A SOFTWARE ENVIRONMENT FOR BEHAVIOR-BASED MOBILE ROBOT CONTROL

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ONUR BEKMEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2006

Approval of the Graduate School of Natural and Applied Science

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Aydın ERSAK
Supervisor

Examining Committee Members

Prof. Dr. Aydan ERKMEN	(METU, EEE)	_____
Prof. Dr. Aydın ERSAK	(METU, EEE)	_____
Prof. Dr. İsmet ERKMEN	(METU, EEE)	_____
Assoc. Prof. Dr. A. Aydın ALATAN	(METU, EEE)	_____
Dr. Murat AKGÜL	(TÜBİTAK-UEKAE)	_____

PLAGIARISM

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Onur, BEKMEN

Signature :

ABSTRACT

A SOFTWARE ENVIRONMENT FOR
BEHAVIOR-BASED
MOBILE ROBOT CONTROL

Bekmen, Onur
M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Aydın ERSAK

December 2006, 128 pages

Robotic science can be defined as a modern multi-disciplinary branch of science, which hosts many technological elements with a huge theoretic base. From electrical and electronics engineering point of view, construction of intelligent agents that produce and/or collects information by interacting the surrounding environment and that can achieve some goal via learning, is investigated in robotic science. In this scope, behavior-based robotic control has emerged in recent years, which can be defined as a hierarchically higher control mechanism over classical control theory and applications.

In this thesis, software which is capable of producing behavior-based control over mobile robots is constructed. Research encapsulates an investigation on behavior-based robotic concept by comparison of different approaches. Although there are numerous commercial and freeware software products for robotics, the number of open source, detail documented software on behavior-based control concept together with easy usage is limited. Aimed to fulfill a necessity in this field, an open source software environment is implemented in which different algorithms

and applications can be developed. In order to evaluate the effectiveness and the capabilities of the implemented software, a fully detailed simulation is conducted. This simulation covers multi-behavior coordination concept for a differential drive mobile robot navigating in a collision free path through a target point which is detected by sensors, in an unstructured environment, that robot has no priori information about, in which static and moving obstacles exists. Coordination is accomplished by artificial neural network with back-propagation training algorithm. Behaviors are constructed using fuzzy control concept. Mobile robot has no information about sizes, number of static and/or dynamic obstacles. All the information is gathered by its simulated sensors (proximity, range, vision sensors). Yielded results are given in detail.

Keywords: Behavior-based robotics, behavior coordination, arbitration, command fusion, neural network, fuzzy control, navigation, obstacle avoidance.

ÖZ

MOBİL ROBOTLAR İÇİN
DAVRANIŞA DAYALI KONTROL
YAZILIM ORTAMI

Bekmen, Onur
Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Aydın Ersak

Aralık 2006, 128 sayfa

Robotik bilimi, geniş bir teori tabanıyla teknolojik pek çok öğeyi içerisinde harmanlayan, çoklu disiplinli modern bir bilim dalı olarak tanımlanabilmektedir. Robotik biliminde elektrik-elektronik mühendisliği bakımından ele alınan ana konu, akıllı, görevleri öğrenerek yerine getirebilen, içerisinde bulunduğu ortamla etkileşerek bilgi yaratan elemanların geliştirilebilmesidir. Bu kapsamda, klasik kontrol teorisi sonrası, bir üst seviye kontrol mekanizması olarak tanımlanabilecek, yaşayan organizma ve organizma gruplarından esinlenilerek temelleri atılan, davranış tabanlı kontrol konsepti son yıllarda robotik camiasını meşgul etmektedir.

Tez çalışmasının kapsamında; davranış tabanlı kontrol kavramının incelenmesi, farklı yaklaşımların karşılaştırılması, bilgi teorisi ve dağıtık kontrol kavramları bakımından incelenmeye uygun bulunan uygulamaların geliştirilebileceği açık kaynak bir yazılımın oluşturulması bulunmaktadır. Algoritma geliştirme ve robot kontrolü için geliştirilmiş bir çok yazılım bulunmasına rağmen davranış tabanlı robot kontrolü için kolay kullanıma sahip, iyi dokümante edilmiş, açık kaynak yazılımların sayısı oldukça azdır. Bu eksiklik tespiti doğrultusunda çalışmalar yürütülmüştür. Bu çalışmada, farklı mobil robotik uygulamalarının geliştirilebileceği, davranış tabanlı kontrol kavramına uygun programlamanın yürütülebileceği bir yazılım ortamı oluşturulmuştur. Oluşturulan yazılım ortamının

kabiliyetlerinin sergilenip, etkinliğinin değerlendirilebileceği, detaylı bir sümülasyon yürütülmüştür. Yürütülen simülasyon, bulunduğu ortam hakkında her hangibir ön bilgiye sahip olmayan, iki sürücü tekerlekli hareketli robotun hedef noktaya engel nesnelere çarpmadan, çoklu davranış koorinasyonu yürüterek varmasını amaçlamaktadır. Koordinasyon amaçlı yapay sinir ağları kullanılmakta, davranış modülleri ise bulanık kontrol içermektedir. Hedef nokta belirlenen kriterler doğrultusunda robota sabitlenmiş simüle edilen sensörler aracılığıyla tespit edilmektedir. Ortamda hareketli ve sabit engel nesneler de bulunmaktadır. Elde edilen sonuçlar detaylı şekilde tez raporunda aktarılmaktadır.

Anahtar Kelimeler: Davranış tabanlı robotik, davranış koordinasyonu, yapay sinir ağları, bulanık kontrol, yön bulma, engel aşma.

To My Mother

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to my supervisor Prof. Dr. Aydın Ersak for his guidance, advice, criticism, and endless patience throughout the research.

I would like to thank my colleagues from TÜBİTAK-UEKAE/İLTAREN for their encouragement and support.

Finally, I would like to express my appreciation and gratitude to my family for their endless support and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	:	Artificial Intelligence
COG	:	Center of Gravity
CDB	:	Context Dependent Blurring
FOV	:	Field of View
METU	:	Middle East Technical University
MIT	:	Massachusetts Institute of Technology
MOM	:	Mean of Maximum
NN	:	Neural Network
ODE	:	Open Dynamics Engine
OpenCV	:	Open Source Computer Vision Library
OpenGL	:	Open Graphics Library
2WDD	:	Two Wheel Differential Drive

CHAPTER 1

INTRODUCTION

A “robot” is an electro-mechanical device that can perform autonomous or preprogrammed tasks. A robot may act under the direct control of a human or autonomously under the control of a programmed computer. Robots may be used to perform tasks that are too dangerous or difficult for humans to implement directly (e.g. nuclear waste clean up) or may be used to automate repetitive tasks that can be performed with more precision by a robot than by the employment of a human. They are also useful in environments which are unpleasant or dangerous for humans to work in, for example bomb disposal, work in space or underwater, in mining, and for the cleaning of toxic waste.

The word "robot" is also used in a general sense to mean any machine which mimics the actions of a human, in the physical sense or in the mental sense. It comes from the Czech and Slovak word *robota*, labour or work. The word robot first appeared in Karel Čapek's science fiction play R.U.R. (Rossum's Universal Robots) in 1921, and was probably invented by the author's brother, painter Josef Čapek.

“Robotics” is the science and technology of robots, their design, manufacture, and application. Robotics requires a working knowledge of electronics, mechanics, and software and a person working in the field has become known as a robotics scientist. The word robotics was first used in print by Isaac Asimov, in his science fiction short story "Runaround" (1941).

Many methods for robot control have been developed which can generally be grouped into two categories: deliberative and reactive.

In deliberative approach, global planning method is used in a completely known environment. These methods build the paths for reaching the target without any collision. A global optimum solution can be achieved in this approach. However, this scheme has well-known drawbacks. Exact model of the world is needed which is very difficult and modifications in this environment after modeling cannot be handled.

In reactive approach model of the world is not needed, actions are determined according to information gathered from sensors. The robot has to react to its sensor data by a set of stimuli-response mechanism. The drawback of these systems is limited and uncertain sensor data because of the limited range, poor observation conditions, and environmental effects.

The fuzzy logic systems are inspired from human reasoning, which is based on perception. Fuzzy logic provides a methodology for representing human expert knowledge and perception-based actions without needing analytical model of the system. Neuro-fuzzy systems add the advantages of fuzzy reasoning to neural networks, which learn fast without needing symbolic representation of the system.

Behavior-based systems try to model the reactive abilities of humans, animals, insects etc. to the sensed environment. In behavior-based approach, goals are achieved by subdividing the overall task into smaller independent behaviors that focus on execution of specific tasks. For example, a behavior can focus on traversing from start to target place, while another behavior focuses on obstacle avoidance.

The need of onboard intelligence on a robot comes from the fact that there can not be an expert operator that guides the robot in every case. This can also result from

the physical facts, like signal propagation delay as it is in the Mars Rover case, where there is more than a eight minutes of delay between Mars and Earth, thus the control center and the robot, avoiding any possible expert control.

This thesis is focused on different properties of behavior based robotics. The literature survey showed that there is a wide range of applications in which behavior-based control is used. Several of these control schemes can be seen in [2, 13, 18, 19, 20, 23, 26, 27, 35, 36, 44, 50, 51]. A full survey investigating much architecture can also be seen in [61]. Recent years witnessed lots of new robots. These robots vary from underwater vehicles to UAV's, from brachiation-type mobile robots to humanoid robots. All of these kinds of robots need an effective solution to intelligent navigation. From this basis, the behavior-based robot control can be applied to a wide range of robot types. Some of the applications can be found in [3, 17, 21, 39, 47, 54] also covering planetary exploration robots, robot arms and humanoid robots.

There has been a lot of research about humanoid robotics. These researches also include behavior-based control of these new challengers of robotic science. The MIT Robotics Laboratory focuses on humanoid robots especially, since it seems like it is the next step in the evolution of robots. Such a research yielded Cog, [59]. Cog is a humanoid robot with two arms, a robot head, and an upper torso. Rodney Brooks, who is in charge of MIT Robotics Laboratory, tries to form a robot that learns how to behave in certain situations interacts and communicates with people. Another such work is on the imitation of human mood, on a head only robot, Kismet, [59].

Behavior-based robot control is the rising value in robotic control for its advantages that are to be presented in the following chapters. Although there are numerous software for robot control, no open source behavior-based control software is found to be released, yet. Such software should be well documented and should show the availability in extension means, such as detailed simulation

models, including various mobile robots, accurate physics, sensors, etc. In order to fill the gap in this software area, this thesis research is conducted.

In order to understand what behavior-based robotics is, a literature survey has been conducted. The various aspects of the concept is shared with the reader in the following chapters of this thesis. Advantages of behavioral control over other methods are listed in detail. A high level programming language, namely C++ is used to construct a simulation environment, with detailed visualization support using OpenGL, taking the advantage of virtual reality. In order not to limit the users of this software with the researchers only, also a user friendly interface is coded, in Visual C++. Even though the ones with a little knowledge on robotics can use the software and see the resulting outcomes visually. This can help other people to have their attention on robotics. The constructed simulation environment is given in no license as a fully open source project to encourage other national and/or international researchers.

The thesis is organized as follows: In chapter 2, the concept of robot control leading to the definition of behavior-based robotics together with the brief explanation of the necessary terminology and different approaches present in the scope of behavior based robotics, with comparisons are presented in general.

In chapter 3, the implementation details and the capabilities of the constructed software is given. In this chapter, structural properties of the software such as object oriented architecture, class hierarchy; dynamic size object sets etc. are given in detail, first. Then the implemented objects such as robots, sensors and behaviors are given. Also the user interface definitions and details are given in this chapter.
In chapter 4, the detailed simulation conducted on the software environment is given. The simulation is given as an example that can be conducted by means of the available modules in the software environment. The simulation is given to show some proof of utilization of neural network in behavior prioritization, during behavior coordination.

Finally, in chapter 5, the conclusions that has been reached on the software environment is discussed. The future work planned to expand the scope of this thesis work, possibly in a PhD. study, is also presented in this chapter.

CHAPTER 2

BEHAVIOR-BASED ROBOT CONTROL

As a design strategy, the behavior-based approach has produced intelligent systems for use in a wide variety of areas, including military applications, mining, space exploration, agriculture, factory automation, service industries, waste management, health care, disaster intervention, and the home. To understand what behavior-based robotics is, it may be helpful to explain what it is not. The behavior-based approach does not necessarily seek to produce cognition or a human-like thinking process. While these aims are admirable, they can be misleading. Blaise Pascal once pointed out the dangers inherent when any system tries to model itself. It is natural for humans to model their own intelligence. The problem is that we are not aware of the myriad internal processes that actually produce our intelligence, but rather experience the emergent phenomenon of "thought." In the mid-eighties, Rodney Brooks (1986) recognized this fundamental problem and responded with one of the first well-formulated methodologies of the behavior-based approach. His underlying assertion was that cognition is a chimera contrived by an observer who is necessarily biased by his/her own perspective on the environment. (Brooks 1991) As an entirely subjective fabrication of the observer, cognition cannot be measured or modeled scientifically. Even researchers, who did not believe the phenomenon of cognition to be entirely illusory, admitted that AI had failed to produce it. Although many hope for a future when intelligent systems will be able to model human-like behavior accurately, they insist that this high-level behavior must be allowed to emerge from layers of control built from the bottom up. While some skeptics argue that a strict behavioral approach could never scale up to human modes of intelligence, others argued that the bottom-up behavioral approach is the very principle underlying all biological intelligence. (Brooks 1990)

To many, this theoretical question simply was not the issue. Instead of focusing on designing systems that could think intelligently, the emphasis had changed to creating agents that could act intelligently. From an engineering point of view, this change rejuvenated robotic design, producing physical robots that could accomplish real-world tasks without being told exactly how to do them. From a scientific point of view, researchers could now avoid high-level, armchair discussions about intelligence. Instead, intelligence could be assessed more objectively as a measurement of rational behavior on some task. Since successful completion of a task was now the goal, researchers no longer focused on designing elaborate processing systems and instead tried to make the coupling between perception and action as direct as possible. This aim remains the distinguishing characteristic of behavior-based robotics.

The sub-sections which follow explain the roots of behavior based robotics, how it rose as a counter to the symbolic, deliberative approach of classical AI and how it has come to be a standard approach for developing autonomous robots.

2.1 HISTORY

While behavior-based robotics is a relatively new field as academic fields go, it is possible to find historical predecessors. Ronald Arkin looks all the way back to 1947, when cybernetics used control theory, information science and biology to seek principles common to biological life and machine intelligence. It is generally agreed that W. Grey Walter's Tortoise, a small robot made from vacuum tubes, was the first behavior-based robot. It had no high-level knowledge and could not translate its actions into symbolic meaning. However, it could effectively exhibit certain behaviors such as backing away from strong light and heading toward weak light. It did not model human intelligence or "cognition" of any kind; rather, it provided reactive response without reliance on representation. The complexity of the action produced lay not in the design but in the behavior that arose through interaction with a chaotic world. (Arkin 1998)

In the middle of the 1980s, due to dissatisfaction with the performance of robots in dealing with the real world, a number of scientists began rethinking the general problem of organizing intelligence. Among the most important opponents to the AI approach were Rodney Brooks, Rosenschein and Kaelbling (Rosenschein, 1986) and Agre and Chapman (Agre & Chapman, 1987). They criticized the symbolic world which Traditional AI used and wanted a more reactive approach with a strong relation to the perceived world and the actions there in. They implemented these ideas using a network of simple computational elements indirectly connecting sensors to actuators in a distributed manner. There were no central models of the world explicitly represented. The model of the world used was the real one perceived by the sensors at each moment. Leading the new paradigm, Brooks proposed the "Subsumption Architecture" which was the first approach to the new field of "Behavior-Based Robotics", [2].

2.2 TRADITIONAL AI VS. BEHAVIOR-BASED APPROACH

Classical AI spent decades trying to model human-like intelligence, using knowledge-based systems that processed representation at a high, symbolic level. Symbolic representation was considered of paramount importance because it allowed agents to operate on sophisticated human concepts and report on their action at a linguistic level. As Donald Michie stated, "In AI-type learning, explain ability is all." (Michie 1988) Since the goal of early AI was to produce human-like intelligence, researchers used human-like approaches. Marvin Minsky, believed an intelligent machine should, like a human, first build a model of its environment and then explore solutions abstractly before enacting strategies in the real world. (McCarthy et al. 1955) This emphasis on symbolic representation and planning had a great effect on robotics and spurred control strategies where functionality was coded using languages and programming architectures that made conceptual sense to a human designer. Although many of the strategies developed were both elaborate and elegant, the problem was that the intelligence in these systems

belonged to the designer. The robot itself had little or no autonomy and often failed to perform if the environment changed. While classical AI viewed intelligence as the ability of a program to process internal encodings, a behavior-based approach considers intelligence to be demonstrated through "meaningful and purposeful" action in an environment. (Arkin 1999)

While many perceived the behavior-based movement to have forsaken the goal of human-like intelligence, others maintained that high-level intelligence would indeed arise once a strong, low-level foundation had been laid. Agre and Chapman argued that, in fact, human beings are actually much more reactive than we imagine ourselves to be. (Agre and Chapman 1987) The planning and cognition that we are consciously aware of represents only the tip of a cerebral iceberg comprised mostly of unconscious, reactive motor skills and implicit behavior encodings. In a sense, the behavioral approach did not abandon modeling human intelligence as much as human consciousness. One of the side effects has been that many behavior-based approaches produce systems that are anything but 'explainable.' High scientific aims aside, a main reason the behavior-based community is so intent on developing automated learning techniques is that a human designer often finds it excruciatingly tedious or impossibly difficult to orchestrate many behaviors operating in parallel. It is worse than frustrating to debug behavior that emerges from the interplay of many layers of asynchronous control. At times, a truly well-implemented, behavior-based approach will result in successful strategies the researchers themselves cannot explain or understand.

Instead of the top-down approach (also referred as horizontal architecture) of Traditional AI (Figure 2.1), Behavior-based systems use a bottom-up philosophy (also referred as vertical architecture) like that in Reactive Robotics (Figure 2.2).

The traditional artificial intelligence architecture to be used for robot control systems is the horizontal architecture in which the tasks of the control systems are broken into several subtasks based on functionality. A typical approach is to

decompose as shown in the Figure 2.1. Control systems using this architecture solve their task in several steps. First, the sensor input is used to modify the internal representation of the environment. Second, based on the internal representation planning is made. This results in a series of actions for the robot to take to reach a specified goal. Third, this series of actions is used to control the motors of the robot. This completes the cycle of the control system and it is restarted to achieve new goals.

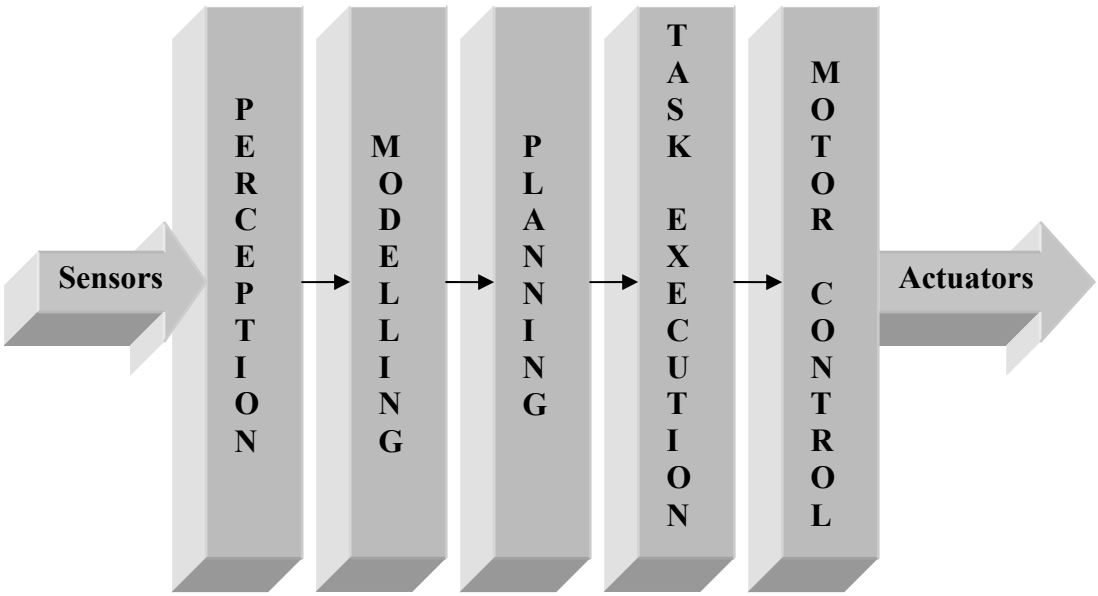

Figure 2.1 Horizontal Control Architecture (adopted from [19])

In Figure 2.2 subsumption architecture is given as an example of a vertical architecture. The new idea was that instead of decomposing the task based on functionality, the decomposition is done based on task achieving behaviors.

A behavior based control system is made up of several parallel running behaviors. Each behavior calculates a mapping from sensor inputs to motor outputs. An output integration method is also required to take actions by driving the actuators. The types of different integration methods will be discussed deeply with in the following chapters.

