Simulation and Behaviour Learning for Robocup SPL

Yongki Yusmanthia

Computer Science

August 24, 2011

Supervisor: Bernhard Hengst

Assessor: Maurice Pagnucco

Abstract

This thesis presents a simulator capable of running the behaviour code of 2011 rUNSWift and its development process. The simulator was later used to develop the supporter's behaviour and to do a series of reinforcement learning.

Acknowledgement

I would like to thank all those people who have supported me throughout this thesis. Firstly, I would like to thank my thesis supervisor Bernhard Hengst for his guidance and support.

I would also like to thank everyone in the 2011 rUNSWift team: Belinda Teh, Brock White, Carl Chatfield, David Claridge, Jimmy Kurniawan, and Sean Harris, for their help directly or indirectly with this thesis.

Contents

List of Figures

Chapter 1

Introduction

Standard Platform League is one of divisions in Robocup Soccer that use real life robots. The use of real robots requires a lot of time and effort to be put into developing better vision, locomotion, and localisation algorithms. As the result, less time is available to write the behaviour of the robots and the team strategy. The robots also tend to break easily, giving the team even less chance to test the robots' behaviour.

The aim of this thesis is to develop a simulator that takes in the behaviour code from 2011 rUNSWift and runs a soccer simulation as if the code is running on real robots, and utilise it to write the robot's behaviour and do a series of reinforcement learning.

1.1 Research Problems and Issues

The main challenge in developing a simulator capable of running the 2011 rUNSWift code lies in simulating the movement of the robot, and running the robot's behaviour code without the behaviour realising that it was not actually running on a real machine. The walk of the robot is difficult to simulate as we need to incorporate parts of the action generator and motion adapter sequence from the real robot into the simulator. The robot's behaviour code also poses a challenge as it needs to be created, run, and maintained somehow throughout the whole simulation. For the behaviour code to run, there are certain parts of the robotic architecture that needs to be simulated.

The development process faces a number of issues along the way. The biggest one is the fact that the whole code was still a work in progress and changed very rapidly. This requires the simulator to be implemented with a flexible design.

One of the goals of this thesis is to utilise the simulator to do a series of reinforcement learning. After considering a number of factors, it was decided that the reinforcement learning would be done on the *go to ball* skill which is one of the robot's behaviour skills. The challenge in this reinforcement learning lies in its state and action spaces as all the state and action spaces in this case are continuous.

1.2 Aim and Delimitations

The aim of this thesis would be to develop a simulator with the following features:

Flexibility: The simulator should be flexible since no assumption can be made about how it will be used. It should be able run any number of robots on both teams with any types of behaviour.

Ease of Use: One of the requirements for the simulator is that it has to take in the behaviour code from the 2011 rUNSWift. Therefore the users should be able to write the behaviour code from scratch and run it seamlessly on the simulator.

Realistic: Since the simulator will be used for the development of the behaviour code, the movement of the simulated robots must be realistic enough for the simulator to be any use.

Performance: The overall performance of the system must be at least as fast as a real time SPL game. Another desired feature would be the ability to increase the speed of the game to tens of magnitude faster.

This thesis attempts to develop a simulator that will only take in the Python behaviour code of 2011 rUNSWift. The simulator will not work with the previous teams' or other teams' code.

1.3 Methodology

I began the development of the simulator by firstly looking at other simulators in this area. It was then decided that it would be more convenient to develop our own simulator from scratch instead using one of simulators already available as a basis. Examining the structure and design of 2011 rUNSWift code, and picking a physics engine to use were the next thing to do.

The development then moved on to actually designing and constructing the simulator. The first version of the simulator was very simple. The robot was only able to walk around. More features, such as kicks, dribbles, and vision model, were gradually added throughout the development process to make the simulator more realistic.

One of the applications of the simulator is to utilise it to perform reinforcement learning on the behaviour skills. It started out very simple, with just a standard Q Learning algorithm. As the learning progressed, more issues were discovered which required the use of additional techniques.

1.4 Outline

This thesis report is divided into six chapters. The major ideas are presented in Chapter 2, 3, and 4. Each of these chapters is self-contained with its own background, methodology, results/evaluation, and future work/conclusion.

Chapter 1 presents the motivation, the structure, and the goals of this thesis.

Chapter 2 covers the development of the simulator. This chapter includes the structure of the simulator, the simulated movement of the robot, and the methods used to run the robot behaviour code in the simulator.

Chapter 3 describes the reinforcement learning done with the simulator. It discusses in details the problems that arose and the techniques used to get around them.

Chapter 4 describes the supporter behaviour. The simulator was used extensively during the development of this behaviour.

Chapter 5 wraps up the report and presents the conclusions derived from the previous chapters.

Finally chapter 6 contains all the references used in this report.

Chapter 2

Simulation

2.1 Background

In recent years, several simulators have been developed and made available to the public for research and education purposes. A decent number of these simulators are made specifically to run robot soccer simulations. This section will look at two of these simulators that are developed for the purpose of Robocup competition. This section will also look at the structure of 2011 rUNSWift code, and the physics engine used to develop the simulator.

2.1.1 Simspark

Simspark is a three-dimensional multiagent simulation system. It is built on the Spark application framework which is a generic physical simulator. Simspark enables its users to run simulations of soccer game with several robot models, including a virtual model of the Nao robot. The development of Simspark was

started in 2003 to provide a platform for Robocup 3D Soccer Simulation League. Since then, it has grown significantly and now supports humanoid players with articulated bodies.

Simspark simulates the environment in very high details. It uses Ruby language and text-based RSG files to generate the field. Simspark models the Nao robot quite accurately, with 22 hinges in each robot's body. The robots are also modelled with simulated cameras and other sensors, including accelerometers, gyroscopes, touch, and sonar sensors [Bea10].

Simspark consists of a simulation server that communicates with agents via TCP or UDP. The agents are developed independently by the users according to the purpose of the simulation. This structure allows the users to develop the agents in any language they prefer, but also introduces transmission delay in exchanging the TCP/UDP messages.

The biggest problem with Simspark is the huge amount of computing resource it requires. Due to the highly detailed simulation, Simspark needs several machines with several cores in each machine just to run a soccer simulation with 8 robots in real time. It was decided that Simspark was too processor-heavy for our needs.

2.1.2 2D Simulation League RCSS

RCSS (RoboCup Soccer Simulator) is a system that enables autonomous agents consisting of programs written in various languages to play a match of soccer against each other. It is the simulation software used for Robocup 2D Simulation League.

