist is Radius + MinDist,
    assertz(flockRadiusWithMinDistance(Frame,Flock,RadiusMinDist)),
    OldFrame is Frame - 2,
    retractall(flockRadiusWithMinDistance(OldFrame,_,_)),!
  ).

:- dynamic flockRadius/3.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the radius of a certain flock in a certain frame
% and stores this value in a predicate
%
% getFlockRadius(+Frame, +Flock, -Radius)
% getFlockRadius(Frame,Flock,Radius) :-
  (  flockRadius(Frame,Flock,Radius),!
  ) ;
  (  calculateFlockRadius(Frame,Flock,Radius),
    assertz(flockRadius(Frame,Flock,Radius)),
    OldFrame is Frame - 2,
    retractall(flockRadius(OldFrame,_,_)),!
  ).

:- dynamic furthestSheepFromFlockInFlock/3.
% Returns the furthest sheep, of the flock, away from the
% average position of the flock
% getFurthestSheepFromFlockInFlock(+Frame, +Flock, -Furthest)
% getFurthestSheepFromFlockInFlock(Frame,Flock,Furthest) :-
  (  furthestSheepFromFlockInFlock(Frame,Flock,Furthest),!
  ) ;
  (  calculateFurthestSheepFromFlockInFlock(Frame,Flock,Flock,Furthest),
    assertz(furthestSheepFromFlockInFlock(Frame,Flock,Furthest)),
    OldFrame is Frame - 2,
    retractall(furthestSheepFromFlockInFlock(OldFrame,_,_)),!
  ).

:- dynamic sheepToFlockDistance/4.
% Returns the distance between a certain sheep and flock in
% a certain frame and stores this data as a predicate
% getSheepToFlockDistance(+Frame, +Flock, +ID, -Distance)
% getSheepToFlockDistance(Frame,Flock,ID,Distance) :-
  (  sheepToFlockDistance(Frame,Flock,ID,Distance),!
  ) ;
  (  calculateSheepToFlockDistance(Frame,Flock,ID,Distance),
assertz(sheepToFlockDistance(Frame,Flock,ID,Distance)),
OldFrame is Frame - 2,
retractall(sheepToFlockDistance(OldFrame,_,_,_)),!.
).

:- dynamic allSheepNotInFlock/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns all sheep that are not in a certain flock in a %
% certain frame and is stored as a predicate %
% getAllSheepNotInFlock(+Frame, +Flock, -SheepNotInFlock)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllSheepNotInFlock(Frame,Flock,SheepNotInFlock) :-
  ( allSheepNotInFlock(Frame,Flock,SheepNotInFlock),!  
  );
  ( once(posSheep(Frame,Pos,ID)),
    getAllSheepNIF(Frame,Flock,ID,SheepNotInFlock),
    assertz(allSheepNotInFlock(Frame,Flock,SheepNotInFlock)),
    OldFrame is Frame - 2,
    retractall(allSheepNotInFlock(OldFrame,_,_)),!).

:- dynamic closestFlockToDog/2.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the flock that is closest to the dog and saves it %
% in case this predicate is called again %
% getClosestFlockToDog(+Frame, -ClosestFlock) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getClosestFlockToDog(Frame,ClosestFlock) :-
  ( closestFlockToDog(Frame,ClosestFlock),!
  );
  ( getFlocks(Frame,Flocks),
    calculateClosestFlockToDog(Frame,Flocks,ClosestFlock),
    assertz(closestFlockToDog(Frame,ClosestFlock)),
    OldFrame is Frame - 2,
    retractall(closestFlockToDog(OldFrame,_,_)),!).

:- dynamic flockDistanceToDog/3.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gets the distance between a certain flock, in a certain frame, and the dog. If returns the value of the predicate and if it does not exist yet it is calculated and stored

% getsFlockDistanceToDog(+Frame, +Flock, -Dis)
%
getFlockDistanceToDog(Frame, Flock, Dis) :-
  (flockDistanceToDog(Frame, Flock, Dis), !)
; (getFlockAveragePosition(Frame, Flock, AvgPos),
  posDog(Frame, PosDog),
  calculateDistance(PosDog, AvgPos, Dis),
  assertz(flockDistanceToDog(Frame, Flock, Dis)),
  OldFrame is Frame - 2,
  retractall(flockDistanceToDog(OldFrame,_,_)), !).

:- dynamic flockAveragePosition/3.
%
% Returns the average position of a certain flock in a certain frame. If it has not been calculated before it calculates it and stores it in a predicate
%
% getFlockAveragePosition(+Frame,+Flock,-AvgPos)
%
getFlockAveragePosition(Frame, Flock, AvgPos) :-
  (flockAveragePosition(Frame, Flock, AvgPos), !)
; (sumPosSheep(Frame, Flock, SumPos),
  avgPos(Flock, SumPos, AvgPos),
  assertz(flockAveragePosition(Frame, Flock, AvgPos)),
  OldFrame is Frame - 2,
  retractall(flockAveragePosition(OldFrame,_,_)), !).

:- dynamic flocks/2.
%
% Returns all flocks based on a certain frame. If the flocks have been calculated before and stored in a predicate it simply returns the value of this predicate. Else it calculates all flocks and stores it in a predicate

APPENDIX C. GETTERS

%%% getFlocks(+Frame, -Flocks)
%%%
getFlocks(Frame, Flocks) :-
    (    flocks(Frame, Flocks),!
    );
    (    generateFlocks(Frame, Flocks),
        assertz(flocks(Frame, Flocks)),
        OldFrame is Frame - 2,
        retractall(flocks(OldFrame,_)),!
    ).