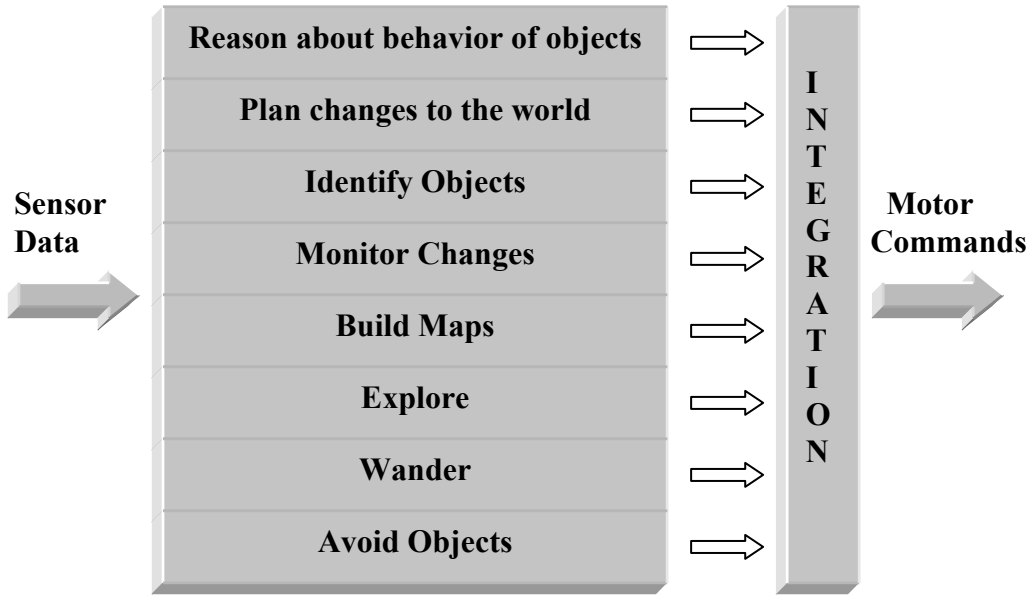

Figure 2.2 Vertical Control Architecture (adopted from [19])

Reactive systems provide rapid real-time responses using a collection of pre-programmed rules. Reactive systems are characterized by a strong response, however, as they do not have any kind of internal states, they are incapable of using internal representations to deliberate or learn new behaviors. On the other hand, Behavior-based systems can store states in a distributed representation, allowing a certain degree of high level deliberation.

The Behavior-based approach uses a set of simple parallel behaviors which react to the perceived environment proposing the response the robot must take in order to accomplish the behavior (Figure 2.2).

Behaviors select actions based on information contained in an internal representation called sensory data space, which can be divided into two distinct subspaces: observation space and state space. Observation space is the n_o-dimensional Cartesian product $O = \otimes_i^{n_o} \mathrm{O}_i$ over all available n_o sensorial dimensions $\mathbf{O} = \{\mathrm{O}_i \mid 1 \le i \le n_o\}$. State space is the n_s dimensional Cartesian

product $\mathrm{S} = \otimes_i^{n_s} S_i$ over n_s state dimensions $\mathbf{S} = \{\mathrm{S}_i \mid 1 \le i \le n_s\}$. Data space is the n_d-dimensional Cartesian product $\mathrm{D} = \otimes_i^{n_d} D_i$ over n_d data space dimensions $\mathbf{D} = \{D_i \mid 1 \le i \le n_d\}$, where $n_d = n_o + n_s$ and $\mathbf{D} = \mathbf{O} \otimes \mathbf{S}$. Motory space is the n_m-dimensional Cartesian product $\mathrm{M} = \otimes_i^{n_m} M_i$ over all available n_m motory or effectory dimensions $\mathbf{M} = \{\mathrm{M}_i \mid 1 \le i \le n_m\}$. Parameter space in the n_p-dimensional Cartesian product $\mathrm{P} = \otimes_i^{n_p} P_i$ over n_p parameter dimensions $\mathbf{P} = \{P_i \mid 1 \le i \le n_p\}$. The domains for any of the dimensions in the defined spaces may be finite or infinite, and discrete or continuous valued. Finally, event space is a finite set $\boldsymbol{\varepsilon} = \{\varepsilon_i \mid 1 \le i \le n_e\}$ of n_e discrete events.

A behavior is a mapping from data space to motory space and events (see Figure 2.3), formally denoted by $b: D(b) \mapsto M(b) \times \varepsilon(b)$, where $D(b)$ is some d_b-dimensional subspace of D, $M(b)$ is some m_b-dimensional subspace of M, and $\varepsilon(b)$ is a finite subset of ε. The set of available behaviors in any particular implementation is referred to as $B = \{b_1, ..., b_m\}$.

In behavior-based control, there are neither problems of world modeling nor real time processing. Nevertheless, another difficulty has to be solved; how to select the proper behaviors for robustness and efficiency in accomplishing goals.

Two new questions also appears, which Traditional AI doesn't take into consideration; how to adapt the architecture in order to improve its' goal achievement, nor how to adapt it when new situations appear. This simple but powerful methodology was in great contrast to Traditional AI and, from its' beginning, provided for simplicity, parallelism, perception-action mapping and real implementations. Some survey can be followed from [1], [34] and [53] on goal oriented behaviors and tradeoffs made on goal oriented behavior construction.

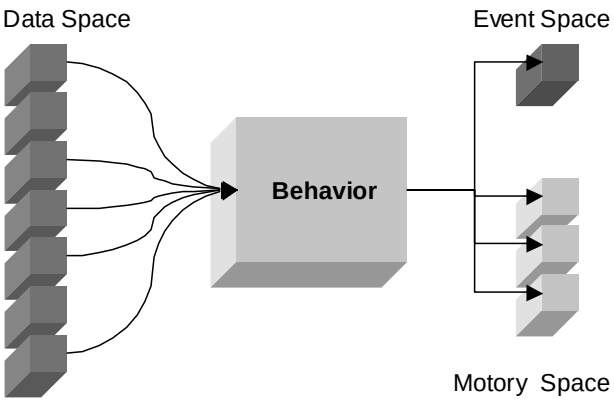

Figure 2.3 Structure of Behavior

2.3 BEHAVIOR SELECTION

Behavior-based control architecture can be organized horizontally which shows that each behavior has full access to all sensor readings and processes its own command to control the mobile robot. The final command is dependent on the priority of each behavior. There are many applications of behavior based approaches that have been presented during the last few years such as Arkin,

Kasper, Fricke, and Puttkamer, Yen and Pfluger, and Payton and Rosenblatt. The research behavior-based topics for mobile robots are still being developed by many researchers [55].

As the behavior has been defined, there comes a natural question, “How to choose among multiple behaviors?” This is the main and a challenging problem that is still being investigated. This phenomenon is referred as behavior selection, behavior fusion, arbitration or behavior scheduling. Some discussion about the behavior selection and coordination topic can be followed from [14], [30], [38] and [58].

Coordination of the several simultaneous independent behavior-producing units to

obtain an overall behavior that achieves the intended task is behavior coordination. The simplest example is the coordination of an obstacle avoidance behavior and a goal reaching behavior, to reach a target in an environment with obstacles. Today behavior coordination is still a major problem. Behavior coordination problem can be evaluated in two categories: behavior arbitration and command fusion.

2.3.1 Arbitration

The arbitration policy determines which behavior should influence the operation of the robot at each moment, and determines the task performed by the robot. Early solutions based on fixed arbitration policy. An example for this approach is the famous subsumption architecture proposed by Brooks [2], which is based on a hard-wired network of suppression and inhibition links. (Figure 2.4) This rigid organization contrasts with the requirement that an autonomous robot can be programmed to perform a variety of different tasks in a variety of different environments. In fact, Brooks' robots were usually built to perform one single task. Subsumption Architecture is a method of reducing a robot's control architecture into a set of task-achievement behaviours or competences represented as separate layers. Individual layers work on individual goals concurrently and asynchronically. All the layers have direct access to the sensory information. Layers are organised hierarchically allowing higher layers to inhibit or suppress signals from lower layers. Suppression eliminates the control signal from the lower layer and substitutes it with the one proceeding from the higher layer. When the output of the higher layer is not active, the suppression node doesn't affect the lower layer signal. On the other hand, only inhibition eliminates the signal from the lower layer without substitution. Through these mechanisms, higher-level layers can subsume lower-levels.

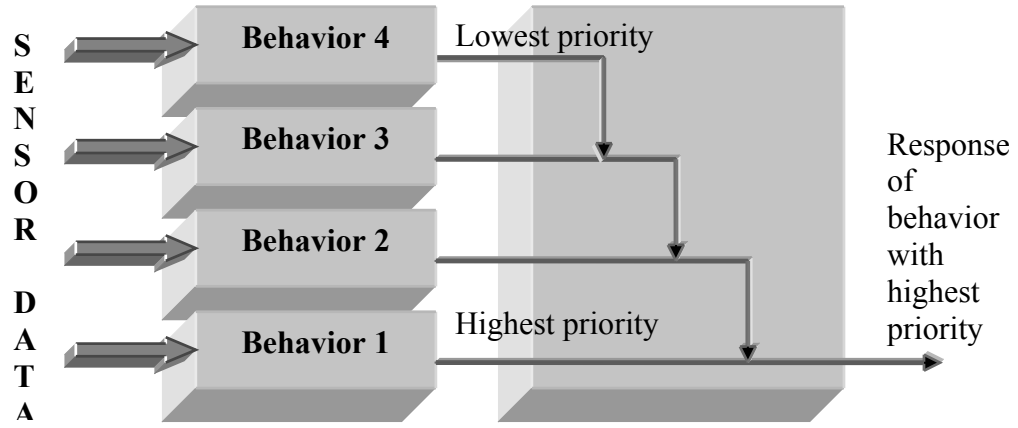

Figure 2.4 Arbitration by Suppression (adopted from [65])

One of the arbitration methods is action-selection. In this method, the activation level (priority) of each behavior is determined according to robots' goals and sensory measurements. The behavior with the highest activation level (priority) is carried out at run time. No hierarchy exists between the behaviors (Figure 2.5). Behavior coordination with voting is another arbitration method. In this architecture, behaviors vote for a predefined set of motor actions and the action receiving the highest vote is accomplished (
Figure 2.6).

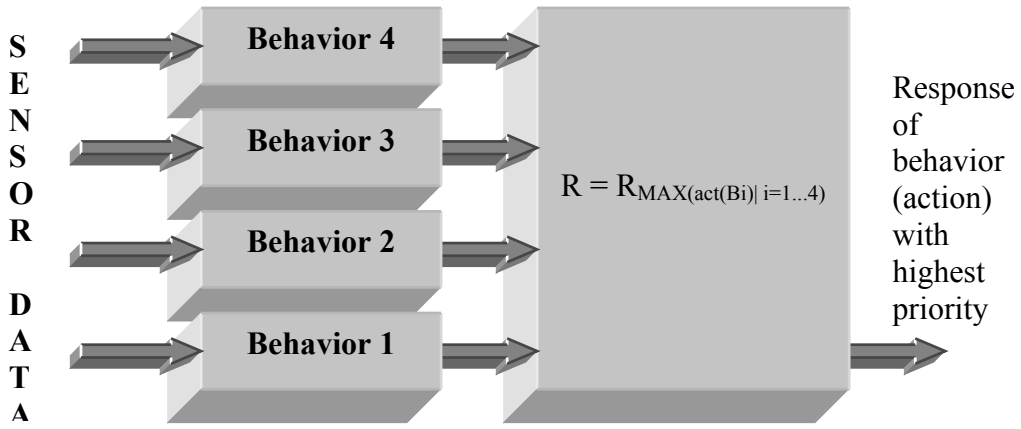

Figure 2.5 Arbitration by Action Selection (adopted from [65])

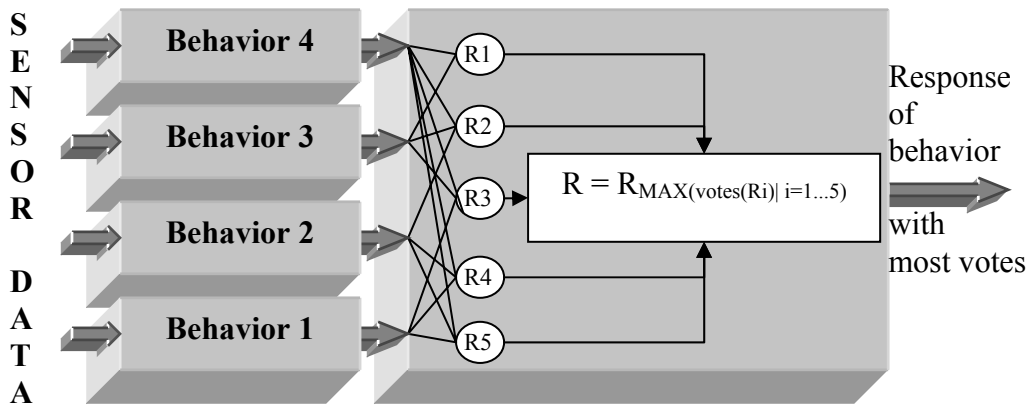

Figure 2.6 Arbitration by Voting (adopted from [65])

2.3.2 Command Fusion

In arbitration, one of the behaviors is selected and accomplished according to arbitration method. This scheme may be inadequate in situations where several criteria should be taken into account. For example, consider a robot that encounters an obstacle while following a path and arbitration policy selects the obstacle avoidance behavior. Going around the obstacle from left or right is unimportant for the obstacle avoidance behavior. However, from the point of view of the path-following behavior, one choice might be dramatically better than the other.

To overcome this problem, different behaviors are executed parallel and outputs of these behaviors are combined. The most popular approaches for this type are based on vector summation scheme (Figure 2.7): a force vector represents each command, and commands produced by different behaviors are combined by vector summation. The robot carries out the resulting action.

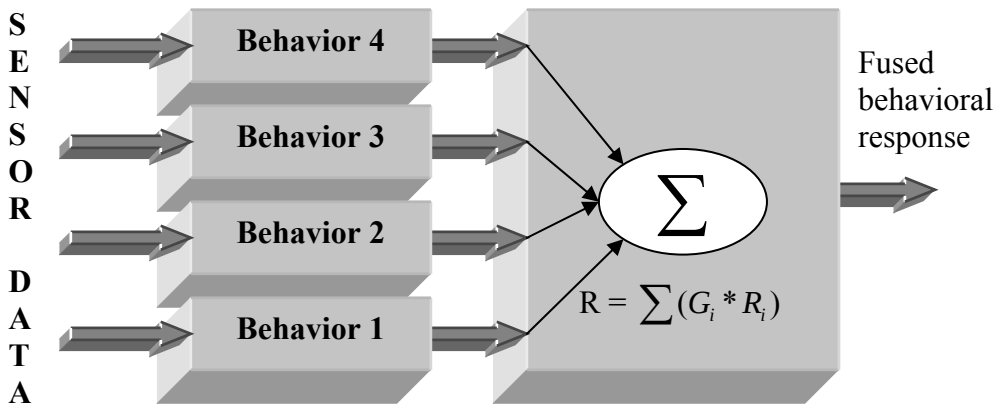

Figure 2.7 Behavior Coordination by Vector Summation

When the output of a behavior is represented by a fuzzy set, fuzzy operators can be used to combine the output of different behaviors into a collective result, and finally choose a command according to this result. Fuzzy logic offers many different operators to perform combination (min. for intersection, max. for union etc.), and many defuzzification functions (Center of Gravity, Mean of Maximum etc.) to perform decision. It is important to note that the decision taken according to the collective output can be different from the result of combining the decisions taken from the individual outputs.

Figure 2.8 graphically illustrates this point in the case of two behaviors.

Figure 2.8 shows two cases: Case 1 represents a two behavior case in which defuzzification is applied before vector summation; Case 2 represents again a two fuzzy behavior case in which defuzzification is applied as the final step. Note that two cases yield different results. This argument explains why fuzzy command fusion is fundamentally different from vector summation.

Another form of behavior combination that can be realized using fuzzy logic is obtained by using both (i) fuzzy meta-rules to express an arbitration policy, and (ii) fuzzy combination to perform command fusion. This form of combination, was initially suggested by Ruspini, and fully spelled out by Saffiotti under the name of context-dependent blending of behaviors, or CDB [66].

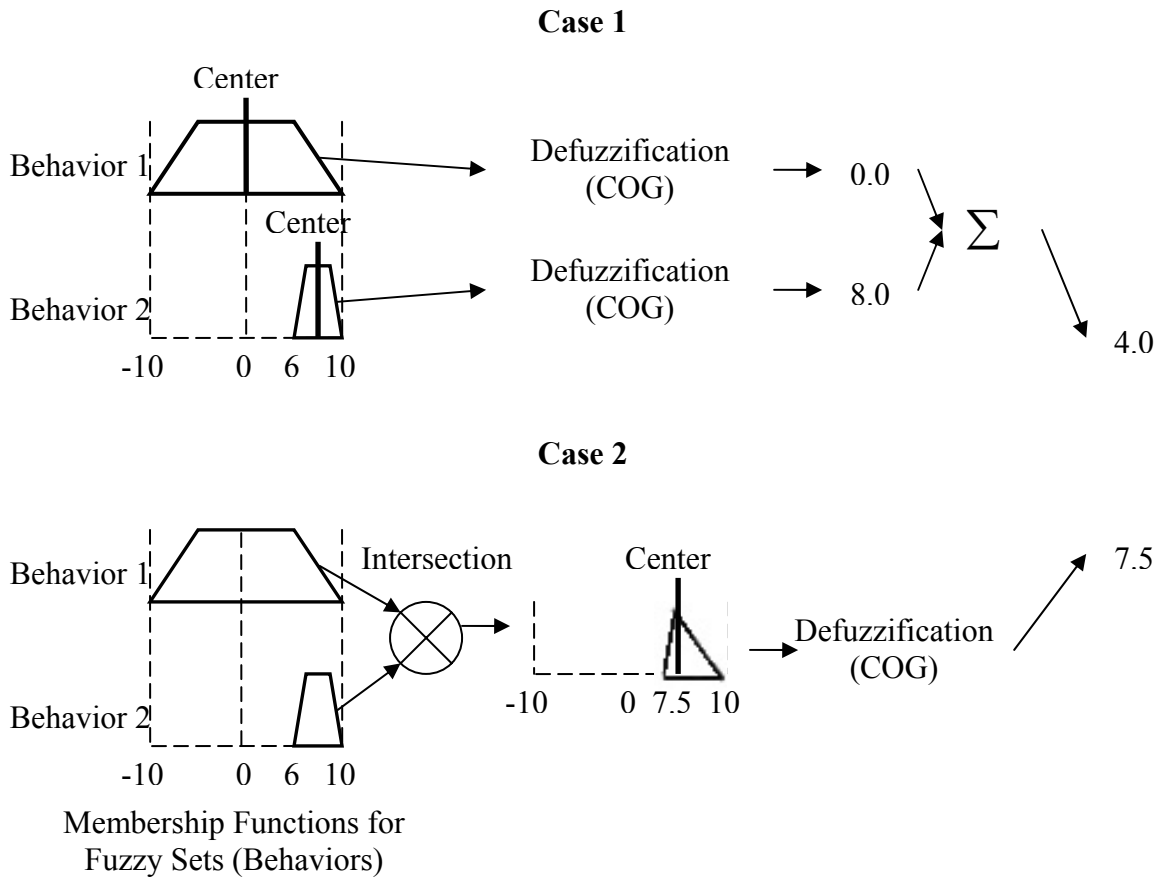

Figure 2.8 Fuzzy Behavior Coordination (adopted from [66])

CDB can be implemented in a hierarchical fuzzy controller as shown in Figure 2.9. In CDB, it is essential that the defuzzification step must be performed after the combination. Although in Figure 2.9 all the context-rules are grouped in one module, each context-rule can be put inside the corresponding behavior and this solution would be more appropriate for distributed implementations. This architecture can be iterated to implement individual behaviors, and combine them using a second layer of context-rules. Defuzzification should still be the last step.

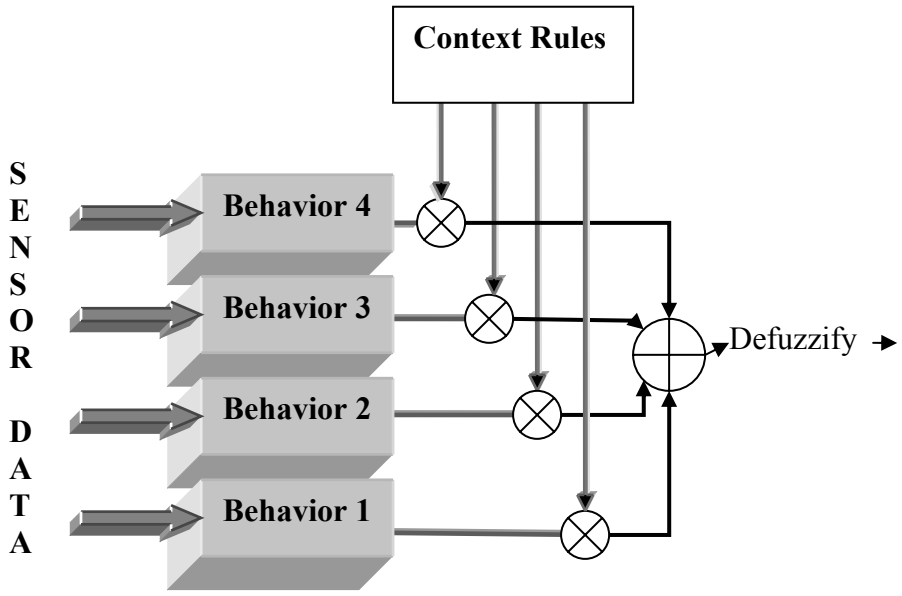

Figure 2.9 Hierarchical Fuzzy Controller (adopted from [66])

2.4 LEARNING IN BEHAVIOR-BASED CONTROL

Also inspired from living creatures, there should be a learning mechanism that continuously enhancing the reactions of the robot. Many strategies are developed in learning systems, not special to behavior-based robot control. In behavior-based robotics, many fields can be found to apply different learning strategies, including well known reinforcement learning, genetic algorithms, fuzzy learning etc. Some of the suitable research can be found in [41, 43, 48, and 49]. These references generally apply a learning mechanism into behavior selection process.