RCSS consists of a simple piece of software, soccerserver, that communicates

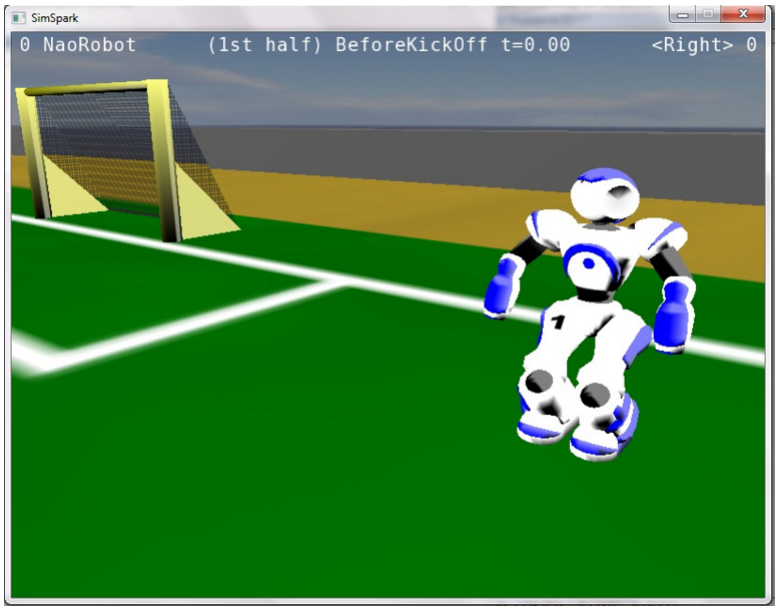

Figure 2.1: Soccer simulation in Simsparks

with agents via UDP/IP messages, and soccermonitor that displays the virtual field of the game on the monitor [Ste03]. The agent receives visual and auditory sensor information from the server via the UDP socket.

Unlike Simspark, RCSS does not model the body of the whole robot. Instead, each agent is represented with a simple body, a head, an audio sensor, stamina, and a simple 2D vision system. The movement of the agents are also much simpler. As the result of this design, RCSS requires significantly less computing power than Simspark, and thus is able to run more agents with faster speed.

2.1.3 Architecture of 2011 rUNSWift

The code of 2011 rUNSWift builds on the work of the previous year's team. The architecture of rUNSWift is task hierarchy for a multi agent team of four

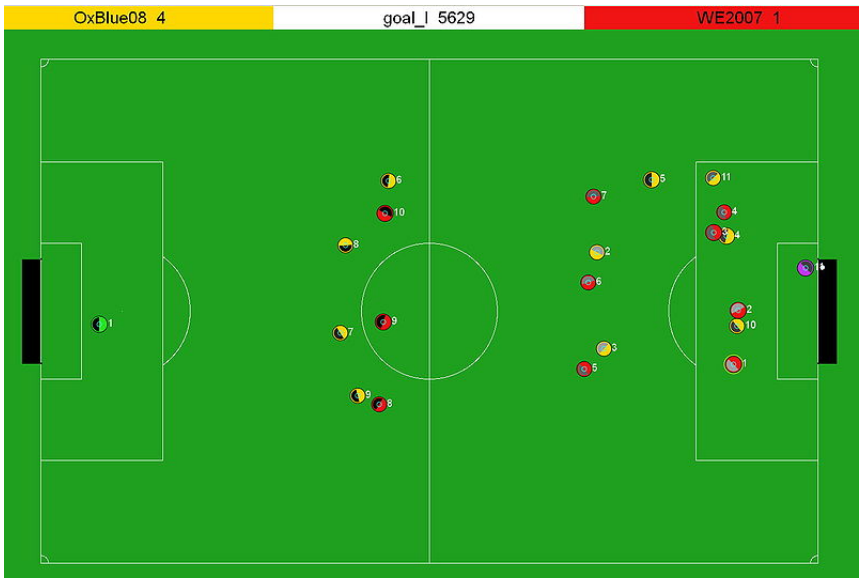

Figure 2.2: 11 vs 11 game in RCSS

Nao robots. There is no central controller in this architecture, meaning that each robot would probably have a different view of the world [RHH+10].

The rUNSWift architecture consists of several states of different levels. The high level states such as the game-controller would invoke lower level states to either execute or invoke other lower level states.

The rUNSWift 2011 code consists of the main program which would then spawn 6 other threads: Perception, Motion, Off-Nao Transmitter, Nao Transmitter, Nao Receiver, and GameController Receiver. The perception thread is responsible for processing images, localising, and deciding actions using the behaviour module. The motion thread is a real-time thread that computes appropriate joint values using the current action command and sensor values.

Some of the robot modules, such as vision, behaviour and localisation, share a common data structure called the *Blackboard*. The blackboard's primary

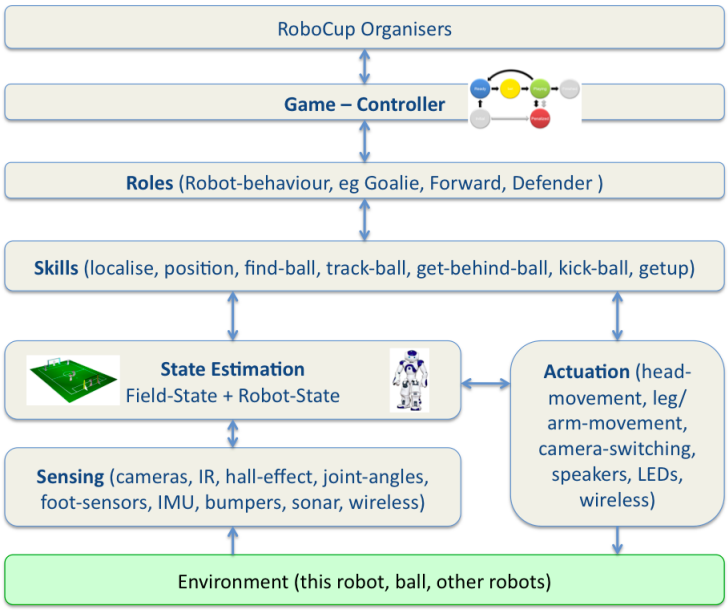

Figure 2.3: rUNSWift robotic architecture [RHH+10]

function is to facilitate inter-module communication. The blackboard is friends with each of the module adapters so that they can access blackboard privates.

2.1.4 PyBox2D Physics Engine

PyBox2D is a two dimensional rigid body simulation library for games. Developers can use it to make objects in their games move in more believable ways. PyBox2D is powered by Box2D library which is written in C++ and bound to Python by SWIG.

The geometrical objects in PyBox2D are called shape, while a chunk of matter is called body. A shape and a body are binded together by a fixture. The bodies have various properties such as velocities and frictions that can be manually set by the developers [Cat09].

PyBox2D allows the developers to decide what to do when two objects collide

with each other. The restitution values of the two objects will decide how much the two objects will bounce off each other. A value of zero means the object will not bounce, while a value of one means the object will be exactly reflected. The two restitution values are combined with the following formula:

$$restitution = max(shape1.restitution, shape2.restitution)$$

This enables us to determine how much object in the simulation, such as the ball, will bounce when it collides with other objects.