%%% Returns the limit between which the dog will circle a flock
%%% getCircleLimit(-CircleLimit)
%%%
getCircleLimit(CircleLimit) :-
    circleLimit(CircleLimit),!
    ;
    CircleLimit is 1.3,!

%%% Returns the cluster distance that is used when creating
%%% flocks. The default is 1 if no predicate is found
%%% getClusterDistance(-ClusterDistance)
%%%
getClusterDistance(ClusterDistance) :-
    clusterDistance(ClusterDistance),!
    ;
    ClusterDistance is 1,!

%%% Returns the minimum distance that is used when closing in
%%% on a sheep or flock. The default value is 2
%%% getMinDist(-MinDist)
%%%
getMinDist(MinDist) :-
    minDist(MinDist)
; MinDist is 2.
Appendix D

Checker

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the amount of primary flocks equals the
% integer that has been given as a parameter
% %
% % checkAmountOfPrimaryFlocks(+Frame, +Amount)
% %
% checkAmountOfPrimaryFlocks(Frame,Amount) :-
% getPrimaryFlocks(Frame,Flocks),
% length(Flocks,Length),
% Amount =:= Length.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the sheep is within the goal
% %
% % checkSheepInGoal(+Frame, +ID)
% %
% checkSheepInGoal(Frame,ID) :-
% posGoal(PosGoal),
% posSheep(Frame,PosSheep,ID),
% calculateDistance(PosGoal,PosSheep,Distance),
% Distance < 60.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the amount of flock matches the amount
% parameter
% %
% % checkAmountOfFlocks(+Frame, +Amount)
% %
% checkAmountOfFlocks(Frame,Amount) :-
% getFlocks(Frame,Flocks),
% length(Flocks,Length),
Amount =:= Length.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the dog is closer to the left side of the
% angle to the goal
%
% closerToLeftGoal(+Frame, +Flock)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
closerToLeftGoal(Frame,Flock) :-
  getFlockToDogAngle(Frame,Flock,DogAngle),
  getFlockToGoalAngle(Frame,Flock,GoalAngle),
  closerToLeft(DogAngle,GoalAngle).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the dog is closer to the left side of the
% angle to the sheep
%
% closerToLeftSheep(+Frame, +Flock, +ID)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
closerToLeftSheep(Frame,Flock,ID) :-
  getFlockToDogAngle(Frame,Flock,DogAngle),
  getFlockToSheepAngle(Frame,Flock,ID,SheepAngle),
  closerToLeft(DogAngle,SheepAngle).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gets a flock and an amount and evaluates to true if the
% amount of members in the flock is less than the received
% amount
%
% lessFlockMembersThan(+Flock, +Amount)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checkLessFlockMembersThan(Flock,Amount) :-
  length(Flock,Len), Len < Amount.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the dog is within the limit that is opposite
% of the flock to the goal. The limit is based on the amount
% of sheep
%
% checkDogWithinGoalLimitScale(+Frame, +Flock)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checkDogWithinGoalLimitScale(Frame,Flock) :-
  getFlockToDogAngle(Frame,Flock,DogAngle),
  getFlockToGoalAngle(Frame,Flock,GoalAngle),
  length(Flock,Len),
CircleLimit is 0.1 * Len,
\%write('Goal. Len: '), write(Len), write(' Limit: '), write(CircleLimit),nl,
GoalAngleInv is GoalAngle + \pi,
LeftLimit is GoalAngleInv + CircleLimit,
RightLimit is GoalAngleInv - CircleLimit,
checkWithinLimit(LeftLimit,RightLimit,DogAngle)
.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the dog is within the limit that is opposite
% of the flock to the goal
%
% checkDogWithinGoalLimit(+Frame, +Flock)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checkDogWithinGoalLimit(Frame,Flock) :-
  getFlockToDogAngle(Frame,Flock,DogAngle),
  getFlockToGoalAngle(Frame,Flock,GoalAngle),
  getCircleLimit(CircleLimit),
  GoalAngleInv is GoalAngle + \pi,
  LeftLimit is GoalAngleInv + CircleLimit,
  RightLimit is GoalAngleInv - CircleLimit,
  checkWithinLimit(LeftLimit,RightLimit,DogAngle)
.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the dog is within the limit that is opposite
% of the flock to the sheep. The limit is based on the amount
% of sheep
%
% checkDogWithinSheepLimitScale(+Frame, +Flock, +ID)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checkDogWithinSheepLimitScale(Frame,Flock,ID) :-
  getFlockToDogAngle(Frame,Flock,DogAngle),
  getFlockToSheepAngle(Frame,Flock,ID,SheepAngle),
  length(Flock,Len),
  CircleLimit is 0.1 * Len,
  SheepAngleInv is SheepAngle + \pi,
  LeftLimit is SheepAngleInv + CircleLimit,
  RightLimit is SheepAngleInv - CircleLimit,
  checkWithinLimit(LeftLimit,RightLimit,DogAngle)
.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the dog is within the limit that is opposite
% of the flock to the sheep
%
% checkDogWithinSheepLimit(+Frame, +Flock, +ID)
%%
% checkDogWithinSheepLimit(Frame,Flock,ID) :-
%  getFlockToDogAngle(Frame,Flock,DogAngle),
%  getFlockToSheepAngle(Frame,Flock,ID,SheepAngle),
%  getCircleLimit(CircleLimit),
%  SheepAngleInv is SheepAngle + pi,
%  LeftLimit is SheepAngleInv + CircleLimit,
%  RightLimit is SheepAngleInv - CircleLimit,
%  checkWithinLimit(LeftLimit,RightLimit,DogAngle).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Evaluates to true if there are any sheep not in the flock
% and returns false if there are
% %
% checkAnySheepNotInFlock(+Frame, +Flock)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checkAnySheepNotInFlock(Frame,Flock) :-
  getAllSheepNotInFlock(Frame,Flock,SheepNotInFlock),
  length(SheepNotInFlock,Len),
  Len > 0, !.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether two sheep are within a certain distance of
% each other
% %
% checkDistance(+Frame, +ID, +ID, +Distance)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
checkDistance(Frame,ID1,ID2,Distance) :-
  posSheep(Frame,(X1,Y1),ID1),
  posSheep(Frame,(X2,Y2),ID2),
  ID1 \= ID2,
  abs(X2 - X1) =< Distance,
  abs(Y2 - Y1) =< Distance.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks whether the first number is larger absolute
% %
% largerThanAbs(+XValue, +YValue)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
largerThanAbs(X,Y) :- (abs(X)) > (abs(Y)).
Appendix E