2.5 BEHAVIOR EVALUATION

Although behavior-based robot programming or behavior-based control is commonly researched field, there does not exist an accepted behavior evaluation

method. In order to evaluate whether a behavior is successful or not, goals of the robot can be investigated. If the goal is achieved than, behavior can be evaluated as success. In order to measure the performance of a behavior and find out how successful the behavior is, there should be some metrics. The literature survey showed that there does not exist, a commonly accepted metric system for the performance evaluation of behaviors. Researchers generally do their own evaluation depending on the problem statement. There are some recent researches on the behavior evaluation metric, such as [46] and [62], but these methodologies are still not commonly accepted. They still can show means to follow in the evaluation process of behaviors.

CHAPTER 3

IMPLEMENTATION

In this chapter, before giving the details of the implemented software, some other existing software on mobile robot control are given briefly in order to give some sense of comparison of the implemented software with the previous software on robotics.

3.1 PREVIOUS SOFTWARE ON ROBOT CONTROL

Since behavior based robotic is a popular research in recent years, many software on the concept have been developed. The reason of developing simulations and simulation environments is due to the fact that the construction of real hardware robots cost much more than their virtual correspondences. After the algorithms are tested and verified in simulation, they can be than be loaded on to real robot hardware.

The limitations of real life can not be simulated completely what ever the resolution of the simulation is. The real experiences that can not be obtained by simulations are gained only by real robots.

The popularity of the concept shows itself by the approximate number of simulators on robot control. There are at least 100 simulators, from experimental small program fragments to commercial products. Even universities are trying to develop and sell such products, [63].

Khepera is one of the most widely used mobile robot. It has a two wheel differential motion system. It comes with 8 ultrasonic transducers for environment sensing, by default. There are numerous accessories of Khepera including low resolution Vision Turret or High resolution Vision Module. Khepera is given in Figure 3.15 and Figure 3.16. The availability of Khepera made it one of the most paper published robot, [56].

3.1.1 Yaks

YAKS is one of the many Khepera simulator projects. It runs on Linux systems or Windows systems utilizing shells like Cygwin. It also requires gtk++ libraries to be installed prior to operation of the software. Figure 3.1 gives the screenshot of limited user interface of YAKS.

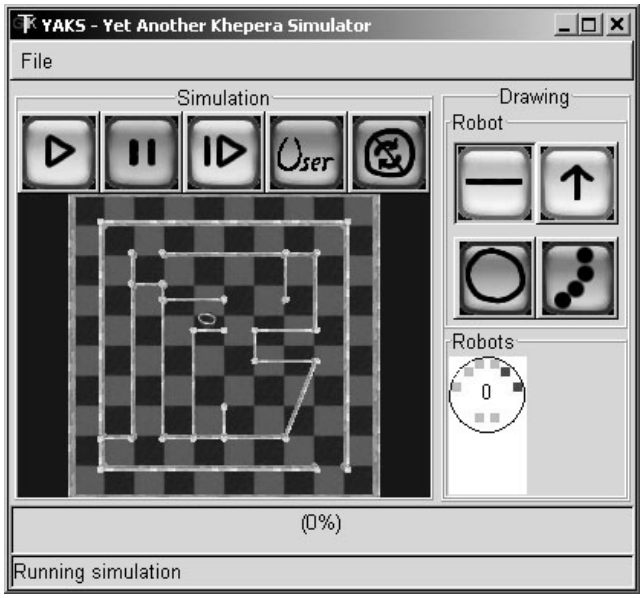

Figure 3.1 YAKS Khepera Simulator User Interface

3.1.2 Simulator Bob

SimBob is a powerful robot simulator with enhanced visual capabilities. Distributed under GNU License, SimBob gives the designer the reliability to use physics based simulations, by embedding ODE, Open Dynamics

Engine. ODE is a powerful open source physics engine that can be embedded into many applications. A screenshot from the simulation of a two wheeled mobile robot with light sensors is given in Figure 3.2.

User is able to write controllers for the simulated robots using the available sensors on the modeled robot.

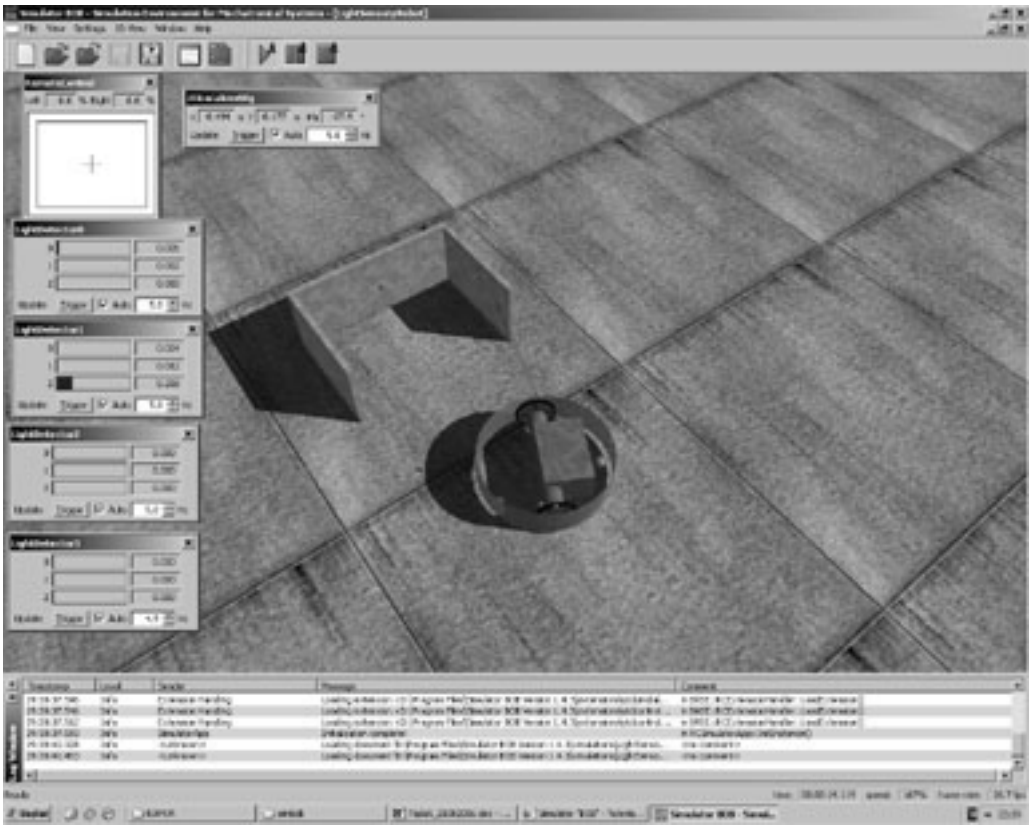

Figure 3.2 Simulator Bob User Interface

3.1.3 German Team

Recently many of the research on behavior-based robotic, is conducted on robot soccer, where two robot teams compete in a predefined soccer arena. Utilizing commercial SONY AIBO robots, robot dogs, a robotic soccer league is constructed and international tournaments are organized. In the mentioned tournament, since 2003 German team becomes the champion. The final release of their

behavior-based control software is called ”GERMAN TEAM” and designed to evaluate different behaviors on AIBO by simulation and then transfer the resulting successful behaviors into real robotic hardware. Although the project became open source, the complex user interface and the necessity to learn XABSL(Extended Agent Behavior Scripting Language) makes this software hard to apply. Very detailed but complex user interface of German Team is given in Figure 3.3. Refer to (www.germanteam.org) for further details.

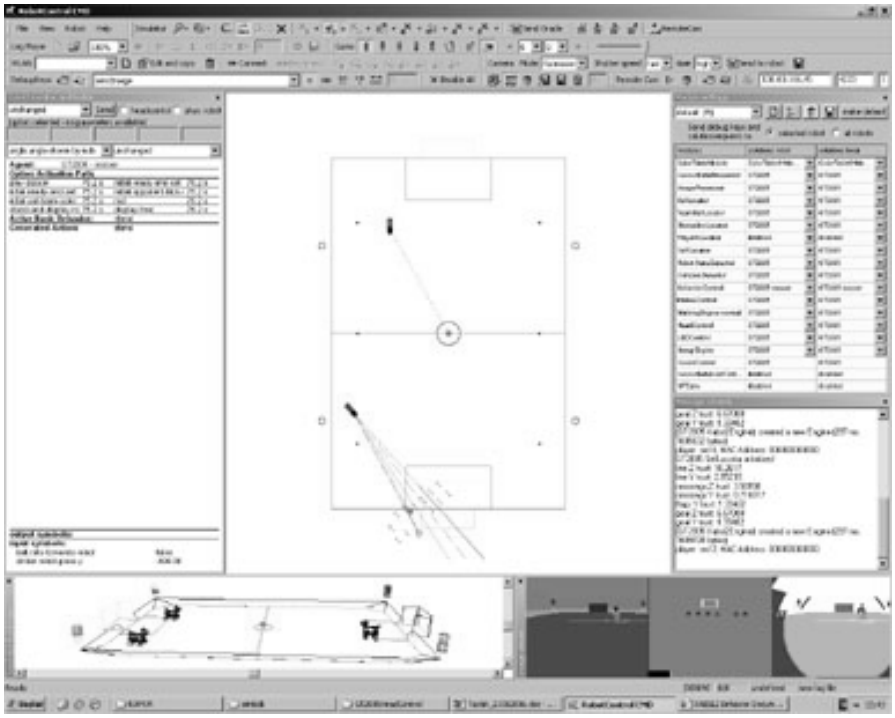

Figure 3.3 GermanTeam AIBO Soccer Simulation Software

3.1.4 MobotSim

MOBOTSIM by MobotSoft (www.mobotsoft.com) is software for 2D simulation of differential drive mobile robots. It provides a graphical interface that represents an environment in which you can easily create, set and edit robots and objects. In order to set these mobots in motion MOBOTSIM has a BASIC Editor in which the user can write macros making use of specific functions to get information about

mobots coordinates and sensor data and to set speed and driving data for them, as well as making use of all the power and ease of BASIC language to program navigation techniques.

MOBOTSIM has been developed thinking in researchers, students, roboticists and hobbyists who want to design, test and simulate mobile robots and research topics like autonomous navigation techniques, obstacle avoidance, artificial intelligence, a-life, data sensor integration, etc. An easy to understand user interface of MobotSim is given as a screenshot still image in Figure 3.4.

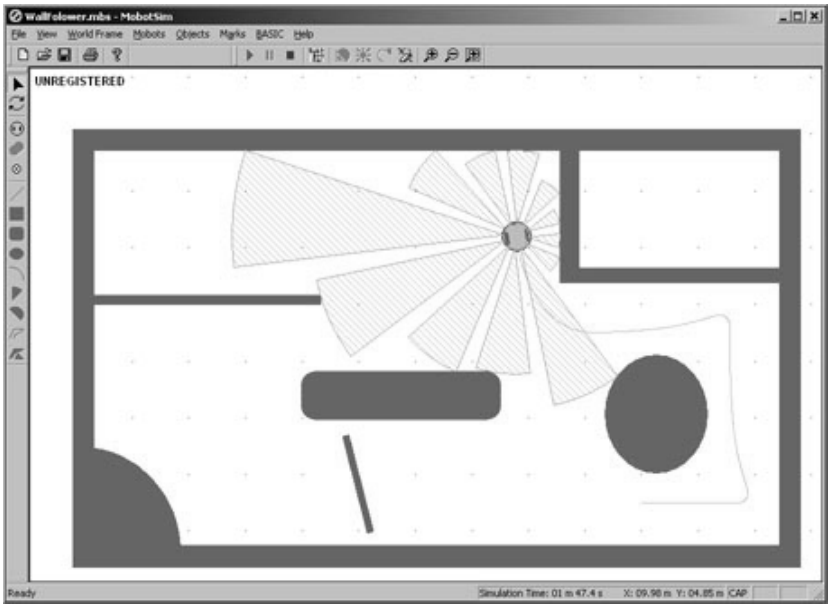

Figure 3.4 A Khepera Robot Navigating in 2D World, MobotSim

3.1.5 Webots

Webots is professional mobile robot simulation software, developed by Cyberbotics in cooperation with EPFL, Lausanne. It contains a rapid prototyping tool allowing the user to create 3D virtual worlds with physics properties, such as mass repartition, joints, friction coefficients, etc. The user can add simple inert objects or active objects called mobile robots. These robots can have different locomotion schemes (wheeled robots, legged robots or flying robots).

Moreover, they are equipped with a number of sensor and actuator devices, like distance sensors, motor wheels, cameras, servos, touch sensors, grippers, emitters, receivers, etc. Finally the user can program each robot individually to exhibit a desired behavior. Webots contains a large number of robot models and controller program examples that help the users get started.

Webots also contains a number of interfaces to real mobile robots, so that once your simulated robot behaves as expected, you can transfer its control program to a real robot like Khepera, Hemisson, LEGO Mindstorms, Aibo, etc.
Webots is well suited for research and education projects related to mobile robotics. Many mobile robotics projects have been relying on Webots for years in the following areas:

- Mobile robot prototyping (academic research, automotive industry, aeronautics, vacuum cleaner industry, toy industry, hobbyist, etc.).
- Multi-agent research (swarm intelligence, collaborative mobile robots groups, etc.).
- Adaptive behavior research (Genetic evolution, neural networks, adaptive learning, AI, etc.).
- Mobile robotics teaching (robotics lectures, C/C++/Java programming lectures, robotics contest, etc.).

In Figure 3.5, a Webots simulation screenshot is given.

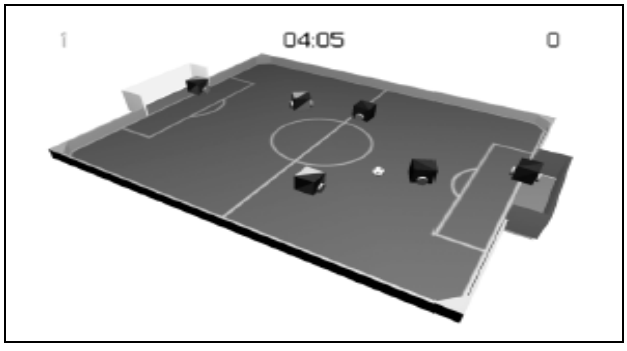

Figure 3.5 A Soccer Simulation in Webots [63]

3.2 SOFTWARE IMPLEMENTATION

Some of the most commonly used robot control programs have been investigated so far. Although they all serve to robot control concept with different properties only Webots, which is a commercial product, gives the user to define different behaviors. No open source program, yet having the comparable capabilities, has been released yet. German Team released their source code on AIBO, but this is only applicable to Sony AIBO.

Investigating the previous commonly used software on mobile robot control, shows that there is no support for behavior based control algorithms. All software focuses on single algorithm evaluation on mobile robots. The parallel execution/evaluation concepts do seem to be out of focus. Different algorithms need a integration block, a selector of algorithm outputs. In this thesis, a software capable of implementing some integration block that evaluates final command as the output from different behaviors' individual output commands is targeted. The software is also capable of showing the common top most capabilities such as virtual reality graphical representations, simple user interfaces, data logging (load, save) etc..

In this section, implementation details are given in subsections. These include structural properties of the software, implemented modules, and user interface issues.

3.2.1 Structural Properties

The software should have been user and developer friendly, meaning that it should not be complex neither for users nor for developers. Object-oriented programming gives the developers the feeling of what is going on actually within the code, since the blocks of code are packaged by containers named classes. These classes show the resemblance to the functional and logical real world agents. From the real world perspective, a robot exists in an environment. Actually, everything exists in

an environment. A robot has some features including sensing capabilities and intelligence. Sensors can be taught as hands or eyes, and the intelligence can be taught as the brain. As it is explained above, any real scene can be represented as functional logical packages. Object-oriented programming let us use this logical packaging representation in coding. The objects are coded as classes. Classes can be constructed by gathering different functional objects, including other classes.
In the case of robotic control, the classes logically are robots, their functional sub-blocks (sensors etc.), and the other logically packaged real objects, like walls, boxes.

After pointing to the general object-oriented programming issues, the structural properties of the implemented software can be given. The software is implemented in an object-oriented approach. The real world case is investigated and this case is divided into functional and logical parts as classes.

The decided classes from the investigation can be listed as follows:

- World,
- Robot,
- Sensor,
- Behavior,
- Behavior Controller.

These classes are constructed using the real world case, where many robots interacting with each other and other objects such as boxes and walls. The encapsulation information within the software is given in Figure 3.6 and Figure 3.7. The listed classes do not show the all classes implemented. There are more than 60 classes in the project, including GUI dialog classes, auxiliary classes such as camera class, quaternion class, Euler class, box class, motor command etc.
CWorld is the main application class that encapsulates all other classes and simulation specific functions.

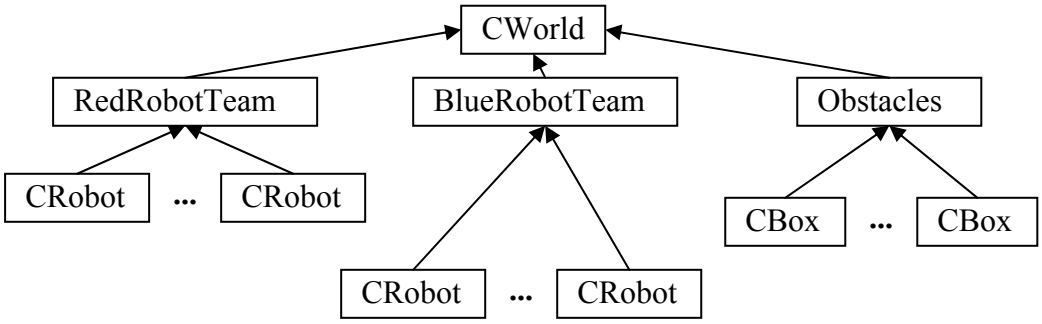

Figure 3.6 Topmost Class Hierarchy

Figure 3.6 shows the topmost class hierarchy, within the software. World is composed of mainly three parts, Red and Blue Robot Teams, and Obstacles. The number of child registered to each of these main groups is dynamic and unlimited in theory. Since the number of elements registered into the world results in computation time and memory, the number of robots in each robotic team in the world is limited to be at most five. Totally, ten robots can reside in the simulation at the same time.

According to the structure, every other object is under the control of CWorld object. Every object is registered into this world and every sub-property can be controlled within this CWorld parent object. As it is the case in the CWorld, CRobot is also a class that encapsulates other classes, namely CSensor, CBehavior, CBehaviorCoordinator. There are individual classes like CBox that are constructed to represent the real world obstacles. These individual boxes are grouped within the world to ease the controllability of the simulation and memory issues.

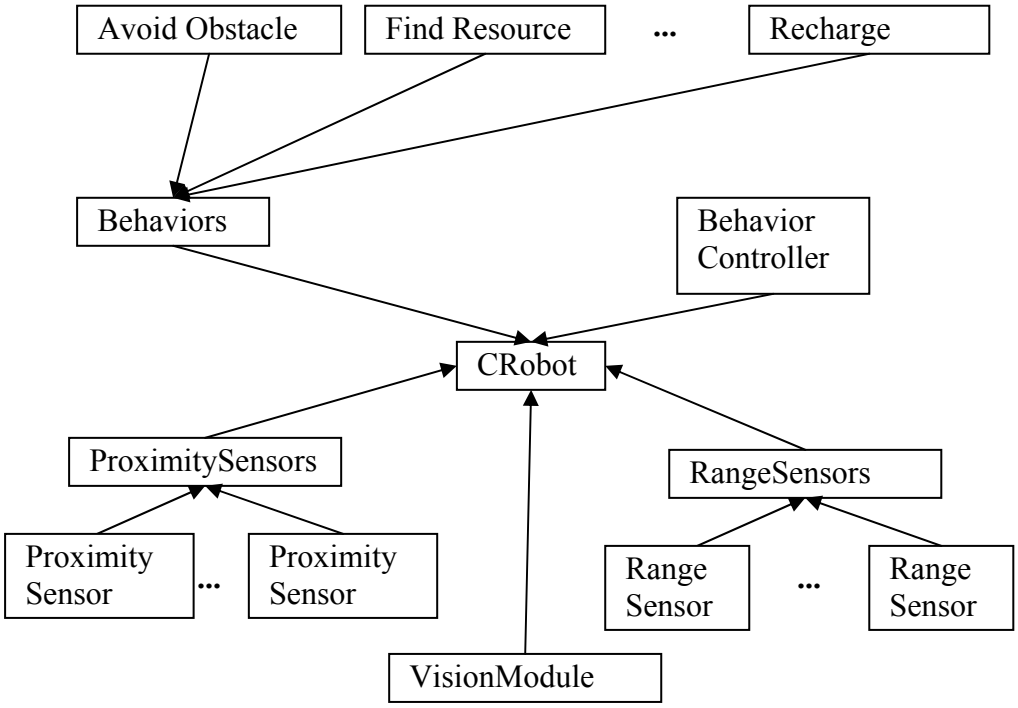

Figure 3.7 Robot Class Hierarchy

3.2.2 Implemented Modules

Software is divided into objects that can be gathered to build different robotic systems. These include environment, static, and dynamic obstacles, mobile robots, sensors, behaviors, and behavior controllers.

3.2.2.1 World

World is defined as the environment in which robots interact each other and the surrounding. In fact, every object lays in the world, including obstacles, robots hence their sensors, behaviors, and controllers. In the simulation framework, the world is defined as a rectangular flat area surrounded by walls that limits the coverage. The robots are all located inside this area. The user is capable of manipulating the contents of the world by adding new dynamic or static obstacles. User can add different size bricks to change the interior paths and construct

different worlds for the robots to interact and experiment. Maze's can be defined for the robots or corners with different sharpness for evaluation of different navigation algorithms. Various worlds constructed for different experiments can be seen on Figure 3.8.