2.2 Methodology

2.2.1 Design and Structure of Simulator

The structure of the runswift code makes it easy to fetch the action command request. Every call to the *tick()* function of instances of the skills will return an action command request which consists of head, body, and LED requests. This action command request can be either processed straight away or put into blackboard. The main problem lies in creating, running, and maintaining the skills that will return those requests. During the development of the simulator, all the skills were written entirely in a high level dynamic language called *Python*. The easiest way to get around this is to write the entire simulator in Python. The problem with writing the simulator in Python is that the programs written in the language tend to be a bit slower. Another option is to write the simulator in other languages like C or C++ and use a language wrapper like SWIG which would make it more complicated. Considering the unlikeliness of the design of the behaviour code to change at the time, it was decided that using Python for the simulator was the best way to go.

Another critical design decision is how to model the robot. As we have seen in the previous section, Simspark models the Nao robot in great details which makes it realy slow, while RCSS has a very simple design which is not good enough for the purpose of this thesis. To compensate for this, the simulator only models the feet of the robot instead of the whole body. This design is able to generate realistic movement on the field while not consuming too much computing power.

The simulator is divided into several classes. The main class is *simulator* class which manages the agents, the ball, the environment, and the simulation loop. It is also the class that is created to start the simulation. Each robot is represented by an *agent* class which contains two instances of *foot* class. The simulator class also has an instance of the *ball* class which is responsible to model the ball and its movement. All the agents and the ball are contained in the *world* class which would update every object in the game during the simulation cycle.

A simulation loop in the simulator consists of the cycle in Figure 2.4. The structure of the simulation cycle enables the simulator to increase the speed of the game. The updateWorld function would tell the world to step 1/30 second. 1/30 second is chosen because rUNSWift code aims to run at that speed on the real Nao robot. A timer would make sure than one run through the cycle would take at least 1/30 second. If the simulation_step is set to 5, all the update functions would be called 5 times before the screen is rendered, meaning the simulation is five times faster. If the simulation_step is set to 1, then the simulation would run at the normal speed.

Another model that was considered was a structure similar to that of Simsparks which enables the agents to be implemented in any languages that supports UDP or TCP. However, such design would limit the speed of the simulation as

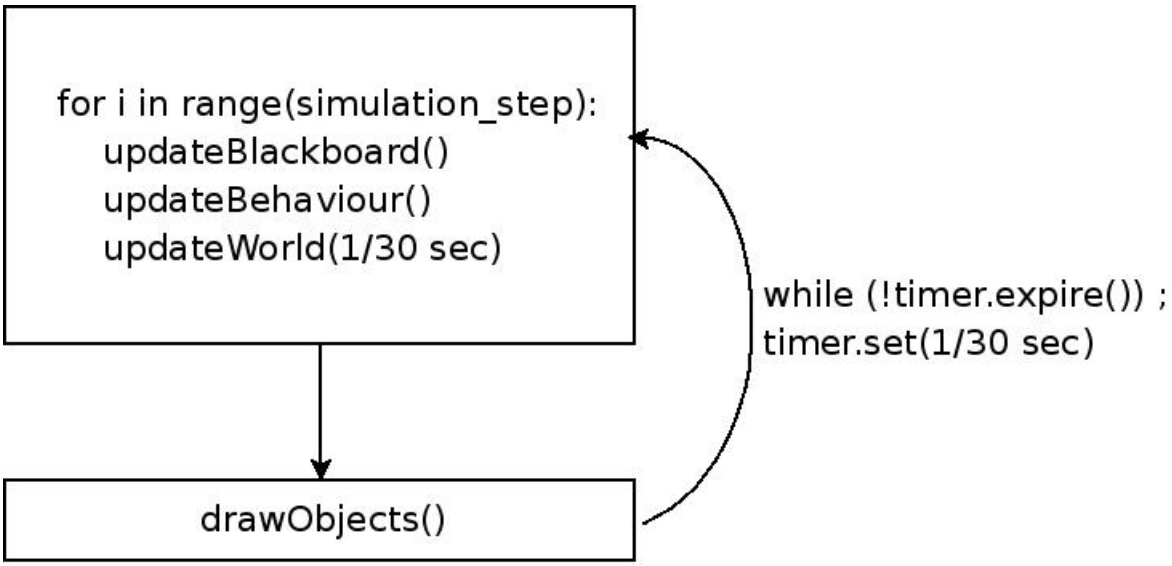

Figure 2.4: The simulation cycle

it introduces a new bottleneck which is the delay in transmitting the UDP/TCP messages.

2.2.2 The robot's walk

Walk Basics

Biped is an open kinematic chain consisting of at least two subchains called legs and often a subchain called the torso. One or both legs may be in contact with the ground. The leg that is in contact with the ground is called the stance leg, and the other leg is called the swing leg [Wes07].

Fast Walk

The walk used in this simulator is based on Fast Walk which is an omni-

directional dynamically stabilised closed-loop bipedal robot motion [RHH+10]. The process to generate the walk is as follow:

1. Get the new ForwardL, ForwardR, Left, Turn, T, and t values. These values are passed to the Python port of the WalkCycle class developed by Brock White [Whi11].
2. The WalkCycle class would generate the current position of the feet relative to their previous position.
3. Compare the current position of the feet with the previous position to get the displacement.
4. Calculate the speed of the feet needed to achieve the displacement.

2.2.3 The robot's kick and dribble

When the development of the simulator reached its later stage, the kick and dribble of the real robot had not been finalised yet. Because of this, it was decided that the kick and dribble in the simulator would be hard coded instead. This design would allow the speed and the power of the kick to be tuned according to the kick and dribble used in the real robot.

The concepts of the kick and dribble in the simulation are very similar. Both the kick and the dribble are divided into 3 phases:

Phase1 This is when the robot adjusts itself to prepare for the kick. The robot would stop moving, and depending on the type of kick used, the robot will do different actions such as shifting its centre of mass onto the

non-kicking leg. Since the simulator only models the feet, the robot in the simulator would simply stop moving for a short period of time.

Phase2 The kicking leg is lifted into the air. The swing of the kick is executed, and the robot would then return the kicking leg to its previous position.

Phase3 The robot would prepare itself to move again. Depending on the type of kick used, the robot will do different actions such as transferring its centre of mass back to both feet. Since the simulator only models the feet, the robot in the simulator would simply stop moving for a short period of time.

The difference between the kick and the dribble lies in the power of swing and the time required to execute it. The kick has a lot more power, but takes longer to execute.

2.2.4 Collision detection and resolution

One of the important parts of a multi-agent simulation system is the collision detection and resolution. In real life SPL games, the Nao robots collide with each other all the time. Two robots would collide, and depending on the scenario, different actions would be taken by the referee. The robots might collide when they are contesting for the ball. In this case, the referee would let the game continue. The robots might also collide when they are not contesting the ball, but rather caused by poor vision or localisation system. In this case, the referee would penalise the pushing robot. Upon closer observation, it can be seen that

player pushing and falling over happens all the time in SPL games. It takes up a lot of the game time, and is a big part of the game.

Due to the simulator modelling only the feet of the robot, the collision and pushing between two robots' bodies and its resulting actions such as falling over cannot be simulated. This issue raises a need to develop a heuristic to detect and resolve collisions between robots.