Helper

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns the first element
% first(+List, -FirstElement)
% first([X|_],X).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates the distance between a sheep and a dog
% calculateClosestSheepToDog(+Frame, +Sheep, -Closest)
% calculateClosestSheepToDog(_Frame,[ClosestSheep],ClosestSheep) :- !.
calculateClosestSheepToDog(Frame,[Sheep1,Sheep2|Tail],ClosestSheep) :-
  getSheepDistanceToDog(Frame,Sheep1,Dis1),
  getSheepDistanceToDog(Frame,Sheep2,Dis2),
  (Dis1 > Dis2 ->
   calculateClosestSheepToDog(Frame,[Sheep2|Tail],ClosestSheep)
  ;
   calculateClosestSheepToDog(Frame,[Sheep1|Tail],ClosestSheep)
  ).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate flocks using only sheep that have a primsheep predicate
% generatePrimaryFlocks(+Frame, -Flocks)
% :- dynamic primSheep/1.
generatePrimaryFlocks(Frame, Cs) :-
   ( getClusterDistance(Distance),
     once((posSheep(Frame,_, First), primSheep(First))),
     generatePrimaryFlocks(Distance, [[First]], Frame, Cs),
     !
   ;
   ( emptyList(Cs), !
   ).

emptyList([]).

generatePrimaryFlocks(Distance, SoFar, Frame, Cs) :-
   ( primarySheepNotInFlock(SoFar, Frame, Tentative),
     addToFlock(SoFar, Tentative, Distance, Frame, Updated)
     ->
     generatePrimaryFlocks(Distance, Updated, Frame, Cs)
     ;
     Cs = SoFar
   ).

primarySheepNotInFlock(Clusters, Frame, ID) :-
   posSheep(Frame,_,ID),
   primSheep(ID),
   maplist(notMember(ID), Clusters).

% Calculates the distance between two points
% calculateDistance(+Pos, +Pos, -Distance)
% calculateDistance((X1,Y1),(X2,Y2),Distance) :-
%   Distance is sqrt((X1-X2)^2 + (Y1-Y2)^2).

calculateFurthestSheepFromGoal(_Frame,
   [Sheep1,Sheep2|LS],Furthest) :-
   getSheepToGoalDistance(Frame,Sheep1,Dis1),
   getSheepToGoalDistance(Frame,Sheep2,Dis2),
   (
Dis1 > Dis2 ->
calculateFurthestSheepFromGoal(Frame,[Sheep1|LS],Furthest)
;
calculateFurthestSheepFromGoal(Frame,[Sheep2|LS],Furthest)
).
!

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates what the furthest flock is from the average when
% looking at all flocks and the goal
%
% calculateFurthestFlockFromAllFlocksAndGoal(+Frame, +List,
% %
% -Furthest)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
calculateFurthestFlockFromAllFlocksAndGoal(_Frame,[Furthest],Furthest).
calculateFurthestFlockFromAllFlocksAndGoal(Frame,[Flock1,Flock2|LS],Furthest) :-
  getFlockToAllFlocksAndGoalDistance(Frame,Flock1,Dis1),
  getFlockToAllFlocksAndGoalDistance(Frame,Flock2,Dis2),
  ( Dis1 > Dis2 ->
    calculateFurthestFlockFromAllFlocksAndGoal(Frame,[Flock1|LS],Furthest)
    ;
    calculateFurthestFlockFromAllFlocksAndGoal(Frame,[Flock2|LS],Furthest)
  )
.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates the closest sheep to a certain flock based on a
% list of sheep IDs
%
% calculateClosestSheepToFlockBasedOnList(+Frame, +Flock,
% %
% +SheepList,
% %
% -ClosestSheep)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
calculateClosestSheepToFlockBasedOnList(_Frame,_Flock,[Closest],Closest):-!.
calculateClosestSheepToFlockBasedOnList(Frame,Flock,[ID1,ID2|Tail],Closest) :-
  getSheepToFlockDistance(Frame,Flock,ID1,Dis1),
  getSheepToFlockDistance(Frame,Flock,ID2,Dis2),
  ( Dis1 > Dis2
    ->
    calculateClosestSheepToFlockBasedOnList(Frame,Flock,[ID2|Tail],Closest)
    ;
    calculateClosestSheepToFlockBasedOnList(Frame,Flock,[ID1|Tail],Closest)
  ).
APPENDIX E. HELPER