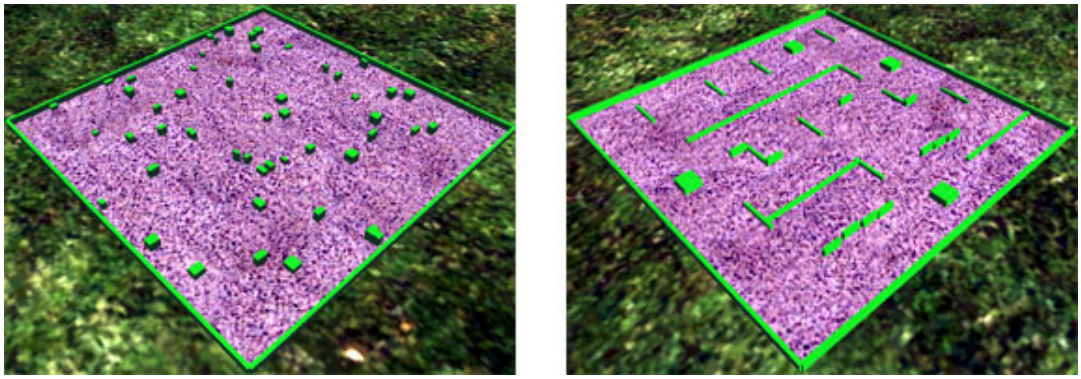

Figure 3.8 Different World Physical Configurations

3.2.2.2 Mobile Robots

Many different mobiles with different physical configurations are evaluated for the simulated robot. The research conducted on this can be seen on [Appendix A]. Any robot that can perform arbitrary rotations and translations is "holonomic". A non-holonomic system is the one in which the actuators do not directly control one or more of the degrees-of-freedom of the system, but instead are coupled such that orientation becomes much more complicated than in a holonomic system. The complexity of control in a non-holonomic system, guides us through the holonomic systems. The basic solutions to kinematics equations of two wheel differential drive systems, is another aspect that is evaluated and given in [Appendix B]. Complementary material can be found about omni-directional and non-holonomic robot control in [4], [7] and [10], for comparison with holonomic robot control.

The research showed that the most convenient mobile robot for navigation in laboratory environment with minimum difficulty in construction is a two-wheel differential drive robot, a holonomic robot. The robot simulated in this thesis is a two-wheel differential drive robot with roller caster wheels. The mobile robot

is in box shape and moves in 3D space. Robot moves as the two motors drives the two wheels on the right and the left side of it. The same but different rotational speeds' of the motor yields the robot to turn in an arc through the motors with less angular velocity. The same rotational speeds, lets the robot to move forward and backwards. The same rotational speeds to different directions, lets the robot to turn in its main cylindrical axis. The maneuverability of the robot makes it ideal for navigational purposes. Figure 3.12 and Figure 3.13 shows some possible translations and rotations with differential drive systems. The construction ease also makes this omni directional robot the main choice for laboratory experimental uses.

The two-wheel differential drive mobile robots are chosen as the target mobile robots in this thesis. These robots are implemented using ODE primitives, bodies and geoms. The implemented mobile robots consist of a main chassis in box shape, two wheels in cylindrical shape, and two support caster wheels in spherical shape. The geometrical shapes are chosen to enhance the collision algorithm hence shorten the computation time. Any geometrically and dynamically definable mobile robots can be constructed using the software, although the targeted mobile robots are limited to differential drive mobile robots. However the various parameters that can be defined is limited intentionally by the user interface, so only two wheel differential drive, above mentioned mobile robot is implemented.

In the software, ODE primitives construct mobile robots. The driving wheels are supposed to be turning in one axis so these wheels are attached to the robot chassis by hinge joints, which have one rotation axis. Y-axis of the robot coordinate system is the main wheel axis, about which the wheels rotate. Figure 3.9 shows the hinge type joint.

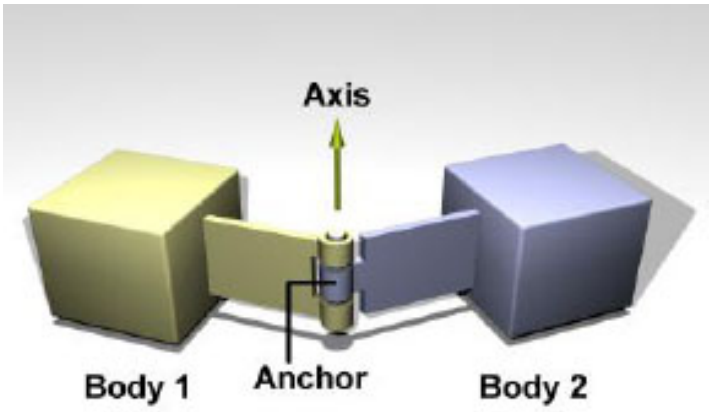

Figure 3.9 Hinge Type ODE Joint Used for Driving Wheels

The caster wheels should be free to turn not only in one axis but also in any axis. Therefore, these wheels are connected to the main robot chassis by ball-socket type joints. The ball-socket type joint is given in Figure 3.10.

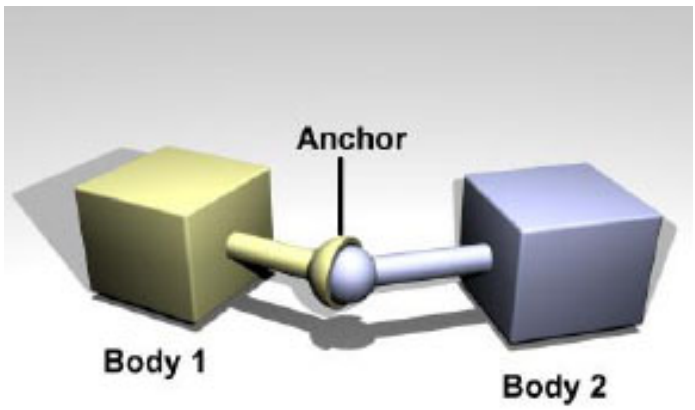

Figure 3.10 Ball-Socket Type ODE Joint Used for Caster Wheels

Using the joints and geometric primitives that used for collision checking, some examples of the constructed mobile robots look as they are given in Figure 3.11.

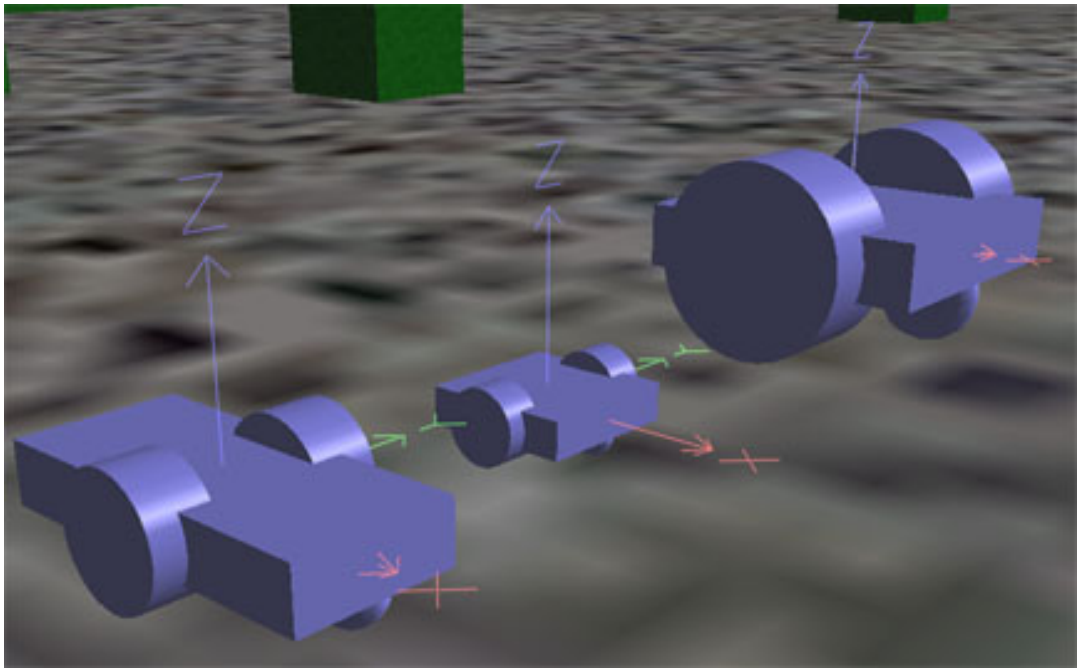

Figure 3.11 Different Size Mobile Robots

The user can define many different mobile robot parameters. Some of these parameters can only be set at the initialization of the robots. Some parameters can be set at any time of the simulation.

The parameters that can be set at the initialization of the robot are as follows, and these parameters do not subject to change again during simulation:

- Robot main chassis size parameters: Width, Length and Height,
- Robot wheel radius,
- Robot main chassis mass,
- Robot wheel mass.

The user editable robot parameters at any time are as follows:

- 3D robot position,
- Maximum robot scalar speed,

- Coverage area,
- Leadership flag.

The mobile robots move in three dimensions during the simulation. Robots interact with static objects like walls and static non-moveable boxes according to the physical constraints. Robots' new position and heading angle are calculated using the dynamics of the mobile robots. The only controlled variables of the robot are the two wheel angular velocities. These velocities are calculated in algorithmic parts, behaviors. The calculated wheel angular velocities are then applied to robot dynamics as variables.

The collision detection algorithm, calculates the contact points robot make with other objects. According to this contact points, some slippage is induced using the predefined contact point slippage constants. The resulting robot pose is drawn in the main user interface screen, in real time.

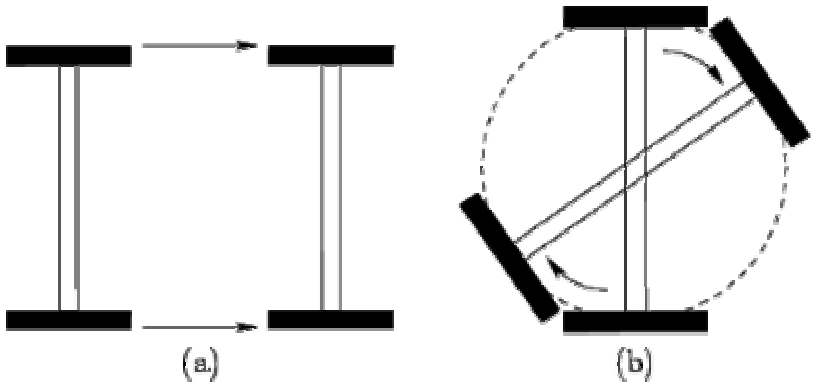

Figure 3.12 2WDD Robot Transformations (a) Pure Translation, (b) Pure Rotation

Pure translation occurs when both wheels move at the same angular velocity, pure rotation occurs when the wheels move at opposite velocities.

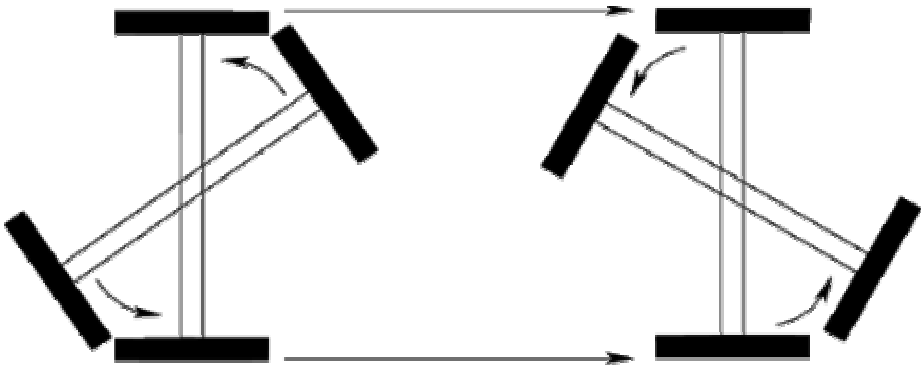

Figure 3.13 2WDD Robot Transformations

The shortest path traversed by the center of the axle is simply the line segment that connects the initial and goal positions in the plane. Rotations appear to be cost-free.

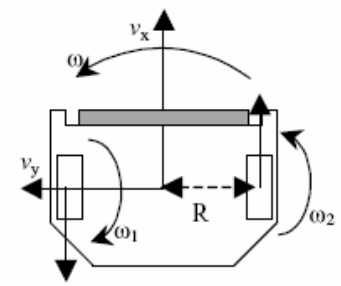

Figure 3.14 Reference Frames of the 2 Wheel Differential Drive Robot

The kinematics of the simulated differential drive robots is as follows, refer Figure 3.14:

All velocity commands $\underline{v} = (v_x, v_y, \omega)^T$ sent to the robot are defined in a robot relative reference frame. To translate the velocity commands into wheel motor angular velocity for each of the wheels, the velocity command is transformed using the inverse of the forward kinematics transform. In other words:

$$\underline{w} = T^{-1}\underline{v} \tag{3.1}$$

The inverse transform for the differential drive robot is given by;

$$T_{Diff}^{-1} = \begin{pmatrix} \frac{-1}{r} & 0 & \frac{R}{r} \\ \frac{1}{r} & 0 & \frac{R}{r} \end{pmatrix} \tag{3.2}$$

Where R is the distance of wheels to the robot main axis, robot radius, and the r is the wheel radius, having the same value for all wheels.

The simulated robot is physically inspired from the well known robot Khepera. Khepera is one of the mostly used mobile robot in robotic research. One configuration of Khepera is given in Figure 3.15 and Figure 3.16.

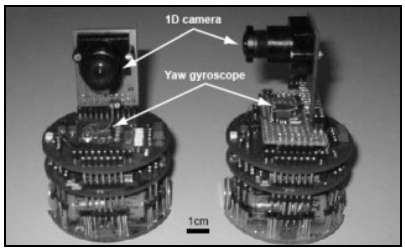

Figure 3.15 Khepera I Equipped with a Vision Turret

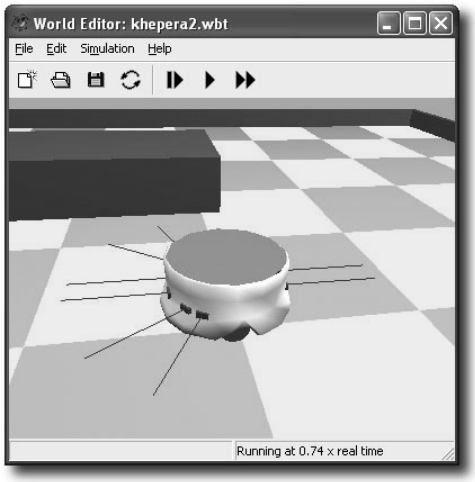

Figure 3.16 Simulated Khepera II in Webots

Since the simulation environment is coded from object oriented point of view, the robot parameters can be tuned to simulate any two wheel differential drive mobile robot. In order to emphasize the flexibility of the software environment, these parameters can also be tuned as to simulate Khepera I or Khepera II physical parameters. Also the sensor configuration can be arranged to be identical to Khepera robots.

The 3D representation of a simulated mobile robot is given in Figure 3.17.

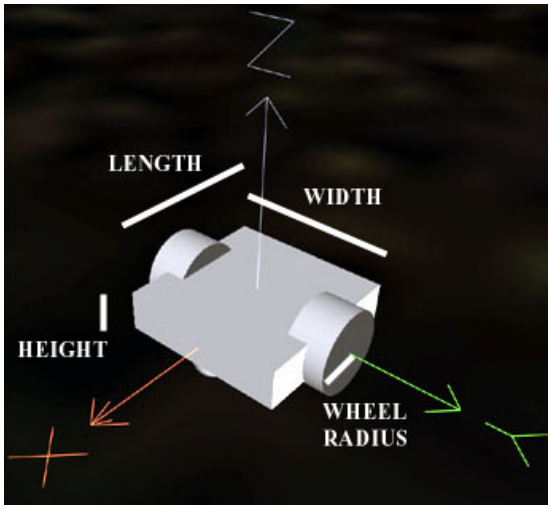

Figure 3.17 3D Representation of Mobile Robot

Derivations on the controllability of the differential robot among the other wheeled robot configurations are given in [Appendix B].

Many different mobile robots can be designed using the software presented in this thesis. The designs are not limited to wheeled robots. Legged robots can also be designed once the geometrical and dynamical properties of the robot are determined. Segmented robots can also be simulated using the software. Figure 3.18 shows an example for a three segment, train type mobile robot. The segments are linked each other by joints with two degrees of rotational freedom.

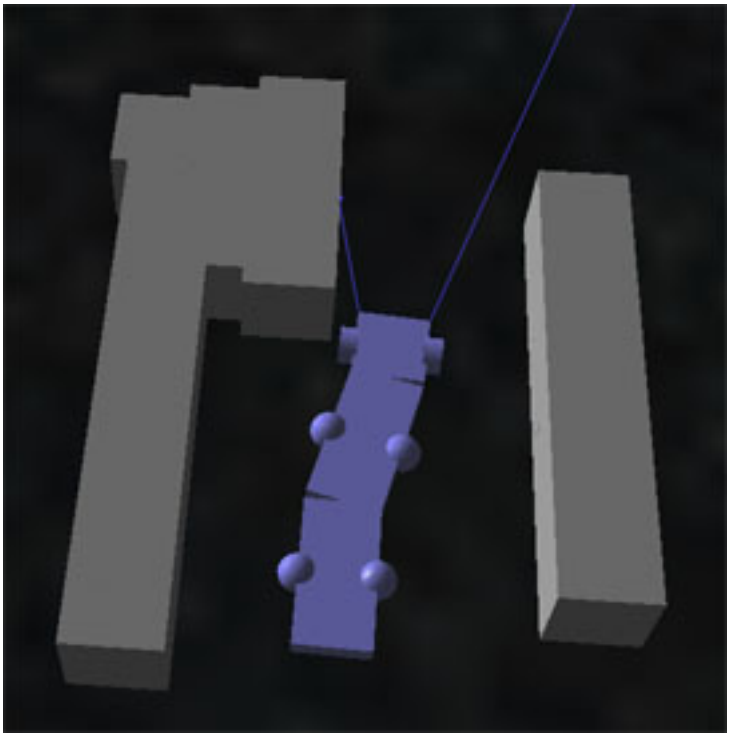

Figure 3.18 Three Segment Mobile Robot

3.2.2.3 Sensors

Since the only information flow between the robot and the environment is achieved via sensors on the robots, different sensors are simulated for different experimental simulations. Since the real sensors are not error prone, some percentage of measurement error is also injected to the data channel of each sensor inherently. The simulated sensors are chosen to cover the real sensor types that are necessary to construct a robot with enough sensor complexity.

3.2.2.3.1 Volume/Proximity Sensors

These sensors are the type of sensors that indicates if there is any object within the coverage of the sensor or not. The output of the sensor is a logical indicator with two alternating values, 0 and 1. These sensors are the virtual corresponding of

proximity sensors. These types of sensors are widely used in any industry and also in robotics, although they do not give any detailed information, except the existence of any object. The maximum range, horizontal and vertical FOV angles are the parameters that can be manipulated by the user. Robots with different proximity sensor configurations (number of sensors, FOV angles and range) can be seen in Figure 3.19 and Figure 3.20 (Both are converted into grayscale to increase readability).

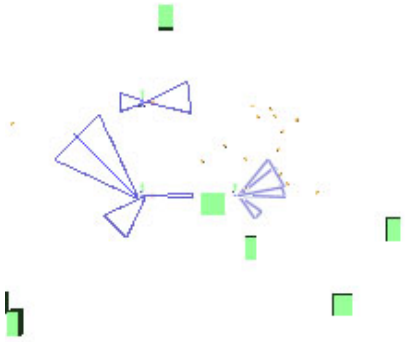

Figure 3.19 Top View Showing Robots with Different Proximity Sensor Configurations

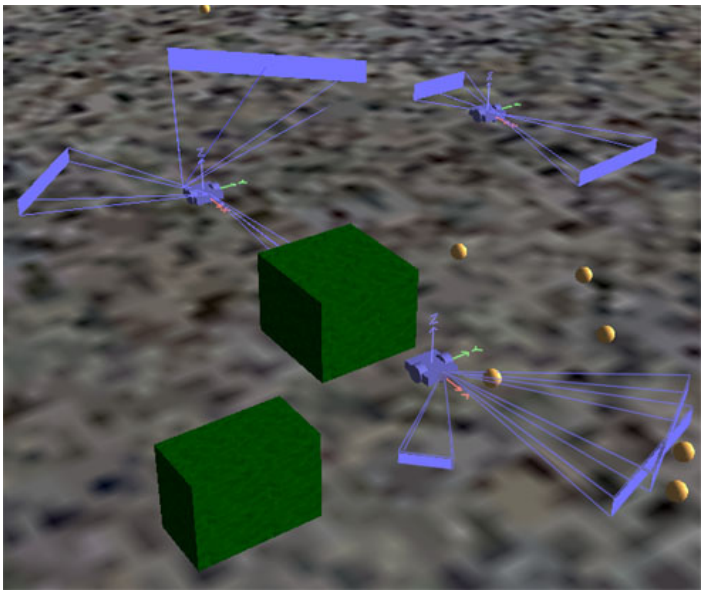

Figure 3.20 Closer View of Robots with Four Proximity Sensors

3.2.2.3.2 Beam/Range Sensors

These sensors are the type of linear proximity sensors with distance measurements. These sensors are the virtual co-responders of laser or narrow ultrasonic transducers. They measure the distance between the robot and the closest object. Since the real sensors do not measure the exact distance, also a percentage error channel is mixed with the exact distance. The mixed channel is fed out as the output of the sensors. The error percentage can be chosen by the user and can also be changed runtime. Figure 3.21 shows a robot with many beam range sensors, from different camera locations.