A simple heuristic was first used for the beta version of the simulator. A collision happens when two robots get too close to each other. The heuristic divides collisions into two cases: when two robots are facing each other, and when two robots are facing the same way. When two robots are facing each other, it is assumed that both of them are contesting for the ball. Therefore the robot closer to the ball would win and get to move while the other robot is stopped. When two robots are facing the same way, the robot that is in front would get to move while the other robot's movement is stopped.

The final heuristic is more complicated and involves penalising the offending robot. A collision is still defined as when two robots get too close to each other. However, instead of simply dividing all the collisions into two cases, the simulator will use a decision tree to decide whether to penalise the robot or not. The decision tree is shown in figure 2.5.

A very simple algorithm, Algorithm 2.1, is used to determine whether the two colliding robots are facing the same way or not. It will change the range of the robots' angles from [-pi, pi] to [0, 2 * pi] and take the difference between the two angles. If the difference is less than pi / 2.0, then the robots are facing the same way.

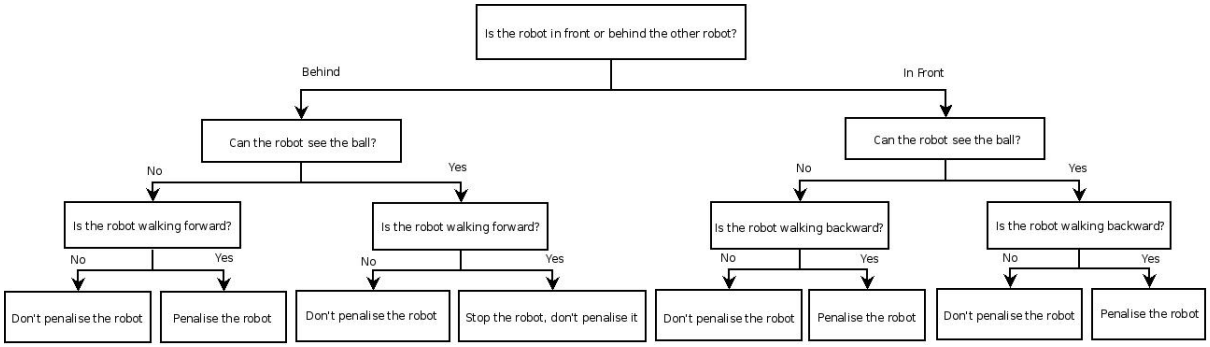

Figure 2.5: Decision tree for robot collision heuristic

Algorithm 2.1 Two robots' orientation

```
angle1 = robot1.angle
angle2 = robot2.angle
if angle1 < 0.0:
   angle1 += pi * 2.0
if angle2 < 0.0:
   angle2 += pi * 2.0
if abs(angle1 - angle2) < pi / 2.0:
   # the robots are facing the same way
   # resolve the collision
else:
   # the robots are facing each other
   # resolve the collision
```

When two colliding robots are facing the same way, one of them will be in front of the other. In order to determine which robot is in front, Algorithm 2.2 is used. The basic idea is that the perpendicular distance from the robot in front to the other robot's coronal plane would be less than the perpendicular distance from the other robot to the robot in front's coronal plane. To get the perpendicular distance from robot A to the robot B's coronal plane, we simply find robot A's coordinate in robot B's robot relative coordinate system.

Algorithm 2.2 Robot's position relative to the other robot

```
# distance from robot B to robot A's coronal plane
p1 = [b.x - a.x, b.y - a.y]
theta1 = radians(90) - runswift_angle(a.angle)
dist_from_line1 = p1[0] * sin(theta1) + p1[1] * cos(theta1)
# distance from robot A to robot B's coronal plane
p2 = [a.x - b.x, a.y - b.y]
theta2 = radians(90) - runswift_angle(b.angle)
dist_from_line2 = p2[0] * sin(theta2) + p2[1] * cos(theta2)
if dist_from_line1 > dist_from_line2:
   # This robot is behind another robot
else:
   # This robot is in front of another robot
```

Algorithm 2.3 The visibility of an object

```
x_relative = x - self.x
y_relative = y - self.y
dist = sqrt(x_relative^2 + y_relative^2) * SCALE
if dist >= self.maxCameraDist:
   return 0
head_angle   = normalise_angle(self.angle + self.yaw)
object_angle = atan2(y_relative, x_relative)
if abs(object_angle - head_angle) >= CAMERA_ANGLE:
   return 0
return (object_angle - (head_angle - CAMERA_ANGLE)) / (2 * CAMERA_ANGLE)
```

2.2.5 Vision Model and Sonar Sensor

During the development of the simulator, it was discovered that the behaviour code was unable to play a game properly without taking into account the objects it saw through the vision system. A simple vision model was then developed to solve this problem.

In the simulator, the robot's virtual camera is mounted on its head. The robot can move the head's pitch and yaw to move the camera around.

The camera of the robot has a viewing angle of 60 degrees. The visibility of an object is dependant on its angle relative to the robot's camera and its distance from the robot. An object is not visible if it is further than maxCameraDist away from the robot. maxCameraDist is dependant on the value of robot's pitch. The formula to update the maxCameraDist is:

$$maxCameraDist = (pitch - MINPITCH) * 3000/(MAXPITCH - MINPITCH) + 1000$$

The function to determine an object's visibility is shown in Algorithm 2.3. It returns the object's x coordinate on the robot's virtual camera.

A simple sonar sensor was also implemented in the simulator. The method to

Algorithm 2.4 Updating sonar values

```
p = [b.x - a.x, b.y - a.y]
theta = radians(90) - runswift_angle(a.angle)
y_relative = (p[0] * sin(theta) + p[1] * cos(theta)) * SCALE
if y_relative > 0.0 and y_relative < 1000.0:
   x_relative = (p[0] * cos(theta) - p[1] * sin(theta)) * SCALE
   if abs(x_relative) < 100.0:
      if x_relative >= 0.0:
         right_sonar = min(y_relative/1000.0, right_sonar)
      else:
         left_sonar = min(y_relative/1000.0, left_sonar)
```

update the sonar values is shown in Algorithm 2.4.

2.2.6 Reading the behaviour code

The simulator would create the behaviour skill for each agent, and to get the action command request, the simulator would run the tick() function of each skill. The simulator then needs to decode the action command request to read the actual request of the behaviour skill.

The problem here is that the action command request is generated through robot.py which is a file generated by SWIG. Simplified Wrapper and Interface Generator (SWIG) is a language wrapper that allows the programs written in C / C++ to connect with a variety of high-level programming languages such as Python. The use of SWIG in the rUNSWift code base enables the C++ code to communicate with the Python. However, the files generated by SWIG generally contain the functions already wrapped and not meant to be changed.

The file robot.py is dependant on the _robot module which is another module generated by SWIG. Due to rUNSWIft's SWIG configuration, the _robot module will be present on the real robot. However, running the simulator on another machine would make the robot.py file complain about the missing _robot module.