% Looks at a certain flock and looks what the furthest sheep away from the average position is
% %
% calculateFurthestSheepFromFlockInFlock(+Frame, +Flock, +Flock, -Furthest)
% %
calculateFurthestSheepFromFlockInFlock(_Frame,_Flock, [Furthest], Furthest).
calculateFurthestSheepFromFlockInFlock(Frame,Flock,[ID1,ID2|Ls],Furthest) :-
  getSheepToFlockDistance(Frame,Flock,ID1,Dis1),
  getSheepToFlockDistance(Frame,Flock,ID2,Dis2),
  (Dis1 > Dis2 ->
    calculateFurthestSheepFromFlockInFlock(Frame,Flock,[ID1|Ls],Furthest)
  ;
    calculateFurthestSheepFromFlockInFlock(Frame,Flock,[ID2|Ls],Furthest)
  ).

% Calculates the closest flock to the dog
% %
% closestGroup(+Frame, +FlockList, -ClosestFlock)
% %
calculateClosestFlockToDog(_Frame,[ClosestFlock],ClosestFlock) :- !.
calculateClosestFlockToDog(Frame,[Flock1,Flock2|Tail],ClosestFlock) :-
  getFlockDistanceToDog(Frame,Flock1,Dis1),
  getFlockDistanceToDog(Frame,Flock2,Dis2),
  (Dis1 > Dis2 ->
    calculateClosestFlockToDog(Frame,[Flock2|Tail],ClosestFlock)
  ;
    calculateClosestFlockToDog(Frame,[Flock1|Tail],ClosestFlock)
  ).

% Takes a list of positions and calculates the average
% %
% averagePositions(+PosList, -PosAvg)
% %
averagePositions(PosList,(XAvg,YAvg)) :-
  sumPositions(PosList,(XSum,YSum)),
  length(PosList,Len),
  XAvg is XSum / Len,
  YAvg is YSum / Len, !.
% Takes a list of positions and sums them
% sumPositions(+PosList, -PosSum)
%
sumPositions([], (0,0)).
sumPositions([(X,Y) | Tail], (XSum,YSum)) :-
    sumPositions(Tail,(XRest,YRest)),
    XSum is XRest + X,
    YSum is YRest + Y,!.

% Checks whether the first angle is closer to the left side
% of the opposite of the second angle
% closerToLeft(+Angle1, +Angle2)
%
closerToLeft(Angle1,Angle2) :-
    Angle2Inv is Angle2 + pi,
    reducedAngle(Angle2Inv,Angle2InvNorm),
    ( Angle1 < Angle2InvNorm
     ->
         Left is 2 * pi - Angle2InvNorm + Angle1,
         Right is Angle2InvNorm - Angle1
     ;
         Left is Angle1 - Angle2InvNorm,
         Right is 2 * pi - Angle1 + Angle2InvNorm
    ),
    reducedAngle(Left,LeftNorm),
    reducedAngle(Right,RightNorm),
    LeftNorm < RightNorm.

% Checks whether the given angle is between the left limit
% and the right limit. If it is it evaluates to true, it it
% is not it evaluates to false
% checkWithinLimit(+LeftLimit, +RightLimit, +Angle)
%
checkWithinLimit(LeftLimit,RightLimit,DogAngle) :-
    reducedAngle(LeftLimit,LeftLimitNorm),
    reducedAngle(RightLimit,RightLimitNorm),
    reducedAngle(DogAngle,DogAngleNorm),
    ...
(LeftLimitNorm > RightLimitNorm)
    ->
    DogAngleNorm < LeftLimitNorm,
    DogAngleNorm > RightLimitNorm
;
(    DogAngleNorm < LeftLimitNorm
    ;
    DogAngleNorm > RightLimitNorm
)
),!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Changes an angle so that it is between 0 and 2pi
% reducedAngle(+Angle, -NewAngle)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
reducedAngle(Angle,ReducedAngle) :-
    (    Angle > (2*pi),
        NewAngle is Angle - 2*pi,
        reducedAngle(NewAngle,ReducedAngle),!
    )
    ;
    (    Angle < 0,
        NewAngle is Angle + 2*pi,
        reducedAngle(NewAngle,ReducedAngle),!
    )
    ;
    (    ReducedAngle is Angle,!
    )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates the angle between two coordinates in pi rad
% angle(+Pos1, +Pos2, -Angle)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
angle(0,0,Angle) :-
    Angle = 0
    .

angle(0,Y,Angle) :-
    (    Y > 0
    )

% angle(0,0,Angle) :-
%    Angle = 0
%
angle(X,0,Angle) :-  
    (  
      X > 0  
      ->  
        Angle is 0 * pi  
        ;  
        Angle is 1 * pi  
    )  .