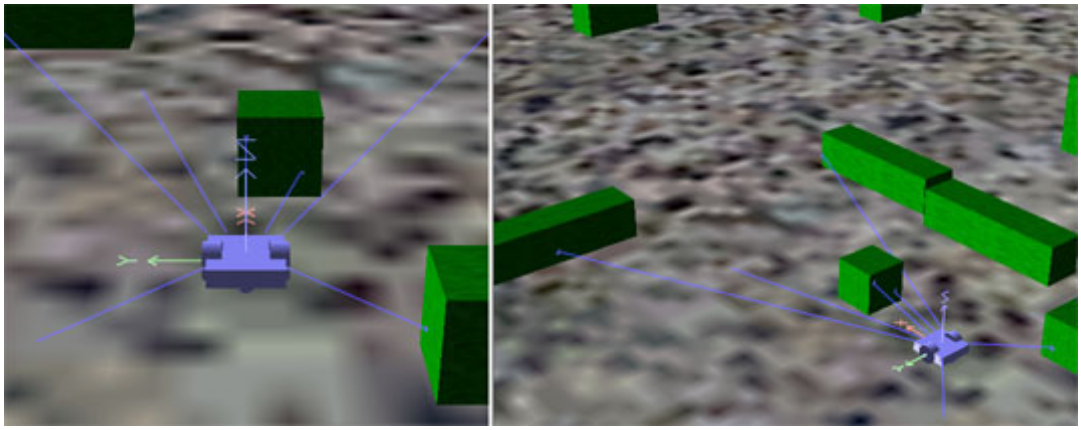

Figure 3.21 Robot Equipped with Many Beam/Range Sensors

3.2.2.3.3 Vision Module

Since there are numerous researches on robotic field involving vision through cameras of different types, a vision module is also included to the available modules. The simulated vision module gives a representation of the environment in 100 pixels by 100 pixels format. The size of the scene is limited to 100x100 pixels due to the processing time of total 10 robots' vision modules, 5 robots in each team. In order to process the image taken by the vision module, OpenCV (Open Source Computer Vision) library is embedded to the software environment.

The output scene can be processed by the available functions of OpenCV. The module gives 3 channel RGB representation of the environment but this raw image can be transformed into different channels of interests (COI) such as HSV using the transformation functions of OpenCV. In Figure 3.22**,** two outputs of the vision module is given. Left image shows the raw image taken by the module, whereas right one shows the image proceesed by the OpenCV functions to locate the red box in the scene. Details on vision-based behaviors can be found in [60], for further details.

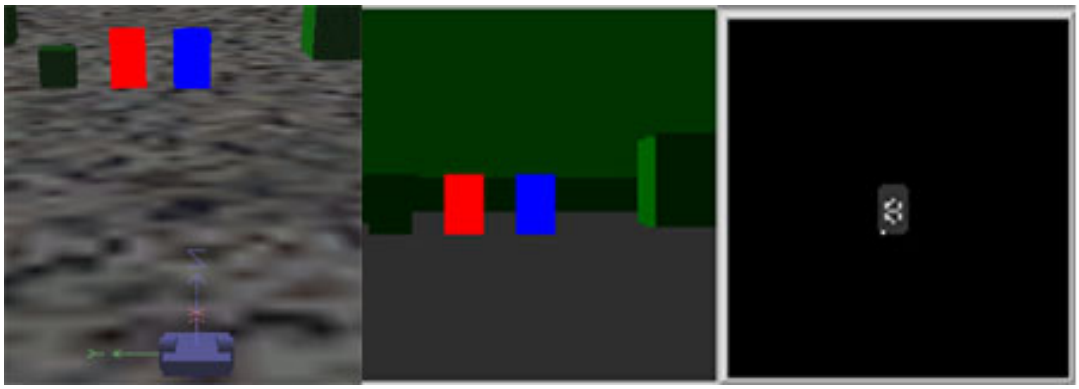

Figure 3.22 Images Showing the Robot Approaching the Resources (left image), the Raw (middle image) and the Processed Outputs of the Vision Module (right image)

3.2.2.4 Behaviors

Behaviors constitute the algorithmic part of the simulation. Behaviors are the actions that robot reacts to sensor readings. Because of the parallel structure of the behaviors, user can develop different and unlimited robot behavior. Individual behaviors can be loaded to any robot with any combination yielding different results.

Every robot can be loaded with any number of available behaviors. These behaviors are registered to the robot. At runtime robot applies all the behaviors that are registered to it. Every behavior returns some modification on robots' state. In order to effectively manipulate the robot, some kind of behavior fusion is required. Different strategies can be applied as decision process. Every behavior is

given a priority to be used in behavior selection (arbitration, fusion) process.

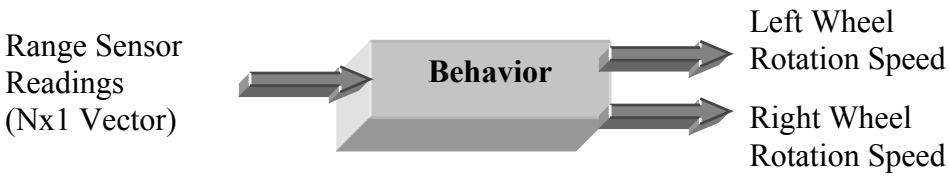

Figure 3.23 Behavior Input-Output Relationship

The input to any behavior is the sensor readings. The output of any behavior is the angular velocities of the right and left wheels. The difference between the velocities of the wheels, results in another posture of the robot in 3D space, both in translational and rotational means.

Different behavior programming techniques can be found in the literature, including genetic and evolutionary programming, [24, 33, and 45]. Since all these require some sort of learning these are omitted in the scope of this thesis. Since all these evolved behaviors can be considered as functions parametrically optimized in some sense, these parameters can be coded as a normal C code and serve as they should in this software environment.

The behaviors are coded with the knowledge obtained from [29, 57, and 42] although the design methodologies are not commonly accepted. The simulated sample behaviors in the simulation environment are given in the following sub-sections: The sample behaviors cover the minimal set of behaviors suggested for a mobile robot given in [25].

3.2.2.4.1 Avoid Obstacle

This behavior serves as the avoidance algorithm. Behavior uses only the beam range sensors if exists on the robot. According to the encountered obstacle relative angular position and range, this behavior invokes the commands to turn right, turn left, or go straight. The turn rates are determined according to the obstacles sensed. This behavior also affects the robots' linear velocity and sets it according to

the obstacles relative distance to the robot. For comparison, an obstacle avoidance algorithm based on behavior can be seen in [52].

The algorithm can be summarized as follows:

- Create an empty motor command in which behavior response will be written to,
- Cycle through all range sensors and apply their range measurements and their relative angles with respect to the robot reference frame to fuzzy rules,
- Accumulate the output commands for each sensor,
- Increase the effect of the sensor with the minimum range measurement by accumulating that sensor's motor command twice,
- Divide accumulated command into the number of participating sensors plus one,
- Send final motor command for this behavior to behavior coordinator.

The code segment can be investigated in Figure 3.24. This code is the main method of CBAvoidObstacle class that is called from the CRobot parent when this behavior is loaded to the robot.

The obstacle avoidance behavior is defined by a fuzzy controller. This fuzzy controller accepts two input variables and gives one output variable. The controller is executed for each Beam/Range Sensor. Inputs parameters for the controller are sensors' measurement and sensors' relative angle with respect to the z-axis of the robot. Since the behaviors have two outputs, right and left wheel angular velocities, two fuzzy controllers are used.

One fuzzy controller is responsible for the right wheel speed, while another fuzzy controller determines the rate of rotation for the left speed.

```
CMotorCommand CBAvoidObstacle::Apply()
{
    #define RANGE_MEASUREMENT   0 // range_measurement is our 1st variable
    #define SENSOR_AZIMUTH      1 // sensor_azimuth is the 2nd variable
    //Construct a motor command
    mycommand = CMotorCommand(0.0f, 0.0f);
    float minmeasurement = 10;
    int mini;
    //Cycle through all range sensors and accumulate their
    //commands according to fuzzy rules
    for (int i = 0;i<Parent->RangeMeasurements.size(); i++)
    {
        if ( Parent->RangeMeasurements[i] < minmeasurement)
        {
            minmeasurement = Parent->RangeMeasurements[i];
            mini = i;
        }
        ffll_set_value(rightmodel, rightchild, RANGE_MEASUREMENT, Parent->RangeMeasurements[i]);
        ffll_set_value(rightmodel, rightchild, SENSOR_AZIMUTH, Parent->BeamRangeSensors[i]->GetRotz());

        ffll_set_value(leftmodel, leftchild, RANGE_MEASUREMENT, Parent->RangeMeasurements[i]);
        ffll_set_value(leftmodel, leftchild, SENSOR_AZIMUTH, Parent->BeamRangeSensors[i]->GetRotz());
        mycommand += CMotorCommand (    ffll_get_output_value(leftmodel, leftchild),
                                        ffll_get_output_value(rightmodel, rightchild)
                                    );
    }
    //Increase the effect of the sensor with min. range measurement
    ffll_set_value(rightmodel, rightchild, RANGE_MEASUREMENT, Parent->RangeMeasurements[mini]);
    ffll_set_value(rightmodel, rightchild, SENSOR_AZIMUTH, Parent->BeamRangeSensors[mini]->GetRotz());

    ffll_set_value(leftmodel, leftchild, RANGE_MEASUREMENT, Parent->RangeMeasurements[mini]);
    ffll_set_value(leftmodel, leftchild, SENSOR_AZIMUTH, Parent->BeamRangeSensors[mini]->GetRotz());
    mycommand += CMotorCommand (    ffll_get_output_value(leftmodel, leftchild),
                                    ffll_get_output_value(rightmodel, rightchild)
                                );
    //Devide accumulated command into the number
    //of participating sensors+1
    if (Parent->RangeMeasurements.size() > 0)
        mycommand /= (Parent->RangeMeasurements.size()+1);
    //Send final motor command for this behavior
    return mycommand;
}
```

Figure 3.24 CBAvoidObstacle Class Apply Method

An example of fuzzy input and output variables with triangular membership functions and fuzzy rules with COG. defuzzification method is given in Figure 3.25.

3.2.2.4.2 Find Resource

This behavior uses visual information. This behavior is meaningful only when the robot has a vision module, since it feeds from this module's outputs. The algorithm is straight forward:

- Take the resources detected coordinates in vision modules' output pixel format,
- Take the vision modules' FOV angle
- Convert the detected resource location into turning command.

The code segment that CBFindResource behavior applies can be seen in Figure 3.26.

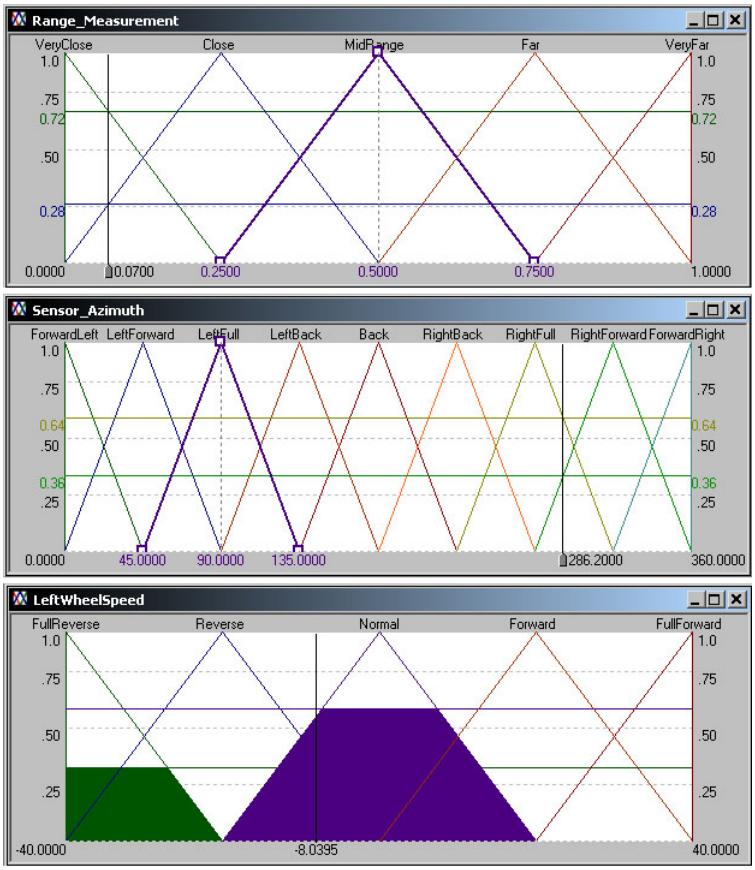

Figure 3.25 Fuzzy Controller View for Left Wheel Speed Control Variable

3.2.2.4.3 Follow the Leader

This behavior is used to investigate the group formation behavior. Once this behavior is loaded to a robot, the robot tries to align itself to a Leader robot with in its' communication coverage area. Figure 3.27 shows the implemented algorithm in CBFollowTheLeader behavior class.

```
CMotorCommand CBFindResource::Apply()
{
    float FOV = 0.0f;
    if (Parent->VisionModule.size() !=0)
        FOV = Parent->VisionModule[0]->GetFov();

    float x_center = Parent->x_c;
    if ( (x_center-50.0f) < -5.0f )
    {
        //resource on left so turn left
        //adjust turn rate according to measured resource location
        mycommand = CMotorCommand(-fabs(x_center-50.0f)/FOV, fabs(x_center-50.0f)/FOV );
    }
    else if ( (x_center-50.0f) > 5.0f )
    {
        //resource on right so turn right
        //adjust turn rate according to measured resource location
        mycommand = CMotorCommand(fabs(x_center-50.0f)/FOV, -fabs(x_center-50.0f)/FOV );

    }
    else
    {
        //resource infront so go straight
        //increase speed in this case

        mycommand = CMotorCommand( 20.0f, 20.0f);
    }

    return mycommand;
}
```

Figure 3.26 CBFindResource Class Apply Method

```
CMotorCommand CBFollowTheLeader::Apply()
{
    mycommand = CMotorCommand(0.0f, 0.0f);
    if ((!Parent->leader) && (Parent->slave))
    {
        //Find the angle between the robot and the Leader robot
        float theta = atan2(    (Parent->Leader2Follow->y) - (Parent->Position->y),
                                (Parent->Leader2Follow->x) - (Parent->Position->x) );
        float theta2 = atan2(Parent->MotionModelview[1], Parent->MotionModelview[0]);

        //Convert radians into degrees
        theta = (theta - theta2) * 180.0f / (float) PI;
        //Find the distance to the Leader robot
        float distance = sqrt( (pow ( (Parent->Leader2Follow->y) - (Parent->Position->y),2) +
                                pow ( (Parent->Leader2Follow->x) - (Parent->Position->x),2))
                             );
        if (distance >=0.5f)            //Relatively Far
        {
            if( theta> 10.0f )          //Turn Left with theta strength
                mycommand = CMotorCommand(-theta*0.10f, theta*0.10f);
            else if (theta < -10.0f)    //Turn Right with theta strength
                mycommand = CMotorCommand(-theta*0.10f, theta*0.10f);
            else                        //Go Straight
                mycommand = CMotorCommand( 40.0f, 40.0f);
        }
        else if (distance >=0.3f)       //Relatively Close
        {
            if( theta> 10.0f )          //Turn Left with theta strength
                mycommand = CMotorCommand(-theta*0.20f, theta*0.20f);
            else if (theta < -10.0f)    //Turn Right with theta strength
                mycommand = CMotorCommand(-theta*0.20f, theta*0.20f);
            else                        //Go Straight
                mycommand = CMotorCommand( 30.0f, 30.0f);
        }
        else                            //Very Close
        {
            mycommand = CMotorCommand( 0.0f, 0.0f);
        }
    }

    return mycommand;
}
```

Figure 3.27 CBFollowTheLeader Class Apply Method

3.2.2.4.4 Recharge

This behavior uses internal states not the sensor readings from the outer environment. The battery status is watched as an internal state for the robot. This behavior guides the robot if the conditions are satisfied, towards the predefined recharging area defined for the team that robot belongs, red or blue. The condition occurs when the energy level decreases below a predetermined level. shows CBRecharge class's Apply method.

```
CMotorCommand CBRecharge::Apply()
{
    #define ENERGYSTEP 25;
    if ( (Parent->GetEnergy() <= (Parent->MaxEnergy/10.0f)) && (!Parent->ChargingStatus))
    {
        //Find the angle between the robot and the charger position
        float theta = atan2(    (Parent->Charger->y) - (Parent->Position->y),
                                (Parent->Charger->x) - (Parent->Position->x) );
        float theta2 = atan2(Parent->MotionModelview[1], Parent->MotionModelview[0]);
        //Convert radians into degrees
        theta = (theta - theta2) * 180.0f / (float) PI;
        //Find the distance to the charger
        float distance = sqrt( (pow ( (Parent->Charger->y) - (Parent->Position->y),2) +
                                pow ( (Parent->Charger->x) - (Parent->Position->x),2))
                             );
        //Approach charger according to the conditions met
        if (distance >=0.5f)            //Relatively far
        {
            if( theta> 10.0f )          //More than 10 degrees
                mycommand = CMotorCommand(-theta*0.10f, theta*0.10f);
            else if (theta < -10.0f)    //More than -10 degrees
                mycommand = CMotorCommand(-theta*0.10f, theta*0.10f);
            else                        //Between -10 and 10 degrees
                mycommand = CMotorCommand( 40.0f, 40.0f);
        }
        else if (distance >=0.1f)       //Relatively close
        {
            if( theta> 10.0f )
                mycommand = CMotorCommand(-theta*0.20f, theta*0.20f);
            else if (theta < -10.0f)
                mycommand = CMotorCommand(-theta*0.20f, theta*0.20f);
            else
                mycommand = CMotorCommand( 30.0f, 30.0f);
        }
        else                            //Very close
        {
            //Change status to "charging" if robot is close enough
            Parent->ChargingStatus = true;
            mycommand = CMotorCommand( 0.0f, 0.0f);
        }
    }
    else if (Parent->ChargingStatus)
    {
        //Increase the robots energy by some defined step
        Parent->SetEnergy( Parent->GetEnergy() + ENERGYSTEP);
        //If the battery is full then change status
        if ( Parent->GetEnergy() >= Parent->MaxEnergy )
        {
            Parent->SetEnergy(Parent->MaxEnergy);
            Parent->ChargingStatus = false;
        }
    }
    else
    {
        mycommand = CMotorCommand( 10.0f, 10.0f);
    }

    return mycommand;
}
```

Figure 3.28 CBRecharge Class Apply Method

3.2.2.4.5 Turn Left

This simple behavior forces the robot to turn left if there is an obstacle detected at the right side of the robot. This behavior uses beam range sensors as input space. The turn rates are proportional to the encountered obstacle's relative position.
This behavior is in fact right half of obstacle avoidance behavior. Turning left only needs the sensitivity to right obstacles, so removing the left sensors from the obstacle avoidance fuzzy rules, just gives the turn left behavior. Figure 3.29 shows the code presenting the algorithm in CBTurnLeft behavior.

```
CMotorCommand CBTurnLeft::Apply()
{
    #define RANGE_MEASUREMENT   0 // range_measurement is our 1st variable
    #define SENSOR_AZIMUTH      1 // sensor_azimuth is the 2nd variable
    //Construct a motor command
    mycommand = CMotorCommand(0.0f, 0.0f);
    float minmeasurement = 10;
    int mini;
    //Cycle through all range sensors and accumulate their
    //commands according to fuzzy rules
    for (int i = 0;i<Parent->RangeMeasurements.size(); i++)
    {
        if ( Parent->RangeMeasurements[i] < minmeasurement)
        {
            minmeasurement = Parent->RangeMeasurements[i];
            mini = i;
        }
        ffll_set_value(rightwheelmodel, rightchild, RANGE_MEASUREMENT, Parent->RangeMeasurements[i]);
        ffll_set_value(rightwheelmodel, rightchild, SENSOR_AZIMUTH, Parent->BeamRangeSensors[i]->GetRotz());

        ffll_set_value(leftwheelmodel, leftchild, RANGE_MEASUREMENT, Parent->RangeMeasurements[i]);
        ffll_set_value(leftwheelmodel, leftchild, SENSOR_AZIMUTH, Parent->BeamRangeSensors[i]->GetRotz());
        mycommand += CMotorCommand (    ffll_get_output_value(leftwheelmodel, leftchild),
                                        ffll_get_output_value(rightwheelmodel, rightchild)
                                    );
    }
    //Devide accumulated command into the number
    //of participating sensors
    if (Parent->RangeMeasurements.size() > 0)
        mycommand /= (Parent->RangeMeasurements.size());
    //Send final motor command for this behavior
    return mycommand;
}
```

Figure 3.29 CBTurnLeft Class Apply Method

The algorithm can be summarized as follows:

- Create an empty motor command in which behavior response will be written to,
- Cycle through all range sensors and apply their range measurements and their relative angles with respect to the robot reference frame to fuzzy rules,
- Accumulate the output commands for each sensor,
- Divide accumulated command into the number of participating sensors,
- Send final motor command for this behavior to behavior coordinator.

The algorithm seems to be not different from the one given for the obstacle avoidance algorithm with minor modifications. In fact, the fuzzy controller makes the major difference. The rules and input variables differ from the ones for the obstacle avoidance algorithm.

Figure 3.30 shows the difference (compare with the Avoid Obstacle behavior).