The only solution to this problem would be to provide robot.py with _robot module. One of the places where the Python interpreter would look for a module is the .so file of the module. In our case, we can provide the _robot module via the _robot.so file. There are several ways to get the _robot module. We can generate a file of the module ourselves with the make utility, or the easier way would be to use a copy of the module who is already compiled in libsoccer.so. Linking the libsoccer.so with _robot.so would be enough to make the robot.py work. The simulator can then read the action command requests from the behaviour skills.

2.3 Results

2.3.1 The Design of the Simulator

The framework of the simulator was quite flexible allowing various parts of the code to be changed as the development progressed. The end result is a reasonably fast simulator capable of running multiple agents.

2.3.2 The Robot's Movement and Vision

The methods that were discussed in the previous section were able to generate realistic movement for the robot. The only problem is the simulator only models one type of walk. When the walk is changed, most of the walking code also needs to be modified.

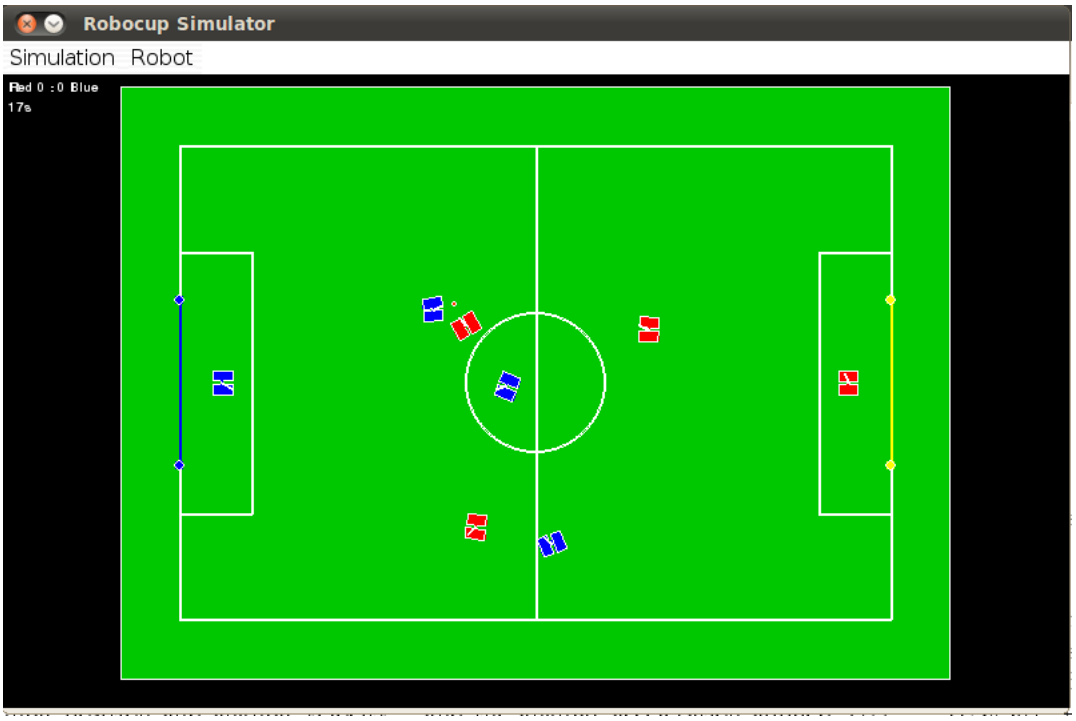

Figure 2.6: 4 vs 4 game on the simulator

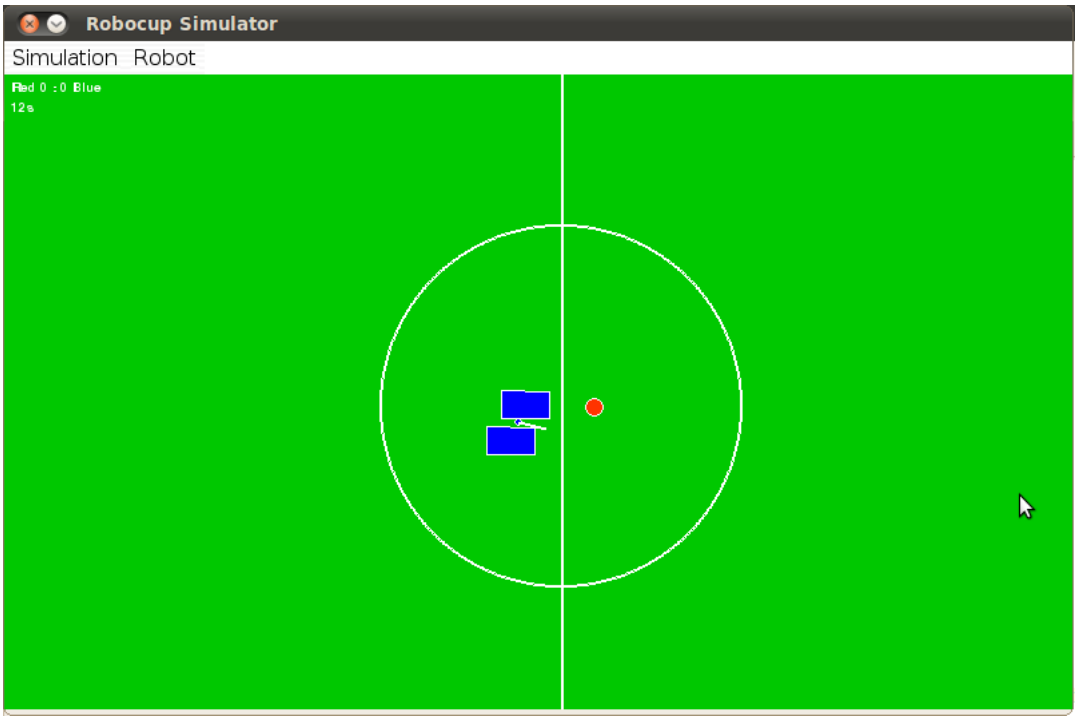

Figure 2.7: The agent performing the kick

The vision system runs reasonably well. It is very simple and doesn't require too much computing power, but good enough for the purpose of this thesis.

2.4 Evaluation and Discussion

The resulting simulator was able to run the behaviour code and simulate the robot's movement. However, the ability of the simulator to run hugely depends on the structure of the rUNSWift to stay consistent. When the important components such as the blackboard are changed, there is high chance that the simulator would also need to be modified.