angle((X1,Y1),(X2,Y2),Angle) :-  
  XDif is X2 - X1,  
  YDif is Y2 - Y1,  
  (  
    (XDif =:= 0; YDif =:= 0)  
    ->  
      angle(XDif,YDif,Angle),!  
    ;  
    (  
      XDif > 0  
      ->  
        (  
          YDif > 0  
          ->  
            Angle is atan(XDif/YDif) + 1.5 * pi  
            ;  
            Angle is ((-1) * atan(YDif/XDif))  
          )  
        ;  
        (  
          YDif > 0  
          ->  
            Angle is -1 * atan(YDif/XDif) + pi  
            ;  
            Angle is atan(XDif/YDif) + 0.5 * pi  
          )  
      )  
      ,!  
    )  .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculates the radius of a certain flock based on the sheep that is furthest away from the middle
% calculateFlockRadius(+Frame, +Flock, -Radius)
% calculateFlockRadius(Frame,Flock,Radius) :-
% getFurthestSheepFromFlockInFlock(Frame,Flock,Furthest),
% getSheepToFlockDistance(Frame,Flock,Furthest,Radius).

% Calculates the distance between a certain sheep and flock in a frame
% calculateSheepToFlockDistance(+Frame, +Flock, +ID, -Distance)
calculateSheepToFlockDistance(Frame,Flock,ID,Distance) :-
  posSheep(Frame,(X,Y),ID),
  getFlockAveragePosition(Frame,Flock,(XAvg,YAvg)),
  calculateDistance((X,Y),(XAvg,YAvg),Distance),!.
%Distance is sqrt(abs(XAvg-X)^2 + abs(YAvg-Y)^2),!.

% Creates a list of all sheep that are not in a certain flock
% getAllSheepNIF(+Frame, +Cluster, +ID, -IDs)
% getAllSheepNIF(Frame,_,ID,[]) :-
%  \+ posSheep(Frame,_,Pos,ID).
% getAllSheepNIF(Frame,Cluster,ID,IDs) :-
%  posSheep(Frame,_,Pos,ID),
%  memberchk(ID,Cluster),
%  ID2 is ID+1,
%  getAllSheepNIF(Frame,Cluster,ID2,IDs).
% getAllSheepNIF(Frame,Cluster,ID,[ID|IDs]) :-
%  posSheep(Frame,_,Pos,ID),
%  \+ memberchk(ID,Cluster),
%  ID2 is ID+1,
%  getAllSheepNIF(Frame,Cluster,ID2,IDs).

% Divides the x and y coordinates by the length of the cluster it receives
%
% avgPos(+Cluster, +Pos, -AvgPos)
% %
avgPos(C,(X,Y),(XAvg,YAvg)) :-
    length(C,Len),
    XAvg is X / Len,
    YAvg is Y / Len.
%
% Sums all x and y coordinates for a single cluster and % returns a list which contains both %%
% sumPosSheep(+Frame, +Cluster, -SumPos)
%
sumPosSheep(_,[],(0,0)).
sumPosSheep(Frame,[ID|IDs],(XSum,YSum)) :-
    posSheep(Frame,(X,Y),ID),
    sumPosSheep(Frame,IDs,(XSum1,YSum1)),
    XSum is XSum1 + X,
    YSum is YSum1 + Y, !.
%
% Generates a list of flocks based on a maximum distance %
% % generateFlocks(+Frame, -Flocks)
%
generateFlocks(Frame, Cs) :-
    getClusterDistance(Distance),
    once(posSheep(_,_,First)),
    generateFlocks(Distance, [[First]], Frame, Cs).

generateFlocks(Distance, SoFar, Frame, Cs) :-
    (    sheepNotInFlock(SoFar, Frame, Tentative),
        addToFlock(SoFar, Tentative, Distance, Frame, Updated)
    ->
        generateFlocks(Distance, Updated, Frame, Cs)
    ;
        Cs = SoFar
    ).
%
sheepNotInFlock(Clusters, Frame, ID) :-
    posSheep(Frame,_,ID),
    maplist(notMember(ID), Clusters).
notMember(E, L) :- \+ memberchk(E, L).

addToFlock([C|Cs], Tentative, Distance, Frame, Updated) :-
    member(Id, C),
    checkDistance(Frame, Id, Tentative, Distance), !,
    Updated = [[Tentative|C]|Cs].

addToFlock([C|Cs], Tentative, Distance, Frame, [C|Rs]) :-
    addToFlock(Cs, Tentative, Distance, Frame, Rs).

addToFlock([], Tentative, _Distance, _Frame, [[Tentative]]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Limits a value to a certain max value. If the value is not
% larger than the max the same value is returned. If it is
% then the max value is returned.
% %
% fixedValue(+Value, -FixedValue, +MaxValue)
% %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fixedValue(V,VFixed,MaxDist) :-
    \+ largerThanAbs(V,MaxDist),
    VFixed is V.

fixedValue(V,VFixed,MaxDist) :-
    largerThanAbs(V,MaxDist),
    numberOne(V,VOne),
    VFixed is MaxDist * VOne.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This converts any number to either 1, or -1, depending on
% whether the number was positive or negative
% %
% numberOne(+Value, -ValueOne)
% %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
numberOne(X,Y) :-
    Y is X / abs(X).
Appendix F