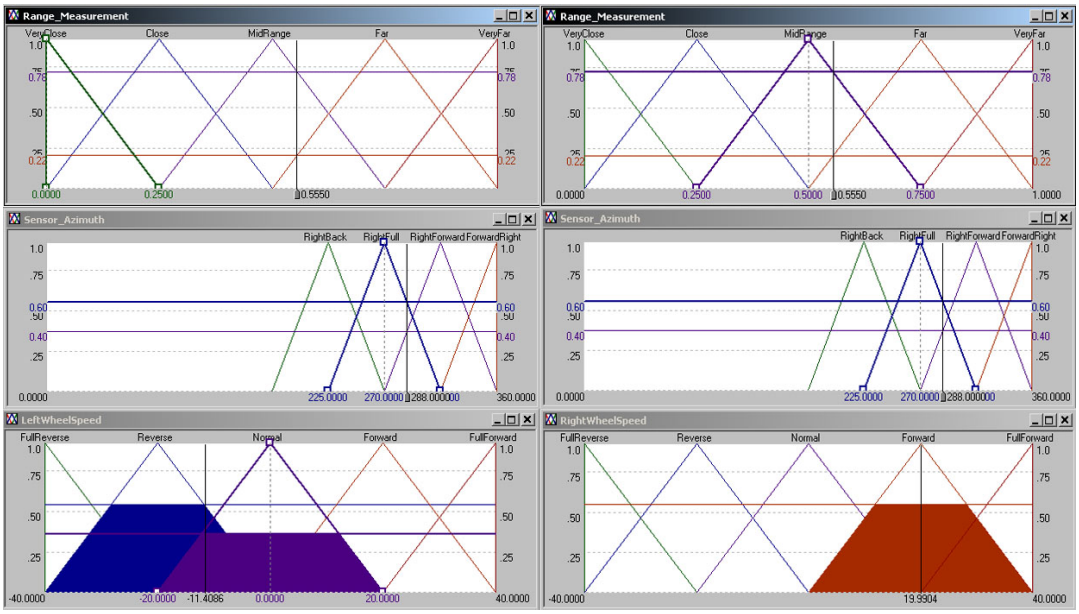

Figure 3.30 Turn Left Behavior Fuzzy Entries

3.2.2.4.6 Turn Right

This simple behavior forces the robot to turn right if there is any obstacle detected at the left side of the robot. This behavior uses only beam range sensors as input space. This behavior is in fact left half of obstacle avoidance behavior. Turning right only needs the sensitivity to left obstacles, so removing the left sensors from the obstacle avoidance fuzzy rules, just gives the turn right behavior.

Since the code is very similar to Turn Left behavior, code segment is not given.

3.2.2.5 Behavior Controllers

Most of the controllers discussed in Chapter 2 are implemented in the software developed in this study. They are given in following subsections:

3.2.2.5.1 Arbiter - Suppression

This behavior controller uses an arbitration policy. The implemented policy is the suppression type. The strategy can be summarized as follows:

- Find the behavior with the highest priority (determined by user)
- Apply only this behaviors' response as the output

The code segment used to apply the suppression policy is given in Figure 3.31. This code used in the CBCArbiterSuppression class, is the method that is called from the CRobot parent.

```
CMotorCommand CBCArbiterSuppression::Apply()
{
    //Construct a motor command
    CMotorCommand resultantcommand;
    int highest_priority = -5;
    int highest_priority_index = 0;
    //Find the behavior with the highest priority value
    for (int Behaviorindex = 0; Behaviorindex < Parent->BehaviorSet.size(); Behaviorindex++)
    {
            if (Parent->BehaviorSet[Behaviorindex]->GetPriority() > highest_priority)
            {
                highest_priority = Parent->BehaviorSet[Behaviorindex]->GetPriority();
                highest_priority_index = Behaviorindex;
            }
    }
    //Apply the highest prioritized behavior
    if ( Parent->BehaviorSet.size() > 0 )
        resultantcommand = Parent->BehaviorSet[highest_priority_index]->Apply();

    //Send final motor command
    return resultantcommand;
}
```

Figure 3.31 CBCArbiter Suppression Class Apply Method

3.2.2.5.2 Fusion – Vector Summation

This is the behavior controller that uses an command fusion policy. The

implemented policy is the vector summation type. The strategy can be summarized as follows:

- Cycle through behaviors and get their responses weighted by
- Accumulate them in order to find the final output

The code segment used to apply the suppression policy is given in Figure 3.32. This code used in the CBCFusionVectorSummation class, is the method that is called from the CRobot parent.

```
CMotorCommand CBCFusionVectorSummation::Apply()
{
    //Construct a motor command
    CMotorCommand resultantcommand;
    //Cycle through behaviors of the robot
    //and accumulate their commands
    for (int Behaviorindex=0; Behaviorindex<Parent->BehaviorSet.size();Behaviorindex++)
    {
            resultantcommand += Parent->BehaviorSet[Behaviorindex]->Apply();
    }
    //Devide accumulated command into the number
    //of participating behaviors
    resultantcommand /= (float) Parent->BehaviorSet.size();

    //Send final motor command
    return resultantcommand;
}
```

Figure 3.32 CBCFusionVectorSummation Class Apply Method

3.2.2.5.3 Fusion - Neural Network

This behavior controller applies a command fusion policy. This behavior controller is in fact different from Vector Summation policy only in the sense that the weights of the behaviors are assigned dynamically by a trained neural network. Then the weighted behaviors are fused by vector summation. The Neural Network is a fully connected feed forward neural network and is trained by a user defined training data file over back propagation algorithm, [Appendix C]. The training is accomplished as offline.

The neural network consists of N layers. Out of these N layers, the first and the last layers are input and the output layers respectively. The other N-2 layers are the hidden layers and they change the complexity of the network. The number

of hidden layers and the number of neurons in each of these layers affect the training performance of the network, hence the resulting outputs to the same input patterns.

The network topology can be defined by the user. The number of the Beam\Range Sensors automatically defines the number of input layer neurons. The number of output layer neurons is similarly assigned automatically by the number of behaviors of the selected robot. The user can define the number of hidden layers and the number of neurons in each of these hidden layers. The user is responsible to supply a suitable training data file according to the network he/she defines. The training data file is an ASCII text document, and can be written in any ASCII text editor, such as Notepad and/or WordPad.

The format of the training data file is as follows:

- Each line in the text file represents n input and m output variables to be used for training,
- The numbers are separated by each other by tabs,
- The first n numbers in a line should represent input pattern, hence the number of these variables should match the number of sensors in the robots to be trained,
- The remaining m numbers other than the n input patterns are the output patterns. The number of these variables should match the number of behaviors of the robot to be trained,
- The length (number of lines) of the training data is unlimited.

An example of a network defined by the above parameters is given in Figure 3.33.

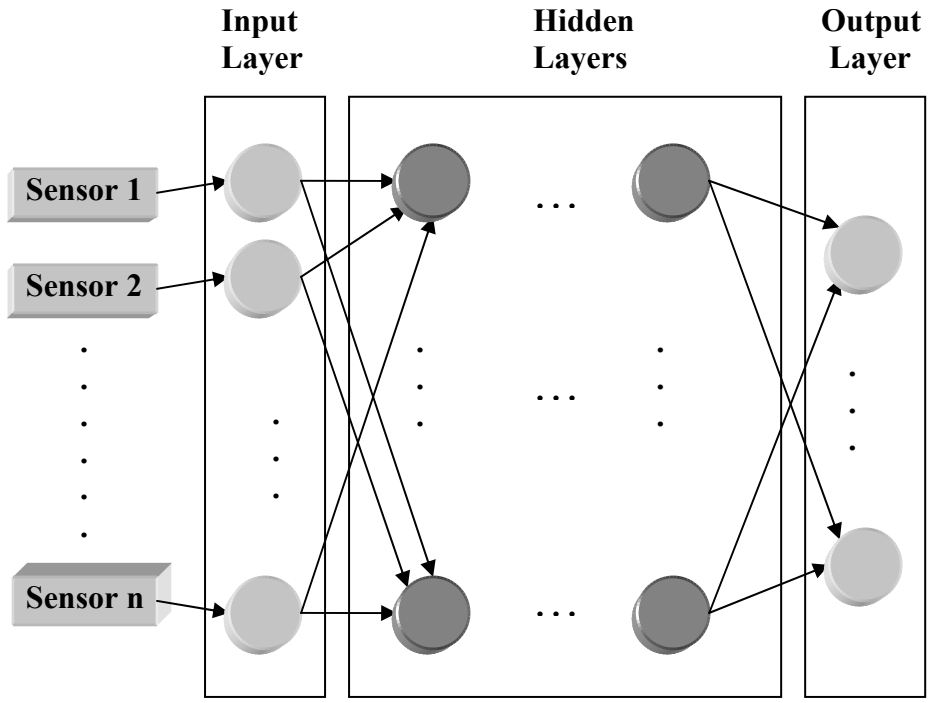

Figure 3.33 Topology of Neural Network

The training data-reading algorithm takes the location of the training data file as input. The user defines the location of this file via user interface. The file is searched until the end of the file is reached. During this search, every line is taken and evaluated to extract (n + m) floating-point numbers to match n sensors plus m behaviors. The extracted data is fed into the network as training data using back propagation algorithm. The example of a training data file is given in Figure 3.34. This file is used for training a robot with three Beam/Range Sensors and two behaviors.

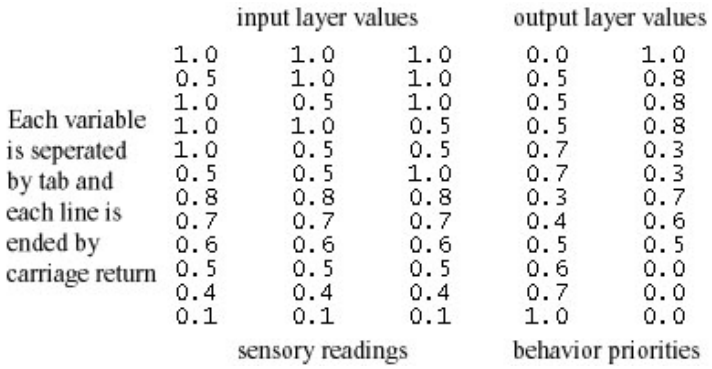

Figure 3.34 Neural Network Training Data File Sample

The dynamically changing priorities of the behaviors can be used in many different fusion techniques. Some of them can be arbitration such as selecting the highest prioritized behavior. Another can be command fusion of vector summation of behaviors weighted by their priorities.

The code segment used to apply the dynamic prioritization policy over Fusion technique is given in Figure 3.35. This code used in the CBCFusionNNBackProp class, is the method that is called from the CRobot parent.

```
CMotorCommand CBCFusionNNBackProp::Apply()
{
    //Construct a motor command
    CMotorCommand resultantcommand;
    //Create a variable for measurement datas from sensors
    std::vector <double> MeasurementData;

    for (int RangeSensorindex=0; RangeSensorindex<Parent->BeamRangeSensors.size();RangeSensorindex++)
    {
        MeasurementData.insert(MeasurementData.end(),Parent->RangeMeasurements[RangeSensorindex]);
    }
    //feed forward neural network
    bp->ffwd(&MeasurementData[0]);

    //Cycle through behaviors and accumulate their response
    //weighted by their priorities determined by neural network
    for (int Behaviorindex=0; Behaviorindex<Parent->BehaviorSet.size();Behaviorindex++)
    {
        //Set the user interface value
        Parent->BehaviorSet[Behaviorindex]->SetPriority(bp->Out(Behaviorindex));
        CMotorCommand tempcommand = Parent->BehaviorSet[Behaviorindex]->Apply();
        tempcommand *= bp->Out(Behaviorindex);
        resultantcommand += tempcommand;
    }
    //Devide accumulated command into the number
    //of participating behaviors
    resultantcommand /= (float) Parent->BehaviorSet.size();

    //Send final motor command
    return resultantcommand;
}
```

Figure 3.35 CBCFusionNNBackProp Class Apply Method

3.2.2.5.4 Arbiter –Neural Network

This behavior coordinator is same as the Fusion – Neural Network behavior coordinator in the sense that they both utilize Neural Network in order to determine behavior priorities. Since the trained Neural Network supplies the dynamic priorities each simulation step, these priorities can be used to determine a single behavior to use. In this case, instead of a fused motor command the motor command that is generated by the behavior having the largest priority value will be executed as the final motor command and will be passed to the wheels to guide the robot.

The code segment used to apply the dynamic prioritization policy over Arbitration technique is given in Figure 3.36.

```
CMotorCommand CBCArbiterNNBackProp::Apply()
{
    //Construct a motor command
    CMotorCommand resultantcommand;
    //Create a variable for measurement datas from sensors
    std::vector <double> MeasurementData;

    for (int RangeSensorindex=0; RangeSensorindex<Parent->BeamRangeSensors.size();RangeSensorindex++)
    {
        MeasurementData.insert(MeasurementData.end(),Parent->RangeMeasurements[RangeSensorindex]);
    }
    //feed forward neural network
    bp->ffwd(&MeasurementData[0]);


    //Cycle through behaviors and find the response
    //of the most prioritized behavior determined by the neural network
    float max_priority = 0.0f;
    for (int Behaviorindex=0; Behaviorindex<Parent->BehaviorSet.size();Behaviorindex++)
    {
        //Set the user interface value
        Parent->BehaviorSet[Behaviorindex]->SetPriority(bp->Out(Behaviorindex));
        CMotorCommand tempcommand = Parent->BehaviorSet[Behaviorindex]->Apply();
        if ( bp->Out(Behaviorindex) > max_priority )
        {
            resultantcommand = tempcommand;
            max_priority = bp->Out(Behaviorindex);
        }
    }

    //Send final motor command
    return resultantcommand;
}
```

Figure 3.36 CBCArbiterNNBackProp Class Apply Method

3.2.3 User Interface

Any software that is used for general purposes should have some sort of user interface. In order to make the implemented software environment usable and attractive, a user friendly user interface is designed. The interface is composed

of a main application frame window that encapsulates all other windows and dialogs. MDI Single Frame Window Template is used as the main application frame window. Floating and docking windows are implemented in order to let the user to modify the style of the user interface. A registration into the computers register is made in order to keep the look and the style of the user interface that has been used lastly. Dynamically created dialogs are used in the Behavior Controller Dialogs, which takes into account the current registered behavior to the selected robot.

The main window consists all the other windows and utilizes all the means to access other functional windows via menu, control bar and buttons. Main window is an OpenGL context that shows the 3D representation of the simulation. User can move in any point by moving the mouse while pressing the Ctrl button on the keyboard. Main window also gives the user a menu that is located on top. Using this menu other functionalities can be reached, layout of the main window can be manipulated, and information about the program can be reached. In Figure 3.37 the main menu is given in expanded form to show the underlying functionalities. Four main buttons are listed in the menu: File, Edit, View, and Help.

Using File menu, one can quit the program. Within the Edit menu, one can enter or change the simulation parameters. View menu, includes the following: a control bar toggle control, different windows toggle control and status bar toggle control. Help gives the general information about the program.

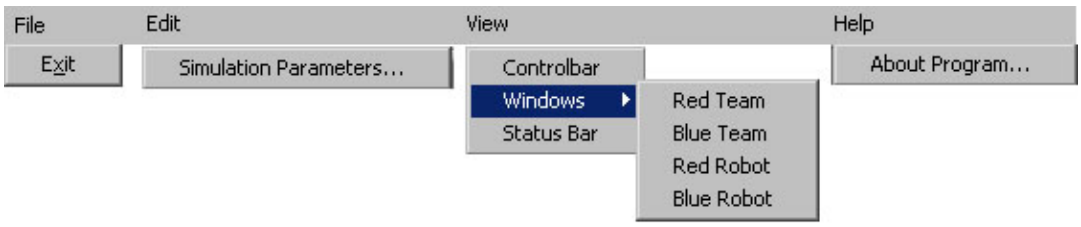

Figure 3.37 Main Window Menu

One can modify the simulation parameters through, Edit->Simulation. These parameters are the ones that affect the physical accuracy of the simulation. The available parameters are listed as follows, and given in a dialog box to user (Figure 3.38).

- Gravity in $m./\sec^2$,
- Error Reduction Parameter (ERP) value between 0 and 1,
- Constraint Force Mixing (CFM) value between 0 and 1,
- Simulation Step Size in seconds.

Gravity parameter is the constant acceleration that will be multiplied by the mass of the objects to create effective gravitational force. The simulation step size, determines the time between consecutive updates of the world, hence of each robot and of each moving object. The selection of large simulation step value will surely yield inaccurate simulations.

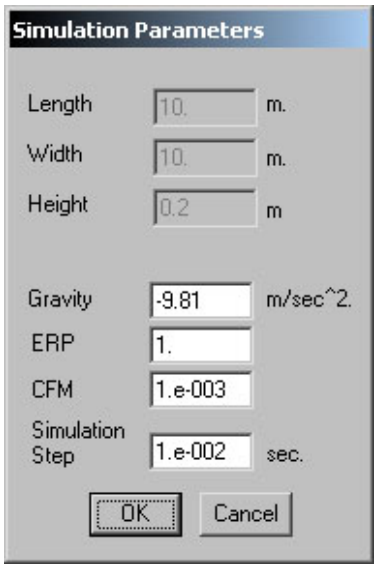

Figure 3.38 Simulation Parameters Dialog

ERP and CFM are two Open Dynamic Engine (ODE) internal variables that should be set suitably. Care should be taken when setting these variables. The default values are set suitable for most simulations, but the user should change these

parameters according to the simulation needs.

When a joint connects two bodies, those bodies are required to have certain positions and orientations relative to each other. However, it is possible for the bodies to be in positions where the joint constraints are not met. This "joint error" can happen in two ways:

1. If the user sets the position/orientation of one body without correctly setting the position/orientation of the other body.
2. During the simulation, errors can creep in that result in the bodies drifting away from their required positions.

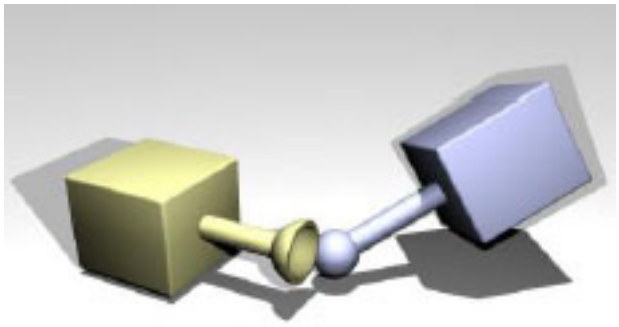

Figure 3.39 An Example of Error in a Ball and Socket Joint.

Figure 3.39 shows an example of error in a ball and socket joint where the ball and socket do not line up.

There is a mechanism to reduce joint error: during each simulation step each joint applies a special force to bring its bodies back into correct alignment. This force is controlled by the *error reduction parameter* (ERP), which has a value between 0 and 1. The ERP specifies what proportion of the joint error will be fixed during the next simulation step. If ERP=0 then no correcting force is applied and the bodies will eventually drift apart as the simulation proceeds. If ERP=1 then the simulation will attempt to fix all joint error during the next time step. However, setting ERP=1 is not recommended, as the joint error will not be completely fixed due to various internal approximations. A value of ERP=0.1 to 0.8 is recommended (0.2 is the default). A global ERP value can be set that affects most joints in the

simulation. However, some joints have local ERP values that control various aspects of the joint.

Most constraints are by nature “hard”. This means that the constraints represent conditions that are never violated. For example, the ball must always be in the socket, and the two parts of the hinge must always be lined up. In practice constraints can be violated by unintentional introduction of errors into the system, but the error reduction parameter can be set to correct these errors. Not all constraints are hard. Some “soft” constraints are designed to be violated. For example, the contact constraint that prevents colliding objects from penetrating is hard by default, so it acts as though the colliding surfaces are made of steel. But it can be made into a soft constraint to simulate softer materials, thereby allowing some natural penetration of the two objects when they are forced together. Two parameters control the distinction between hard and soft constraints. The first is the error reduction parameter (ERP) that has already been introduced. The second is the constraint force mixing (CFM) value that is described below.

Traditionally the constraint equation for every joint has the form

$$Jv = c \tag{3.3}$$

where v is a velocity vector for the bodies involved, J is a “Jacobian” matrix with one row for every degree of freedom the joint removes from the system, and c is a right hand side vector. At the next time step, a vector λ is calculated (of the same size as c) such that the forces applied to the bodies to preserve the joint constraint are

$$force = J^T \lambda \tag{3.4}$$

Open Dynamics Engine adds a new twist. ODE’s constraint equation has the form:

$$Jv = c + CFM\,\lambda \tag{3.5}$$

where CFM is a square diagonal matrix. CFM mixes the resulting constraint force in with the constraint that produces it. A nonzero (positive) value of CFM allows the original constraint equation to be violated by an amount proportional to CFM times the restoring force λ that is needed to enforce the constraint. Solving for λ gives:

$$(JM^{-1}J^T + CFM / h)\lambda = c / h \tag{3.6}$$

where h is the step size in seconds.

Thus, CFM simply adds to the diagonal of the original system matrix. Using a positive value of CFM has the additional benefit of taking the system away from any singularity and thus improving the factorization accuracy.

By adjusting the values of ERP and CFM, you can achieve various effects. For example, you can simulate spring like constraints, where the two bodies oscillate as though connected by springs. Alternatively, you can simulate more spongy constraints, without the oscillation. In fact, ERP and CFM can be selected to have the same effect as any desired spring and damper constants. If you have a spring constant k_p and damping constant k_d, then the corresponding ODE constants are:

$$\begin{aligned} ERP &= hk_p /(hk_p + k_d) \\ CFM &= 1/(hk_p + k_d) \end{aligned} \tag{3.7}$$

These values will give the same effect as a spring-and damper system simulated with implicit first order integration. Increasing CFM, especially the global CFM, can reduce the numerical errors in the simulation. If the system is near-singular, then this can increase stability. In fact, if the system is misbehaving, one of the first things to try is to increase the global CFM.