With the always changing code base, a simulator that is specifically developed

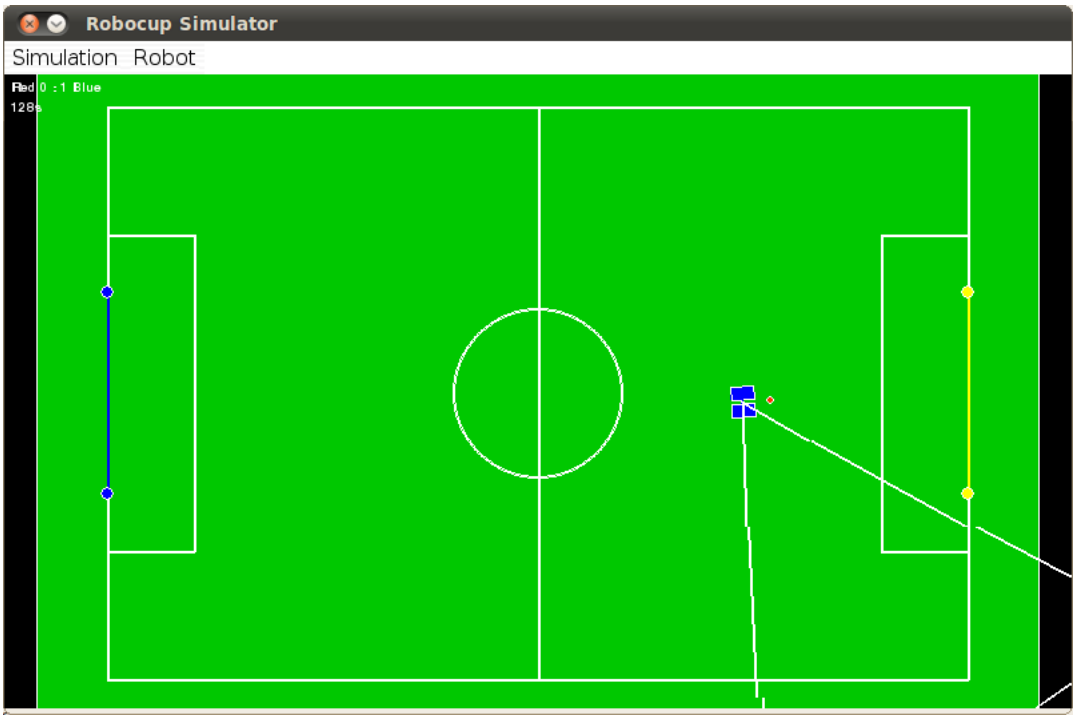

Figure 2.8: The agent looking for the ball. The simulator's Field of View is turned on.

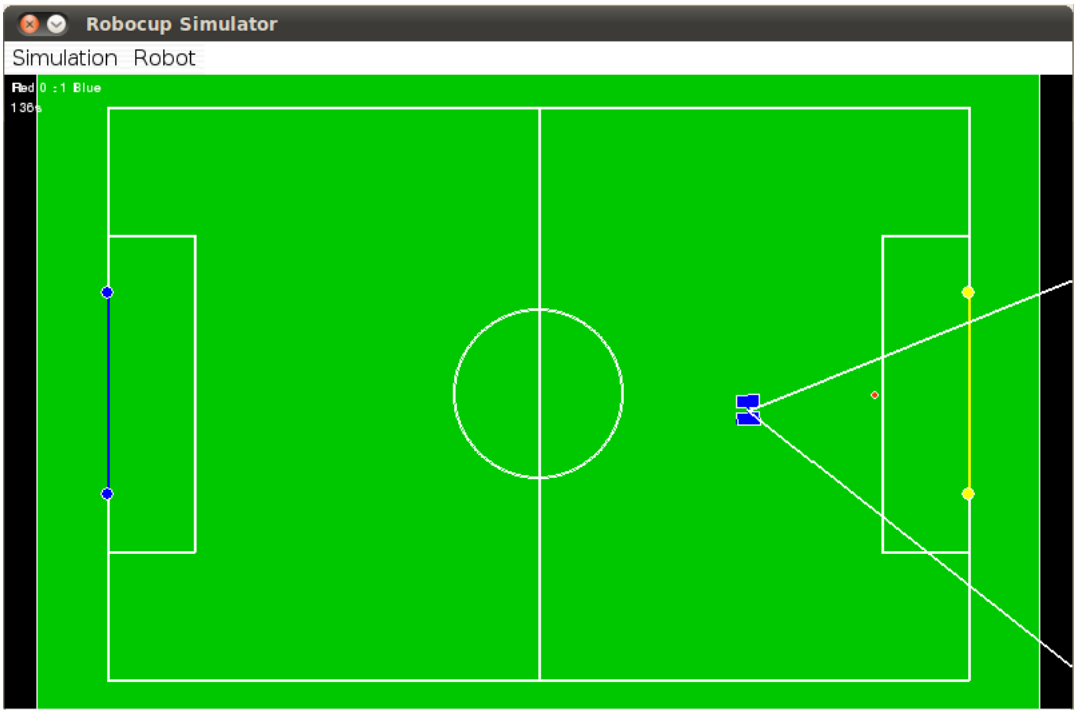

Figure 2.9: The agent finished kicking the ball. The simulator's Field of View is turned on.

to run rUNSWift behaviour code can only provide so much help. Towards the start of the 2011 competition, the behaviour code was ported to C++ to help speed up the overall system. The simulator was not usable at this time as it can only run the Python code.

In summary, the simulator does provide some help with the development of the behaviour skills, but it's limited and dependant on too many factors.

2.5 Future Work

An interesting idea for the future would be a very generic simulator that uses faster language, like C or C++. Instead of modelling one specific walk, the simulator should let the users specify the parameters of the walk. The simulator should also give the options to the users to run the behaviour code in a few languages. A very generic simulator like that would be more useful and can be used over a very longer period of time.

Chapter 3

Reinforcement Learning

3.1 Background

One of the more interesting applications of the simulator would be to do a series of reinforcement learning. There are a few behaviour skills that can be learned with reinforcement learning. One of the skills that would be challenging to learn is approaching the ball. In approaching the ball, the robot would pick a point close to ball, set it as the target coordinate and walk to it. The method to pick a point would change depending on the game strategy and the scenario. In this thesis, we would simplify the case by always picking a fixed point as the target.

Q-Learning

In reinforcement learning, the learning system must discover by trial and error which actions are most valuable in particular states. Q-learning is one of the

reinforcement learning techniques that works by learning a value function that gives the expected reward of taking a given action in a given state and following a fixed policy thereafter.

In Q-Learning, a typical problem model consists of an agent, a set of states S, and a set of actions per state A. The agent would earn a reward by doing an action[SB98]. Performing the action might cause the agent to move from one state to another state. The agent's goal is to maximise its total reward.

Each time the agent is given a reward, the value function is updated with the following formula:

$$Q(s_t, a_t) = Q(s_t, a_t) + alpha_t(s_t, a_t) * [R(s_{t+1}) + Gamma * maxQ(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where:

$alpha_t(s_t, a_t)$ is the learning rate which determines how much the new reward would override the old reward

$R(s_{t+1})$ is the reward earned from performing action a_t

$Gamma$ is the discount factor which determines the importance of future rewards for the agent

3.2 Methodology

In this experiment, the state space is the location and heading of the robot, while the action space is all the possible actions that the agent could perform.

The state and action spaces are continuous, therefore they need to be discretized first.

The state space consists of the x and y coordinate of the robot and its heading. A 1 meter x 1 meter section of the field was chosen as the state space, it was discretized by dividing it into smaller boxes of size 10 mm x 10 mm. The heading of robot is discretized into 8 angles: 0, PI / 4, PI / 2, 3 / 4 * PI, PI, 5 / 4 * PI, 1.5 * PI, 7 / 4 * PI.