Behaviours

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% When run the herding dog always picks the furthest sheep
% from the flock and brings them together before again
% finding the furthest sheep
%
% furthestSheepFromFlock(+Frame, -Pos)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
furthestSheepFromFlock(FrameNew, Pos) :-
  Frame is FrameNew - 1,
  {
    checkAmountOfFlocks(Frame, 1)
    ->
      (getClosestFlockToDog(Frame, Flock),
       circleBehindFlockFacingGoal(Frame, Flock, Target))
    ;
    {
      checkAmountOfPrimaryFlocks(Frame, 1)
      ->
        getPrimaryFlocks(Frame, PrimaryFlocks),
        first(PrimaryFlocks, FirstFlock),
        getFurthestSheepFromFlockNotInFlock(Frame, FirstFlock, Furthest),
        setPrimaryFlockSheep(Furthest),
        moveBehindFlockFacingTarget(Frame, FirstFlock, Furthest, Target)
      ;
      {
        checkAmountOfPrimaryFlocks(Frame, 0)
        ->
          getClosestSheepToDog(Frame, ClosestSheep),
          setPrimaryFlockSheep(ClosestSheep),
          getFurthestSheepFromFlockNotInFlock(Frame, [ClosestSheep],
FurthestSheep),
setPrimaryFlockSheep(FurthestSheep),
moveBehindFlockFacingTarget(Frame,[ClosestSheep],FurthestSheep, Target)
;
getClosestPrimaryFlockToDog(Frame,ClosestPrimaryFlock),
getClosestPrimarySheepToFlockNotInFlock(Frame,ClosestPrimaryFlock,
Closest),
moveBehindFlockFacingTarget(Frame,ClosestPrimaryFlock,Closest,
Target)
)
)
moveToLocation(Frame,Target,Pos).

pushFurthestSheep(NewFrame,Pos) :-
Frame is NewFrame - 1,
getFocusSheepFurthest(Frame,ID),
circleBehindFlockFacingGoal(Frame,[ID],Target),
movableLocation(Frame,Target,Pos).

pushToGoalWithCircleGathering(FrameNew,Pos) :-
Frame is FrameNew - 1,
( checkAmountOfFlocks(Frame,1)
  ->
  getClosestFlockToDog(Frame,Flock),
circleBehindFlockFacingGoal(Frame,Flock,Target)
  ;
  getAllSheep(Frame,AllSheep),
circleFlock(Frame,AllSheep,clockwise,Target)
),
movableLocation(Frame,Target,Pos).
% Runs the lien algorithm and returns a new location for the
% dog to move to. The circle limit is based on
% the amount of sheep that is in the flock the dog is
% following
%
lienScale(+Frame, -Position)
%
lienScale(FrameNew,Pos) :-
    Frame is FrameNew - 1,
    getClosestFlockToDog(Frame,ClosestFlock),
    (    checkAnySheepNotInFlock(Frame,ClosestFlock)
        ->
        getClosestSheepToFlockNotInFlock(Frame,ClosestFlock,Sheep),
        circleBehindFlockFacingSheepScale(Frame,ClosestFlock,Sheep,Target)
    ;
        circleBehindFlockFacingGoalScale(Frame,ClosestFlock,Target)
    ),
    moveToLocation(Frame,Target,Pos),
    !.
%
% Runs the lien algorithm and returns a new location for the
% dog to move to. If the flock the dog is following has less
% than three members the dog will not circle around the flock
% but simply push it
%
lienv2(+Frame, -Position)
%
lienv2(FrameNew,Pos) :-
    Frame is FrameNew - 1,
    getClosestFlockToDog(Frame,ClosestFlock),
    (    checkAnySheepNotInFlock(Frame,ClosestFlock)
        ->
        getClosestSheepToFlockNotInFlock(Frame,ClosestFlock,Sheep),
        (            checkLessFlockMembersThan(ClosestFlock,3)
                ->
                moveBehindFlockFacingTarget(Frame,ClosestFlock,Sheep,Target)
            ;
                circleBehindFlockFacingSheep(Frame,ClosestFlock,Sheep,Target)
        )
    ;
        circleBehindFlockFacingGoal(Frame,ClosestFlock,Target)
    ),
moveToLocation(Frame, Target, Pos), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Runs the lien algorithm and returns a new location for the
% dog to move to
%
% lien(+Frame, -Position)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
lien(FrameNew, Pos) :-
    Frame is FrameNew - 1,
    getClosestFlockToDog(Frame, ClosestFlock),
    (    checkAnySheepNotInFlock(Frame, ClosestFlock)
        ->
        getClosestSheepToFlockNotInFlock(Frame, ClosestFlock, Sheep),
        write(ClosestFlock), write(‘\n’), write(Sheep),
        circleBehindFlockFacingSheep(Frame, ClosestFlock, Sheep, Target)
    ;
    circleBehindFlockFacingGoal(Frame, ClosestFlock, Target)
    ),
    moveToLocation(Frame, Target, Pos), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Runs the miki algorithm and returns a new location for the
% dog to move to
%
% miki(+Frame, -Position)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
miki(FrameNew, Pos) :-
    Frame is FrameNew - 1,
    getClosestFlockToDog(Frame, ClosestFlock),
    (    checkAnySheepNotInFlock(Frame, ClosestFlock)
        ->
        getClosestSheepToFlockNotInFlock(Frame, ClosestFlock, Sheep),
        moveBehindFlockFacingTarget(Frame, ClosestFlock, Sheep, Target)
    ;
    moveBehindFlockFacingGoal(Frame, ClosestFlock, Target)
    ),
    moveToLocation(Frame, Target, Pos), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If run the dog will circle around all sheep
%

circleAllSheep(FrameNew, Pos) :-
    Frame is FrameNew - 1,
    getAllSheep(Frame, AllSheep),
    circleFlock(Frame, AllSheep, clockwise, Target),
    moveToLocation(Frame, Target, Pos).
Appendix G