Using View->Controlbar, a control bar is toggled. Figure 3.40. shows the control bar. This control bar is used to toggle the visibilities of other windows, and serves simulation start functionality together with edit simulation environment functionality. The functionalities can also be seen on Figure 3.40.

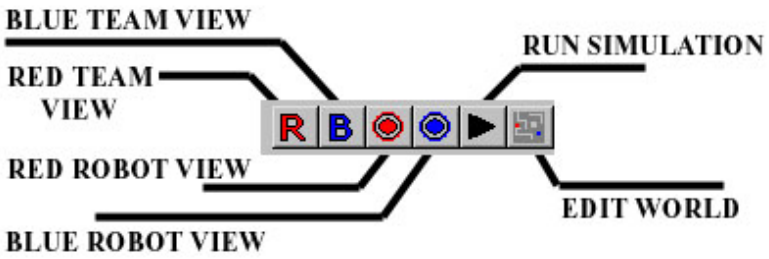

Figure 3.40 Control Bar with Associated Functions

All of the view functionalities on the control bar can also be accessed over the main menu via View->Windows.

Using “BlueTeam” view function one can access a dialog that lets the user to add robots in to the blue team. Addition of sensors, behaviors, behavior coordination schemes, and editing of the available parameters is done using this dialog.

Team Dialogs can be seen on Figure 3.41 with 2 blue robots added into the blue team, and 1 robot into the red team. Note that different sensory capabilities are added on the robots using the underneath buttons.

Any item on the tree view controls (colored rectangular areas on each team view dialog) can be right clicked. Right click event of the mouse triggers edit event of the item if exists. Right clicking on “Robot_i” item brings a new dialog that lets the user to manipulate the robot. Right clicking on any sensor also brings new dialogs that show the editable parameters for that right clicked sensor item. If there are no editable parameters, no dialog appears for editing.

In team view dialogs user manipulates the capabilities of the robot by adding sensors, behaviors, and behavior coordinators with correspondingly labeled buttons. The function of each button is clear by its label.

Note that in order to add any sensor, behavior, or behavior coordinator, a robot should be created first. Then in order to let the GUI to know where the item will be attached to (which robot), a robot should be selected priori just by clicking the robot with the mouse.

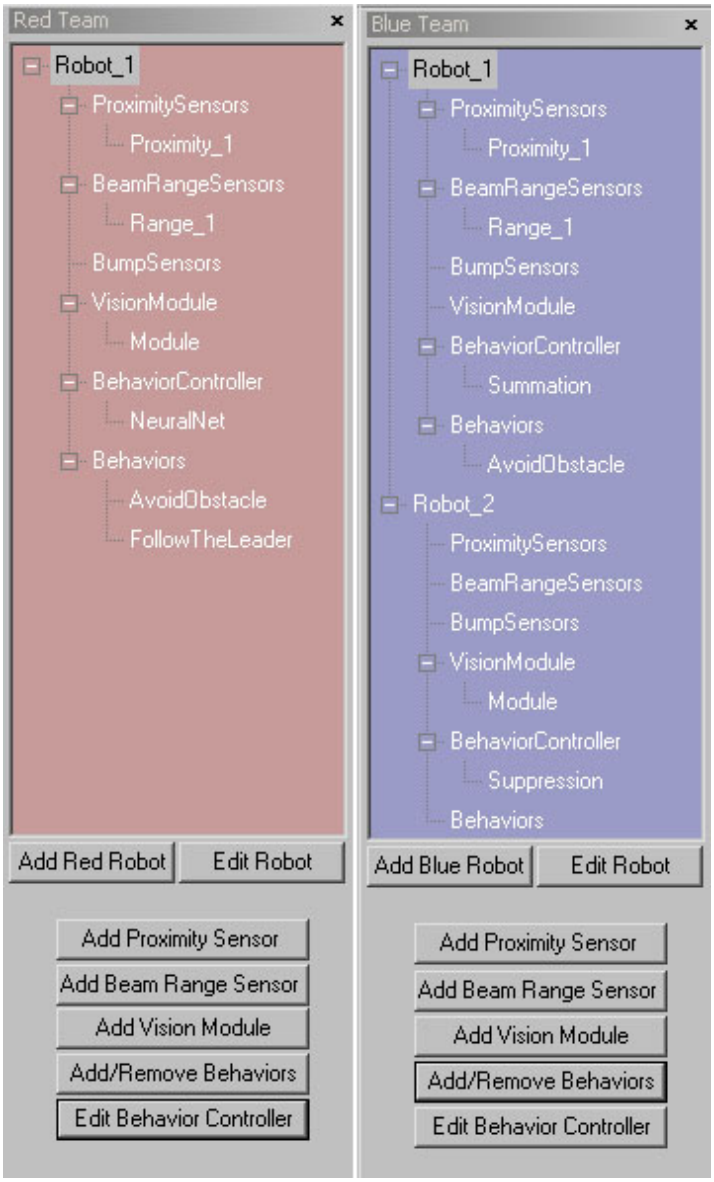

Figure 3.41 TeamView Windows

When "Add Blue/Red Robot" button is pressed on either of the team windows, a dialog appears to determine the initial parameters for the robot. These parameters should be set during initialization and once created these parameters can not be edited. Figure 3.42 shows the initialization dialog for the robot. The default values for the parameters of this dialog are as follows:

- Max Speed: 0.5 m/sec.
- Coverage: 0.5 m.

- Wheel Radius: 0.02 m.
- Chassis Mass: 0.1 kg.
- Wheel Mass: 0.05 kg.
- Full Battery: 5000 units.
- Sizes (W, L, H): (0.08 m., 0.08 m., 0.08 m.)
- Position (X, Y, Z): (0 m., 0 m., 0 m.)

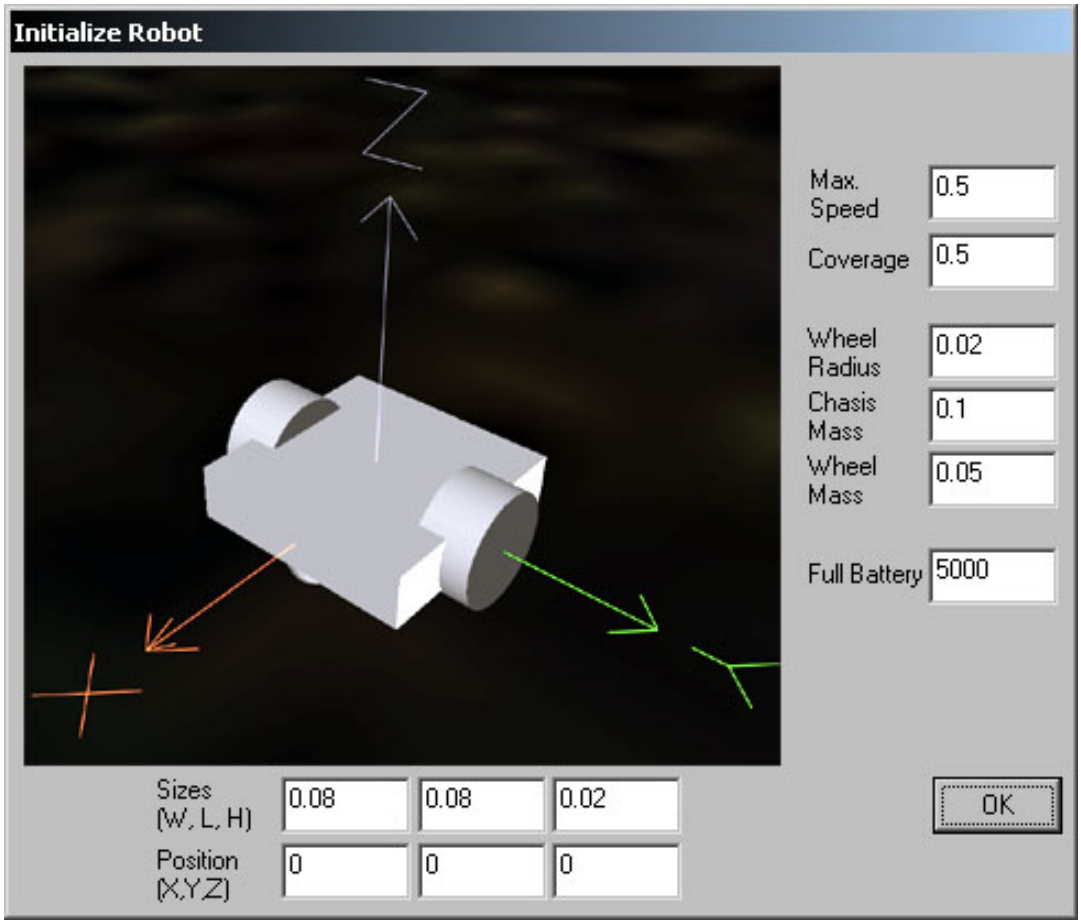

Figure 3.42 Robot Initialization Dialog

When a pre-created robot is right clicked by mouse, another dialog appears. This dialog for robot editing is given in Figure 3.43. This dialog brings the editable parameters for the robot, and is created using the current values for listed the parameters. By checking the "Apply Changes Immediately" checkbox, any change can be observed simultaneously on the main window OpenGL screen. The editable

parameters at runtime for a robot are as follows:

- Position (X, Y, Z),
- Maximum Speed,
- Coverage.

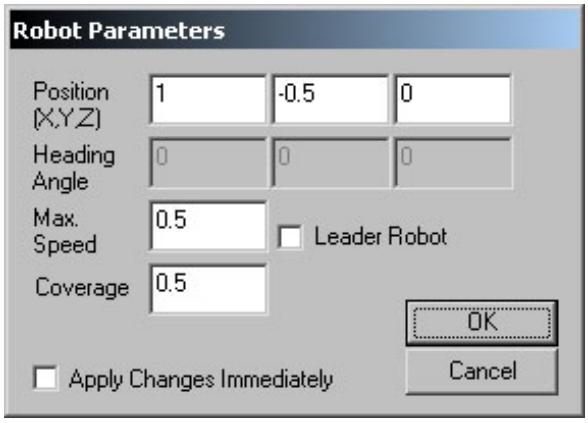

Figure 3.43 Robot Editing Dialog

When "Add Proximity Sensor" button is pressed, a dialog appears for sensor parameters. Figure 3.44 shows Proximity Sensor dialog. Some of the default values for the parameters are calculated automatically and they are as follows:

- Heading Angle: Random value between 0 and 90 degrees.
- Horizontal FOV Angle: 40 degrees.
- Vertical FOV Angle: 5 degrees.
- Maximum Range: 0.5 m.
- Offset Along X: Robot's Half Width in meters.
- Offset Along X: 0 m.
- Offset Along Z: Height of the top surface of the Robot chassis in meters.
- Energy Cost per Sec: 1 unit.

Note that Proximity Sensors are located with respect to the robot frame. Any offset values can be set for the sensor. As the result sensor may necessarily not physically be on the robot, but still sensor will move together with the robot as robot moves, keeping its' offset values fixed with respect to the moving robot frame.

When a Proximity Sensor (Proximity_i) is right clicked by mouse, the same dialog as in Figure 3.44 appears. This time the values for the parameters are set automatically to show the current values for that sensor.

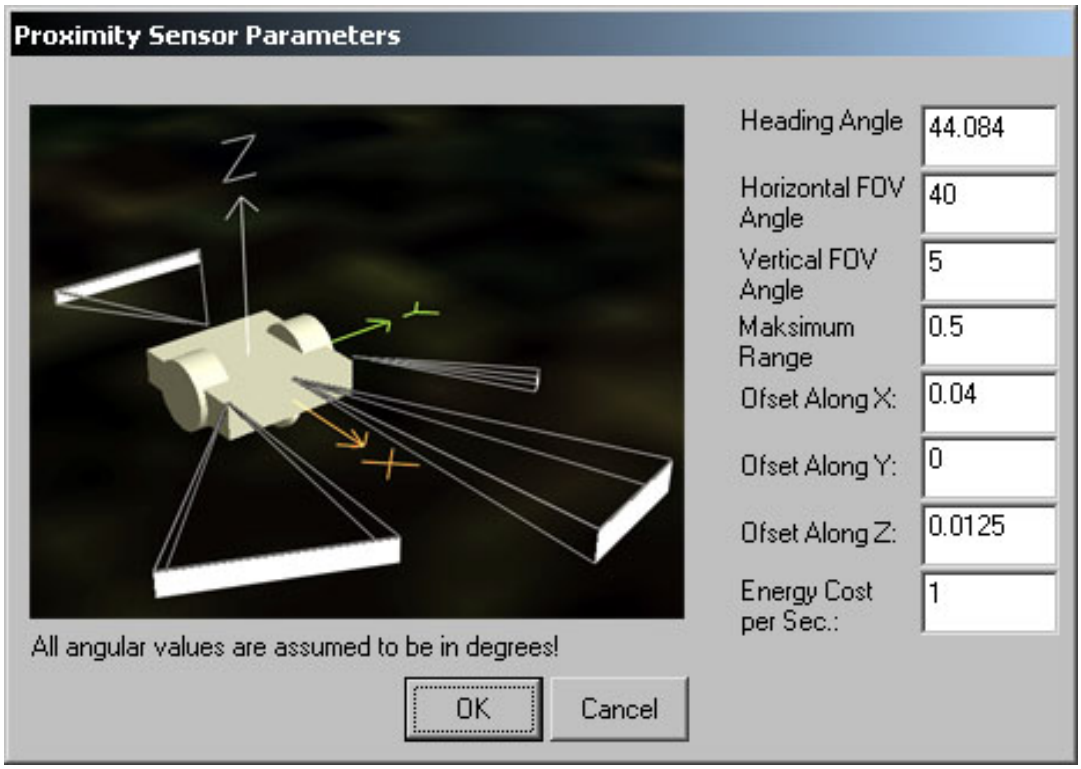

Figure 3.44 Proximity Sensor Dialog

When "Add Beam Range Sensor" button is pressed, a dialog shown in the Figure 3.45 appears. This dialog box is used to set the initial values for the Range Sensor. The default parameters are as follows:

- Pitch: 0 degrees
- Yaw: Random value between 0 and 90 degrees.
- Maximum Range: 1 m.
- Offset Along X: Robot's Half Width in meters.
- Offset Along X: 0 m.
- Offset Along Z: Height of the top surface of the Robot chassis in meters.
- Error: 0 m.
- Energy Cost per Second: 1 units

When a sensor of this type is right clicked for editing, same dialog box appears. The editing dialog reflects the current settings for that sensor.

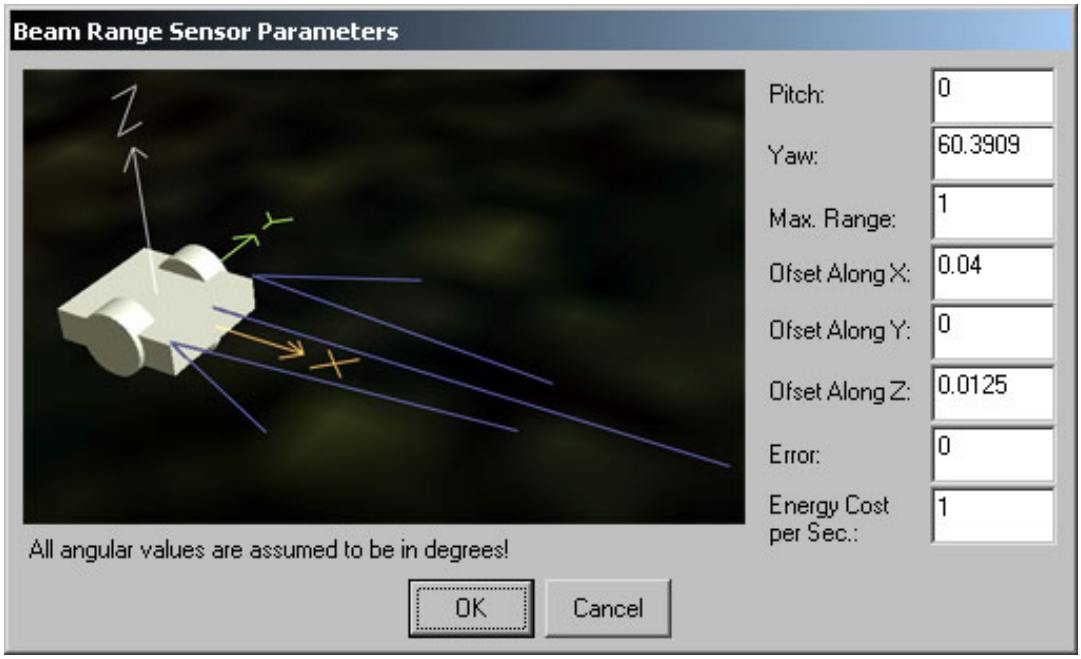

Figure 3.45 Beam/Range Sensor Dialog

When "Add Vision Module" button is pressed while a robot is selected, a dialog box lets the user to specify the parameters for the vision module. This dialog box is shown in Figure 3.46.

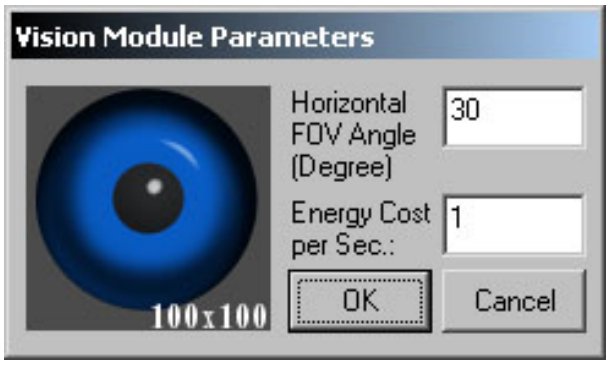

Figure 3.46 Vision Module Dialog

Using the Vision Module dialog, following parameters are shown as default and can be set:

- Horizontal FOV Angle: 30 degrees.
- Energy Cost per Second: 1 units

These parameters can be changed when a vision module is right clicked using the above dialog.

To add new behaviors to the robot or to remove preloaded behaviors from the robot, "Add/Remove Behaviors" buttons are used. When clicked "Behavior Dialog" appears as it is given in Figure 3.47.

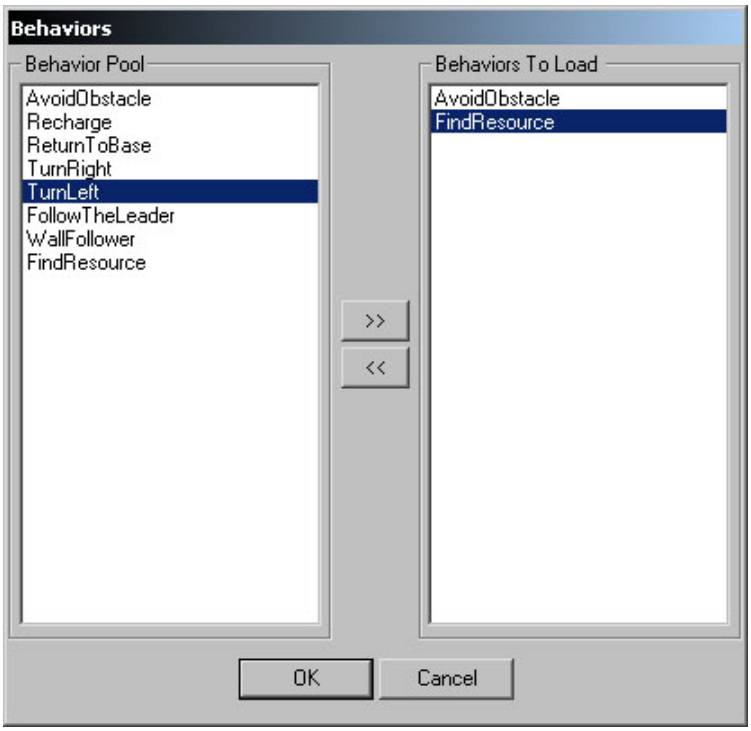

Figure 3.47 Behavior Dialog

Using Behavior Dialog, behaviors from behavior pool (left hand side) are selected via single click of mouse and moved to the loading bin using the appropriate arrow (>>) button. The behaviors can be selected one by one from the pool and should be moved to the loading in one by one. No behaviors can be loaded more than one

instance. If this button is pressed when a robot with previously loaded behaviors, “Behaviors To Load” area lists the behaviors loaded on the robot and does not let the user to reload the same behavior on the robot again. To remove previously loaded behaviors of the robot, behaviors to be unloaded are selected one by one with a single click from the right hand side area. Then appropriate arrow button (<<) is used to move the selected behavior to the behavior pool.

The changes are reflected to the robot when “OK” button is pressed. If “Cancel” button is pressed, no changes are reflected to the selected robot. The changes in the loaded behaviors can be observed on the team tree view (Figure 3.41). Behaviors leaf shows the currently loaded behaviors. On the tree view, if any of these loaded behaviors are right clicked a dialog box showing the priority setting for that behavior appears, Figure 3.48.

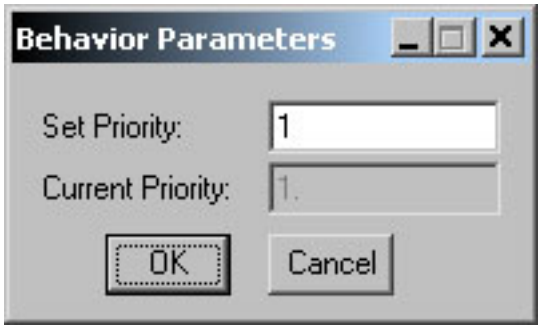

Figure 3.48 Behavior Priority Showing/Editing Dialog

This dialog is used to set the priorities of the behaviors. The priorities are used in Behavior Coordination Schemes and these floating-point valued parameters can be regarded as the weights on the robot. The dialog is also used to show the priority value in “Current Priority” field. When Neural Network coordination scheme is assigned to the robot, automatically calculated the priorities for the behaviors according to the sensor data can be seen using this dialog.