The action space consists of all the actions the agent can do. It is discretized into 9 actions:

Action[0] Forward= -80 Left = 0 Turn = 0

Action[1] Forward= -40 Left = 0 Turn = 0

Action[2] Forward= 0 Left = 0 Turn = 0

Action[3] Forward= 40 Left = 0 Turn = 0

Action[4] Forward= 80 Left = 0 Turn = 0

Action[5] Forward= 0 Left = -40 Turn = 0

Action[6] Forward= 0 Left = 40 Turn = 0

Action[7] Forward= 0 Left = 0 Turn = -30

Action[8] Forward= 0 Left = 0 Turn = 30

The first method that was explored was a simple Q Learning with the coordinate (1000, 1000) as the target. The agent would start anywhere inside the 1 meter x 1 meter area.

Algorithm 3.1 Eligibility trace for Q Learning

```
Q[prevState] = Q[prevState] + LEARNING_RATE * (r + max_Q - Q[prevState])
T_STATE.append(prevState)
if len(T_STATE) > 20:
   T_STATE.pop(0)
T_ACTION.append(action)
if len(self.T_ACTION) > 20:
   T_ACTION.pop(0)
T_R.append(r)
if len(T_R) > 20:
   T_R.pop(0)
for i in range(len(T_STATE)-2, -1, -1):
   pState = T_STATE[i] + T_ACTION[i]
   curState = T_STATE[i+1]
   max_Q = -1e308
   # Actions are indexed from 0 to 8
   for j in range(0,9):
      Q_value = Q[curState+j]
      if Q_value > max_Q:
         max_Q = Q_value
   Q[pState] = Q[pState] + LEARNING_RATE * (T_R[i] + max_Q - Q[pState])
```

The second method was Q Learning with eligibility traces with the coordinate (1000, 1000) as the target. Eligibility trace is a temporary record of an event. Using eligibility trace in Q Learning involves updating not only the Q value of the current state, but also those of previous states. In this experiment the algorithm would update the Q values of 20 last states. The algorithm to update the previous states is presented in Algorithm 3.1.

The last method used was the uniform case-based Q Learning with eligibility traces where the target is set to the coordinate (1000, 1000) and heading = 0 radian. Each case in the case-based Q Learning represents a point in the state space instead of an interval [SSR96]. Therefore in our experiment, instead of dividing the state space into small boxes, it should be discretized into equally spaced points. When updating the current state, the algorithm would look

at the points closest to the current state and update them instead of simply assuming that the current state lies inside an interval and update the Q value for that interval.

3.3 Results

The first method with just a simple Q Learning was unable to yield a good result as it took a really long time to converge. Several combinations of learning rate and discount factor were tried, but none of them performed well.

The second method with eligibility traces worked fine with the x and y coordinate as its target. The learning rate used was 0.3, while the discount factor was not used (set to 1.0). However, when the second method was used to learn the value function with x, y, and heading as its target, the value function was unable to converge in reasonable time.

The last method, uniform case-based Q Learning with eligibility traces, was able to learn the value function when x, y, and heading are set as its target. The learning rate used was 0.1, while the discount factor was not used (set to 1.0). The value function was learned in acceptable time.

3.4 Evaluation and Discussion

The biggest challenge in this experiment is the continuous state and action space, and the algorithm's inability to make the Q values converge properly.

Since reinforcement learning only works with a set of predefined states and actions, the continuous state and action space need to discretized. However

discretizing the state and action space does not solve the problem straight away. This is because a simple method of discretizing does not accurately represent the problem. Other techniques like eligibility traces and uniform case-based need to be utilised for the Q learning to yield a good result.

Due to the nature of the problem, the Q values never converge properly. If the learner algorithm is run forever, then the Q values would just keep changing. After it is left running for a while, it would change only around the approximately optimal values. The only way to check if the learner has found good Q values is by running the Q values on the agent.

3.5 Future Work

This method of reinforcement learning can be applied to learn other behaviour skills as well. Since rUNSWift's behaviour code consists of skills of different levels, some form of hierarchical reinforcement learning might be used in the future experiments. However, the reinforcement learning described here depends greatly on the simulator and the simulator, as discussed in the previous chapter, is hugely dependant on the structure of the rUNSWift code to remain consistent. As the rUNSWift code is always a work in progress, it cannot maintain the same code structure for too long. Reinforcement learning for the behaviour skills is definitely plausible. However, other means would have to be explored in the future.

Chapter 4

Behaviour Skill

4.1 Background

In the rUNSWift architecture, the behaviour skills are the highest level skills which consist of other lower level skills such as shoot skill and go to ball skill. The behaviour skills are divided into three types: striker skill, supporter skill, and the goalie skill. Each of these skills has its own role and responsibility within the team.

One particular behaviour that was developed as part of this thesis is the supporter behaviour. The supporter is the robot that is neither staying near the goal line to protect the goal nor attacking the ball. Once the robot gets too close to the ball, it will usually turn into a striker. As its name suggests, the primary responsibility of the supporter is to provide support to the goalkeeper and the striker. This chapter would describe this behaviour and its development process. The simulator was used extensively in the development of this behaviour.

4.2 Methodology

The behaviour of the supporter would be written and then tested firstly on the simulator. This speeds up the development process as there is no need to upload the code to the robot everytime something changes. The developer also does not need to run around the field minding the robot, or testing it with different scenarios. After the behaviour code is working on the simulator, it is then tested on a real Nao robot.

Since there are two supporter robots, the supporter skill is divided into two types: the midfielder skill and the defender skill. Both of these skills will always try to face in the direction of the ball to help the striker and the goalie with team ball data. They will also never step back into the goal box as they will be penalised for doing so. The more specific characteristics of both skills are described below.

Midfielder Skill

The midfielder is in charge of providing support to the striker when attacking the opposition goal. When in front of the ball, the midfielder will try to avoid the direct path from the ball to the opposition goal [Kur11]. Once it is out of the shooting path, it will position itself depending on the position of the ball. At the start of the game and after the midfielder has not seen the ball for a long time, it will walk to its predefined position.

Defender Skill

The primary duty of the defender is to provide support to the goalie. When in front of the ball, the defender will try to avoid the direct path from the ball

to the opposition goal. When behind the ball, the defender will try to block the direct path from the ball to the its team's goal. At the start of the game and after it has not seen the ball for a long time, it will walk to its predefined position.

4.3 Results

As expected, our development cycle (writing the code - testing on the simulator - testing on real robot) speeds up the whole development process. In most cases, the behaviour that is working well on simulator will only need minimal parameter tuning for it to work on the real robot.

Midfielder Skill

The midfielder will avoid the striker's shooting path and then position itself some distance away from the ball. Upon experimenting with its distance from the ball, it was found that using a distance that is dependant on the ball's position on the field is much better than using a static distance. The algorithm used to determine the distance from the ball is described in Algorithm 4.1.