Java Info

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns all relevant info that is needed in java to draw
% the representation of the area that the dog will circle.
% In this case the limit is based on the amount of sheep
%
% getAllAngleInfoScale(+Frame, -Info)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllAngleInfoScale(Frame,AllInfo) :-
  getClosestFlockToDog(Frame,Flock),
  getFlocks(Frame,Flocks),
  getFlockAveragePosition(Frame,Flock,AveragePosition),
  getFlockRadius(Frame,Flock,Radius),
  mergeAllFlockInfo(Flock,AveragePosition,Radius,FlockInfo),
  getFlockToDogAngle(Frame,ClosestFlock,DogAngle),
  length(Flock,Len),
  CircleLimit is 0.1 * Len,
  %write('CircleLimit: '), write(CircleLimit), nl,
  length(Flocks,AmountOfFlocks),
  ( AmountOfFlocks > 1
    ->
      getClosestSheepToFlockNotInFlock(Frame,ClosestFlock,ClosestSheep),
      getFlockToSheepAngle(Frame,ClosestFlock,ClosestSheep,TargetAngle)
    ;
      getFlockToGoalAngle(Frame,Flock,TargetAngle)
  ),
  TargetAngleInv is TargetAngle + pi,
  reducedAngle(TargetAngleInv,TargetAngleInvNorm),
  mergeAllAngleInfo(FlockInfo,DogAngle,TargetAngleInvNorm,CircleLimit,AllInfo)
.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns all relevant info that is needed in java to draw

101
% the representation of the area that the dog will circle
%
% getAllAngleInfo(+Frame, -Info)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllAngleInfo(Frame,AllInfo) :-
    getClosestFlockToDog(Frame,Flock),
    getFlocks(Frame,Flocks),
    getFlockAveragePosition(Frame,Flock,AveragePosition),
    getFlockRadius(Frame,Flock,Radius),
    mergeAllFlockInfo(Flock,AveragePosition,Radius,FlockInfo),
    getFlockToDogAngle(Frame,ClosestFlock,DogAngle),
    getCircleLimit(CircleLimit),
    length(Flocks,AmountOfFlocks),
    (AmountOfFlocks > 1 ->
        getClosestSheepToFlockNotInFlock(Frame,ClosestFlock,ClosestSheep),
        getFlockToSheepAngle(Frame,ClosestFlock,ClosestSheep,TargetAngle) ;
        getFlockToGoalAngle(Frame,Flock,TargetAngle)
    ),
    TargetAngleInv is TargetAngle + pi,
    reducedAngle(TargetAngleInv,TargetAngleInvNorm),
    mergeAllAngleInfo(FlockInfo,DogAngle,TargetAngleInvNorm,CircleLimit,AllInfo).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Merges the flock info and all required angles into a single tuple so that this info can be requested with a single query
%
% mergeAllAngleInfo(+FlockInfo, +DogAngle, +TargetAngle, +CircleLimit, -Info)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mergeAllAngleInfo(FlockInfo,DogAngle,TargetAngle,CircleLimit, (FlockInfo,split,DogAngle,TargetAngle,CircleLimit)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Returns all sheep and the average position and radius of all sheep
%
% getAllSheepFlockInfo(+Frame,- FlockInfo)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getAllSheepFlockInfo(Frame,AllFlockInfo) :-
    getAllSheep(Frame,AllSheep),
    getFlockAveragePosition(Frame,AllSheep,FlockAvg),
getFlockRadius(Frame, AllSheep, FlockRadius),
mergeAllFlockInfo(AllSheep, FlockAvg, FlockRadius, AllFlockInfo).

% Returns all flocks and their average position and radius
% getAllFlockInfo(+Frame, -FlockInfo)
%
getAllFlockInfo(Frame, AllFlockInfo) :-
  getFlocks(Frame, Flocks),
  maplist(getFlockAveragePosition(Frame), Flocks, FlockAvgList),
  maplist(getFlockRadius(Frame), Flocks, FlockRadiusList),
  maplist(mergeAllFlockInfo, Flocks, FlockAvgList, FlockRadiusList, AllFlockInfo).

% Takes a flock, its average position and its radius and Puts
% it in one tuple
% mergeAllFlockInfo(+Flock, +Avg, +Radius, -Info)
%
mergeAllFlockInfo(Flock, FlockAvg, FlockRadius, (Flock, FlockAvg, FlockRadius)).
Appendix H

Targets

circleBehindFlockFacingGoalScale(+Frame, +Flock, -Target)

circleBehindFlockFacingGoalScale(Frame,Flock,Target) :-
    Limit = checkDogWithinGoalLimitScale(Frame,Flock),
    Left = closerToLeftGoal(Frame,Flock),
    circleBehindFlock(Frame,Flock,Limit,Left,Target).

circleBehindFlockFacingGoal(+Frame, +Flock, -Target)

circleBehindFlockFacingGoal(Frame,Flock,Target) :-
    Limit = checkDogWithinGoalLimit(Frame,Flock),
    Left = closerToLeftGoal(Frame,Flock),
    circleBehindFlock(Frame,Flock,Limit,Left,Target).

circleBehindFlockFacingSheepScale(+Frame, +Flock, +Sheep, -Target)

circleBehindFlockFacingSheepScale(Frame,Flock,Sheep,Target) :-
    Limit = checkDogWithinSheepLimitScale(Frame,Flock,Sheep),
Left = closerToLeftSheep(Frame,Flock,Sheep),
circleBehindFlock(Frame,Flock,Limit,Left,Target).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Moves the dog behind the flock flacing a certain sheep and
% circles there
%
% circleBehindFlockFacingSheep(+Frame, +Flock, +Sheep, -Target)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
circleBehindFlockFacingSheep(Frame,Flock,Sheep,Target) :-
  Limit = checkDogWithinSheepLimit(Frame,Flock,Sheep),
  Left = closerToLeftSheep(Frame,Flock,Sheep),
  circleBehindFlock(Frame,Flock,Limit,Left,Target).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Circles a certain flock between a limit in a certain
% direction
%
% circleBehindFlock(+Frame, +Flock, +Limit, +Left, -Target)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
circleBehindFlock(Frame,Flock,Limit,Left,Target) :-
  ( %
    \+ Limit,
    ( %
      Left
      ->
      setDirection(clockwise)
      ;
      setDirection(counterClockwise)
    ),
    circleFlock(Frame,Flock,Target)
  )
  ;
  circleFlock(Frame,Flock,Target)
),
setTargetLocation(Frame,Target).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generates a new position for when the dog is circling a
% certain flock, the direction is based on a predicate
%
% circleFlock(+Frame, +Flock, -Pos)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
circleFlock(Frame,Flock,Pos) :-
  direction(Direction),
  circleFlock(Frame,Flock,Direction,Pos).

circleFlock(Frame,Flock,clockwise,Pos) :-
  circleFlock(Frame,Flock,-1,Pos).

circleFlock(Frame,Flock,counterClockwise,Pos) :-
  circleFlock(Frame,Flock,1,Pos).

circleFlock(Frame,Flock,Direction,Pos) :-
  getFlockRadiusWithMinDistance(Frame,Flock,Radius),
  getFlockAveragePosition(Frame,Flock,AvgPos),
  getFlockToDogAngle(Frame,Flock,Angle),
  NewAngle is Angle + Direction * 1/12 * pi + pi,
  getDogTarget(AvgPos,NewAngle,Radius,Pos),
  setTargetLocation(Frame,Pos).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generates the new target location for behind a position
% for the dog so that it can move it to the goal
% getDogTarget(+FlockPosition, +GoalPosition, +Angle,
%   +Radius, -TargetPosition)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getDogTarget((XAvg,YAvg),Angle,Radius,(XTar,YTar)) :-
  XDif is cos(Angle + pi) * Radius,
  YDif is sin(Angle + pi) * Radius,
  XTar is XAvg + XDif,
  YTar is YAvg - YDif,!

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Puts the target location directly behind a flock facing a
% certain sheep and saves this target
% moveBehindFlockFacingTarget(+Frame, +Flock, +ID, -Target)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
movedBehindFlockFacingTarget(Frame,Flock,ID,TargetPos) :-
  getFlockAveragePosition(Frame,Flock,AvgPos),
  getFlockToSheepAngle(Frame,Flock,ID,Angle),
  getFlockRadiusWithMinDistance(Frame,Flock,Radius),
  getDogTarget(AvgPos,Angle,Radius,TargetPos),
  setTargetLocation(Frame,TargetPos),
setTargetLocation(Frame,TargetPos),!
.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Puts the target location directly behind a flock facing
% the goal and saves this target
%
% moveBehindFlockFacingGoal(+Frame, +Flock, -Target)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
moveBehindFlockFacingGoal(Frame,Flock,TargetPos) :-
  getFlockAveragePosition(Frame,Flock,AvgPos),
  getFlockToGoalAngle(Frame,Flock,Angle),
  getFlockRadiusWithMinDistance(Frame,Flock,Radius),
  getDogTarget(AvgPos,Angle,Radius,TargetPos),
  setTargetLocation(Frame,TargetPos),!
.\n
Appendix I

Movers

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Moves the dog closer to the target location
% % moveToLocation(+Frame, +Target, -NewPosition)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
moveToLocation(Frame,(XTarget,YTarget),(X,Y)) :-
  posDog(Frame,(XDog,YDog)),
  XDif is XTarget - XDog,
  YDif is YTarget - YDog,
  numberOne(XDif,XOne),
  numberOne(YDif,YOne),
  maxDist(MaxDist),
  (largerThanAbs(XDif,YDif) ->
   YMove is MaxDist/abs(XDif) * abs(YDif) * YOne,
   fixedValue(YMove,YMoveFixed,MaxDist),
   X is XDog + MaxDist * XOne,
   Y is YDog + YMoveFixed)
  ;
  XMove is MaxDist/abs(YDif) * abs(XDif) * XOne,
  fixedValue(XMove,XMoveFixed,MaxDist),
  X is XDog + XMoveFixed,
  Y is YDog + MaxDist * YOne),!.