Basically the algorithm will ask the midfielder to move to a predefined spot (2200, 0) after the ball has crossed the x = 2000 line. When the ball is between x = 1200 and x = 2000, the robot will stand parallel to the ball by staying at the same x value and 1200 mm away in the y direction. When the ball is to the left of the x = 1200 line, the midfielder's distance from the ball is a linear function of the ball's x coordinate. When the ball is at x = -3000, the midfielder will stay MAX_DIST_FROM_BALL in front of the ball, and when the ball is

Algorithm 4.1 Determining the midfielder's distance from ball

```
if (ball.x >= 2000) {
   targetX = 2200;
   targetY = 0;
   targetTheta = ball.heading;
   robot.move(targetX, targetY, targetTheta);
} else if (ball.x >= 1200) {
   targetX = ball.x;
   if (y < ball.y) {
      targetY     = ball.y - 1200;
      targetTheta = PI / 2.0;
   } else {
      targetY     = ball.y + 1200;
      targetTheta = -PI / 2.0;
   }
   robot.move(targetX, targetY, targetTheta);
} else {
   MAX_DIST_FROM_BALL = 2500;
   distFromBall = (1200.0 - ball.x) * MAX_DIST_FROM_BALL / 4200.0;
   if (ABS(x - ball.x - distFromBall) > 0) {
      targetX        = ball.x + distFromBall;
      targetY        = y;
      targetTheta    = ball.heading;
      robot.move(targetX, targetY, targetTheta);
   }
}
```

Algorithm 4.2 Determining the defender's distance from ball

```
if (ball.x <= -800.0) {
   targetX      = -1800.0;
   targetY      = 0.0;
   targetTheta  = ATAN2(ball.y, ball.x + 1750.0);
   robot.move(targetX, targetY, targetTheta);
} else {
   DISTANCE_FROM_BALL = -1250.0;
   if (ABS(x - ball.x - DISTANCE_FROM_BALL) > 0) {
      if (ball.x > -DISTANCE_FROM_BALL) {
         targetX = -1200.0;
      } else {
         targetX = ball.x + DISTANCE_FROM_BALL;
      }
      targetY = y;
      targetTheta = ATAN2(ball.y - y, ball.x - targetX);
      robot.move(targetX, targetY, targetTheta);
   }
}
```

at x = 1200, the midfielder will stay parallel to the ball.

Defender Skill

The defender would avoid the striker's shooting path and stay behind the ball blocking the opponent striker's shooting path. The algorithm used to determine defender's distance from the ball is described in Algorithm 4.2.

The algorithm states that the defender should move to a predefined spot (-1800, 0) once the x coordinate of the ball is less than -800. This means when the ball gets really close to the goal, it will use a tighter defense position. When the x coordinate of the ball gets higher than 1250, the robot should not move up front with the ball and instead stop at the x = -1200 line. This means the defender will never move too far up front.

Algorithm 4.3 The walking code for supporter skill

```
targetX     = targetABS[0]
targetY     = targetABS[1]
targetTheta = targetABS[2]
# Decision tree
if self.arrived:
   body = actioncommand.walk()
else:
   if abs(ballRR.heading()) > math.radians(15):
      body = actioncommand.walk(turn=ballRR.heading())
   else:
      y_diff = targetY - self.y
      x_diff = targetX - self.x
      theta  = math.radians(90) - self.theta
      fwd    = x_diff * math.sin(theta) + y_diff * math.cos(theta)
      lft    = x_diff * math.cos(theta) - y_diff * math.sin(theta)
      body = actioncommand.walk(fwd, -lft, ballRR.heading())
```

These two types of supporter skill are able to position themselves quitely quickly. The only issues occur when they are not localised properly as they would spend a lot of time spinning on the spot trying to locate the ball.

Supporter's Walking Code

The walking code that was used in the supporter skill is presented in Algorithm 4.3.

The algorithm basically looks at the forward and left movement, and makes sure that the sum of the two vectors would sum up to the target coordinate. A quick glance at the algorithm suggests that it would work fine. However, the problem with this algorithm is that it assumed that the robot would walk forward as fast as it walks sideways. In reality the robot walks forward and sideways at different pace, meaning the addition of the two vectors would not result in the target coordinate.

The solution implemented for this problem is configuring the robot to walk forward first until it has crossed a certain threshold, then Algorithm 4.3 would be executed. This way the different pace between the forward and sideways movement would be small and negligible.

4.4 Future Work

The behaviour skill developed in thesis generally performs quite well in the competition. Both the striker and the supporter skills would position themselves in strategic spots around the field to help the team attack and defend. However these behaviour skills are too dependant on the developer's idea of the game. An interesting idea for the future would be to do some machine learning on these skills. The learned skills might show better strategies that have never been considered before.

Chapter 5

Conclusions

5.1 Overview

This thesis report has documented the rUNSWift 2D game simulator and its development process. The simulator is able to process the Python behaviour code of 2011 rUNSWift team, and run it as if it is running on a real robot. It was developed with the help of PyBox2D physics engine.

This report also covers the application of the simulator in machine learning the go to ball skill and developing the robot's supporter behaviour. The Q Learning was used to learn the go to ball skill. The eligibility traces and uniform case-based approach were implemented to get a better result. The supporter skill is divided into two types: the midfielder skill and the defender skill. Both of these behaviour skills are able to position the robot strategically on the field.

5.2 Summary

The simulator decreases the time it takes to develop the behaviour code. It is able to run the game faster than real life, and enables me to machine learn the go to ball skill. Unfortunately, the simulator is hugely dependant on the structure of the rUNSWift code to stay consistent. As the rUNSWift code will always be a work in progress, the components of the code structure will be changing all the time. When such changes are made, there is a real possibility that the simulator will have to be modified as well. In summary, the simulator has helped the development of the behaviour code to some extent, but it is difficult to maintain.

The Q Learning provides an insight on how the reinforcement learning techniques can be utilised to learn some of the behaviour skills. This is useful as it shares the same vision with the Robocup competition which is to create completely autonomous robot soccer player.

Chapter 6

References

[Bea10] Joschka Boedecker et al. *Simspark User's Manual*, version 1.2. 2010.

[Cat09] Erin Catto. *Box2D v2.0.1 User Manual*, 2009.

[Kur11] Jimmy Kurniawan. *Multi-Modal Machine-Learned Robot Detection for RoboCup SPL.* 2011.

[SB98] Richard Sutton and Andrew S. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[RHH+10] Adrian Ratter, Bernhard Hengst, Brad Hall, Brock White, Benjamin Vance, David Claridge, Hung Nguyen, Jayen Ashar, Stuart Robinson, and Yanjin Zhu. *rUNSWift Team Report 2010.* 2010.

[SSR96] Juan Carlos Santamaria, Richard S. Sutton, Ashwin Ram. *Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces.* 1996.

[Ste03] Timo Steffens et al. *Users Manual RoboCup Soccer Server*, 2003.

[Wes07] Eric R Westervelt. *Feedback control of dynamic bipedal robot locomotion*, volume 1. CRC Press, Boca Raton, 2007.

[Whi11] Brock White. *Humanoid Omni-directional Locomotion.* 2011.