To change the behavior coordination scheme, “Edit Behavior Controller” button is used when a robot is selected. This event gives the options for the behavior

coordination scheme. The currently developed behavior controllers are as follows:

- Arbiter-Suppression
- Fusion –Vector Summation
- Fusion – Neural Network
- Arbiter – Neural Network

The controller to be loaded is selected by means of mouse and when "OK" button is pressed the controller for the robot is updated and listed in the tree view under Behavior Controller leaf. If one of the controllers utilizing Neural Network is selected a further dialog box (Figure 3.49) appears before the controller is loaded on the robot.

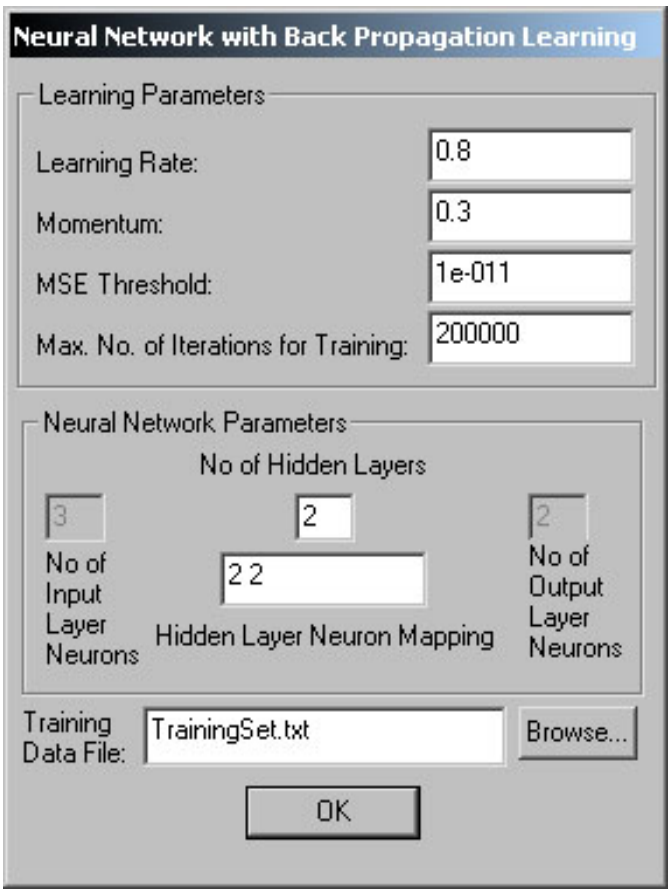

Figure 3.49 Neural Network Dialog

The Neural Network dialog lets the user to specify the learning parameters of the network, to be used in Bayesian Learning algorithm. The default learning parameters are as follows:

- Learning Rate: 0 .8 (should be between 0 and 1)
- Momentum: 0.3 (should be between 0 and 1)
- Mean Square Error Threshold: 1e-11 (main stop criteria)
- Maximum Number of Iterations for Training: 200000 (alternative stop criteria)

The dialog also lets the user to specify the topology of the network by selecting the number of hidden layers to be used, and the number of neurons in each of these hidden layers. The number of neurons for the input and the output layer is set automatically by the program, and is shown as non-editable just for remark. Note that the number of the input layer neurons is equal to the Beam/Range Sensor currently on the robot, whereas the number of the output layer neurons is equal to the number of the behaviors on the robot. A training file including the training data should be determined using “Browse” utility. When pressed ”OK” corresponding neural network is formed, trained and loaded on to the robot. The time required for these steps depends on the network setting and the training data size, but should generally be in seconds scale.

The loaded Neural Network controller can be seen on the tree view under the corresponding robots’ “Behavior Controller” leaf. When this item is right clicked, the network is cleared and should be set again.

One should care attention not to define a wrong formatted training data. The number of the floating-point valued variables per line in the training data file should match the total number of input and output neurons. The user is recommended to form a training data according to the experiment that is planned to be conducted, before starting the program.

CHAPTER 4

RESULTS

The software environment is constructed to evaluate the effectiveness of behavior-based control. For this task, a detailed simulation is conducted on navigating a single two-wheel differential-drive mobile robot in an unstructured environment, using multi-behavior coordination.

4.1 SINGLE ROBOT, MULTIPLE BEHAVIOR SIMULATION

In this simulation, one robot is registered to the environment. This robot is equipped with two beam range sensors. These sensors are located to be 30^0 apart from each other and symmetrical about the robot heading axis, 15^0 and -15^0 respectively. A vision module with 30^0 of FOV angle is also loaded to the differential drive robot. The behaviors that map the readings from these sensors into motor actions are selected to be “Avoid Obstacle” and “Find Resource”. The behavior controller is selected to be “Fusion - Neural Network” in one simulation and “Arbitration - Neural Network” in the other simulation.

All simulations are conducted in the same environment. The environment is chosen to have 10 m. width and 10 m. length. The environment is surrounded by non-movable static walls. Four resources are registered into the environment. The robot is located to be at the origin of the environment, (0, 0, 0) in (x, y, z) coordinates.

The simulations are terminated at the time all the resources are collected by the robot. These simulations are considered successful. Figure 4.1 shows the initial conditions for the simulations.

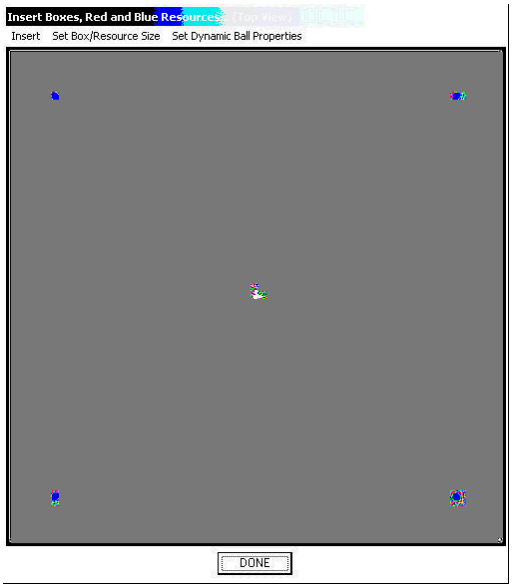

Figure 4.1 Initial conditions (top view)

The robot is initially located at (0,0). There are four resources located at (4.0, 4.0), (-4.0, 4.0), (-4.0, -4.0) and (4.0, -4.0).

The robot is loaded with two behaviors. The priorities of these behaviors are evaluated by a neural network. Neural network is trained using a simple training data file using offline training, back-propagation algorithm. The evaluated priorities are then used in coordination scheme. For comparison of their performances, two types of coordination schemes are investigated: "Arbitration - Suppression" and "Fusion – Vector Summation".

The simulations are conducted from same initial conditions. The neural network learning parameters are also kept constant. The parameters are as follows:

Learning Rate	:	0.8
Momentum	:	0.3
MSE Threshold	:	1e-11
Max. No. of Iterations for Training	:	200000

The training, for all cases is done with the same training data set.. The training data is as follows:

1	1	0	1
0.8	1	1	0
0.6	1	1	0
0.2	1	1	0
1	0.8	1	0
1	0.6	1	0
1	0.2	1	0
0.2	0.2	0.3	0
0.5	0.5	0.6	0

The training results for each case are given in Table 4-1.

Table 4-1 Training Results for Case 1 Through Case 4

	Arbitration		**Fusion**	
	No. of Iterations	**MSE Achieved**	**No. of Iterations**	**MSE Achieved**
Case 1	63149	0.000000000	200000	0.000300313
Case 2	200000	0.000034753	200000	0.000037821
Case 3	200000	0.123801203	200000	0.123801695
Case 4	200000	0.123800440	200000	0.000004546

The different topologies of the neural network differs the simulations from each other. Since the number of range sensors on the robot determines the input layer of the neural network and the number of behaviors determines the output layer of the neural network, the only variable in the network topology that can be tuned is the number of hidden layers and the neurons in these layers.

Four cases are investigated for comparison in evaluating hidden layer topology dependency, and they are summarized in Table 4-2:

Table 4-2 Hidden Layer Topology for Case 1 Through Case 4

	No. of Hidden Layers	**Hidden Layer Neuron Mapping**
Case 1	2	2 – 2
Case 2	2	4 – 4
Case 3	4	2 - 2 - 2 - 2
Case 4	4	4 - 4 - 4 - 4

The column related with the hidden layer neuron mapping, indicates the number of neurons in each hidden layer.

For different hidden layer topologies, the performances of arbitration and fusion schemes are compared. The measure for success is collecting all resources present within the field. The total distance traveled to collect all the resources and the total time passed elapsed during the simulation is considered as a performance measure. Figure 4.2 to Figure 4.5 shows some figures from the simulations. Table 4-3 shows the result data extracted from the simulations in tabular form. Considering the data displayed in Table 4-3, fusion and arbitration schemes show different performances under different neural network topologies. The data in Table 4-3 show the mean values obtained after several times of runs for each simulation. Each case is simulated 5 times and the mean values are recorded. Clearly it is experimented that fusion scheme is found to be successful where as the arbitration scheme is found to be unsuccessful in nearly half of the experiments. Comparing the arbitration and fusion performances under the condition that both are found to be successful, when fusion scheme is used the distance traveled is found to be nearly %50 lower then the distance found when arbitration is used. Considering the velocities of the robots, when the mission is successfully completed, fusion scheme is found to be slower. Considering the unsuccessful experiments in arbitration schemes, it is found that there is a trade off between mission completion speed and success.

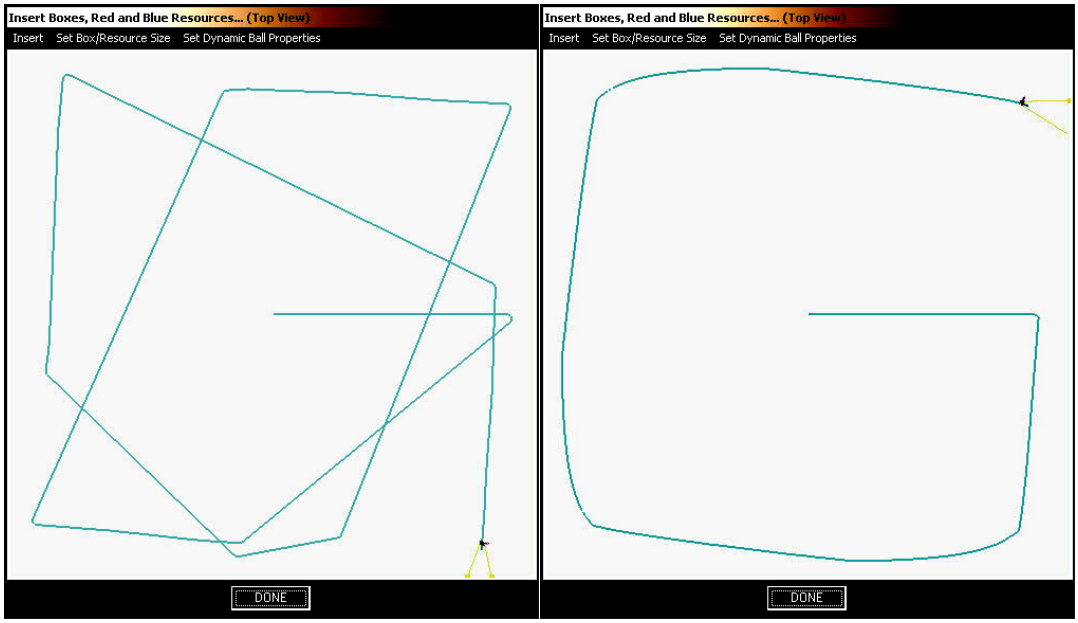

Figure 4.2 Case 1: (Left Image) Arbitration vs. Fusion (Right Image)

Figure 4.3 Case 2: (Left Image) Arbitration vs. Fusion (Right Image)

Figure 4.4 Case 3: (Left Image) Arbitration vs. Fusion (Right Image)

Figure 4.5 Case 4: (Left Image) Arbitration vs. Fusion (Right Image)

Table 4-3 Case 1-4 Comparison for Arbitration and Fusion

	Arbitration			Fusion		
	Distance Traveled (m.)	Total Time (sec.)	Success / Failure	Distance Traveled (m.)	Total Time (sec.)	Success / Failure
Case 1	64.79	170.50	success	32.98	293.70	success
Case 2	72.57	189.00	success	32.87	170.40	success
Case 3	171.95	279.70	failure	41.54	220.40	success
Case 4	152.20	249.10	failure	41.12	546.41	success

Another case study is carried out for evaluating the performances of the schemes under occlusion of obstacles between the robot and the target resource. Figure 4.6 and Figure 4.7 shows the two cases (top views). Case 5 has one obstacle and Case 6 has two obstacles between the robot and the resource.

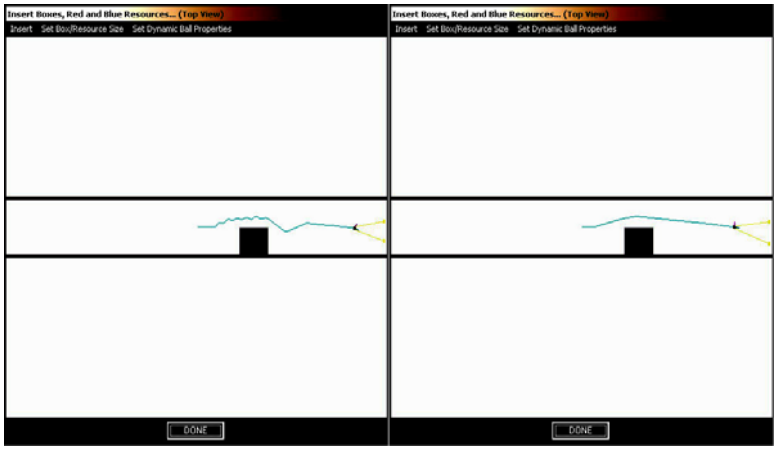

Figure 4.6 Case5: (Left Image) Arbitration vs. Fusion (Right Image)

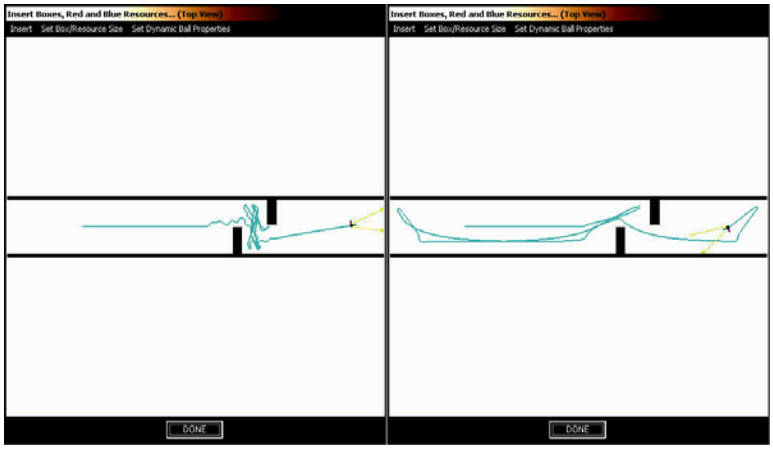

Figure 4.7 Case6: (Left Image) Arbitration vs. Fusion (Right Image)

Both cases are found to be successful as shown in, Table 4-5. Investigating the results, reveals that the fusion cases yield a shorter collision-free path between the robot and the resource. It is also observed that the arbitration scheme can be trapped between one behavior output and another which conflicts each other. Training results of the neural network for Case 5 and Case6 are given in Table 4-4.

Table 4-4 Case 5 and Case 6 Training Results

	Arbitration		**Fusion**	
	No. of Iterations	**Achieved MSE**	**No. of Iterations**	**Achieved MSE**
Case 5	200000	0.000543424	200000	0.000143434
Case 6	200000	0.000023683	200000	0.000015438

Table 4-5 Case 5 and Case 6 Comparison for Arbitration and Fusion

	Arbitration			Fusion		
	Distance Traveled (m.)	Total Time (sec.)	Success / Failure	Distance Traveled (m.)	Total Time (sec.)	Success / Failure
Case 5	4.492422	15.300022	success	4.069355	50.999794	success
Case 6	34.262062	89.599205	success	26.398300	155.700302	success

The comparison of arbitration and fusion schemes has also been investigated in highly cluttered environment. This experiment is conducted for a different sensor configuration. The change in this configuration is made for the relative horizontal angles of range sensors. Two sensors in this case are symmetrically located to be 35 and -35 degrees with respect to robot heading axis.

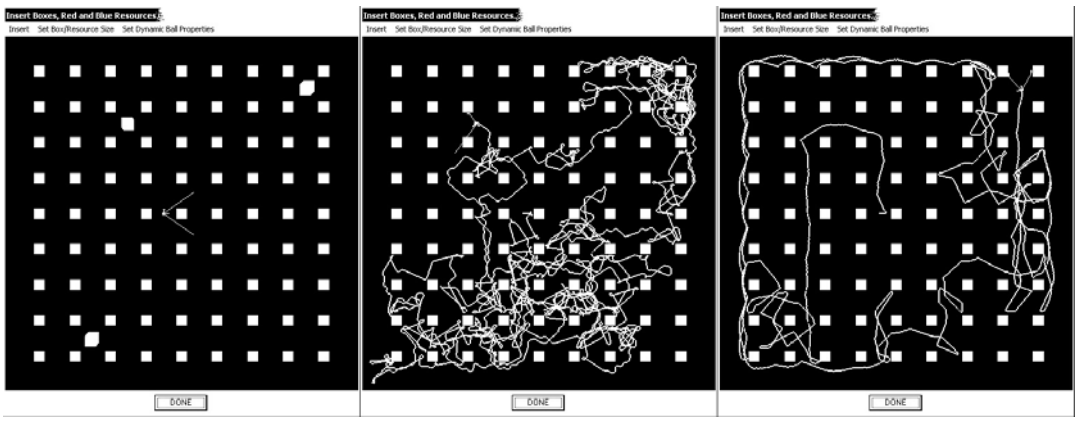

Figure 4.8 Highly cluttered environment simulation: Initial state (Left Image), Arbitration simulation final (Middle Image), Fusion simulation final (Right Image)

As Table 4-6 displays that both simulations are found to be successful in collecting the three resources that are disobeying the homogeneity of the obstacles. Comparison of the results shows that fusion scheme is capable of collecting the resources in a shorter travel distance that the arbitration scheme is under the same circumstances.

Table 4-6 Case 7 Comparison for Arbitration and Fusion

	Arbitration			Fusion		
	Distance Traveled (m.)	Total Time (sec.)	Success / Failure	Distance Traveled (m.)	Total Time (sec.)	Success / Failure
Case 7	285.22	998.40	success	134.13	1111.57	success

4.2 MULTIPLE ROBOTS WITHIN A GROUP SIMULATION

Multiple robot teams are commonly researched in the robotic field. The topic, itself offers many research areas, such as group formation, group coordination. The topic can be investigated in two main research areas, namely centralized and decentralized control.

Centralized control can be summarized as a control type where information or knowledge is accumulated on a common knowledge base where all members of the group can access any information obtained via other members.

Decentralized control runs on a limited information sharing strategy, so that individuals can access only local information about their group. Many researches are conducted on centralized and decentralized control schemes. From the behavior-based robot control point of view, some discussion and applications can be investigated in [6, 8, 9, 11, 12, 15, 16, 22, 28, 32, 37, 38, 40, 48, and 51].

A simulation has been conducted over the group behavior of a number of robots forming a robot team. One of the robots is assigned to be the leader robot. This robot is equipped with several sensors, including beam range sensors in different configurations, vision sensor, and volume proximity sensors. The rest of the team is left as default and are not loaded with any sensory equipment.

All robots including the leader robot are loaded with “Follow the Leader” behavior. The leader robot is also loaded with “Avoid Obstacle” and “Recharge” behaviors. The robots all located within the effective communication area with the leader robot are influenced by the presence of the leader and obeyed their “Follow the Leader” behavior. As a result, all simple and sensorless robots gather together and follow the leader successfully. The group shows in the simulation a successful group behavior.

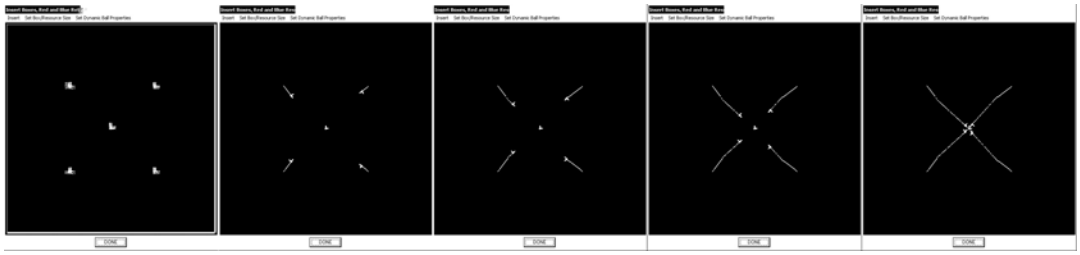

Figure 4.9 Robot Group Gathering in Static Leader Case

Figure 4.9 shows the initial status and locations of the robot group (left most image). The consecutive images are taken at consecutive time steps to display gathering together around the leader. The lines in the figures show the accumulated trajectories of each robot.

Another simulation is conducted when the leader has a behavior loaded. Figure 4.10 shows the simulation history from top view when the leader is loaded with “find resource behavior”. The group can clearly be seen to approach the resource altogether. Right most point in the initial case is the resource where as the left most point is the leader robot.

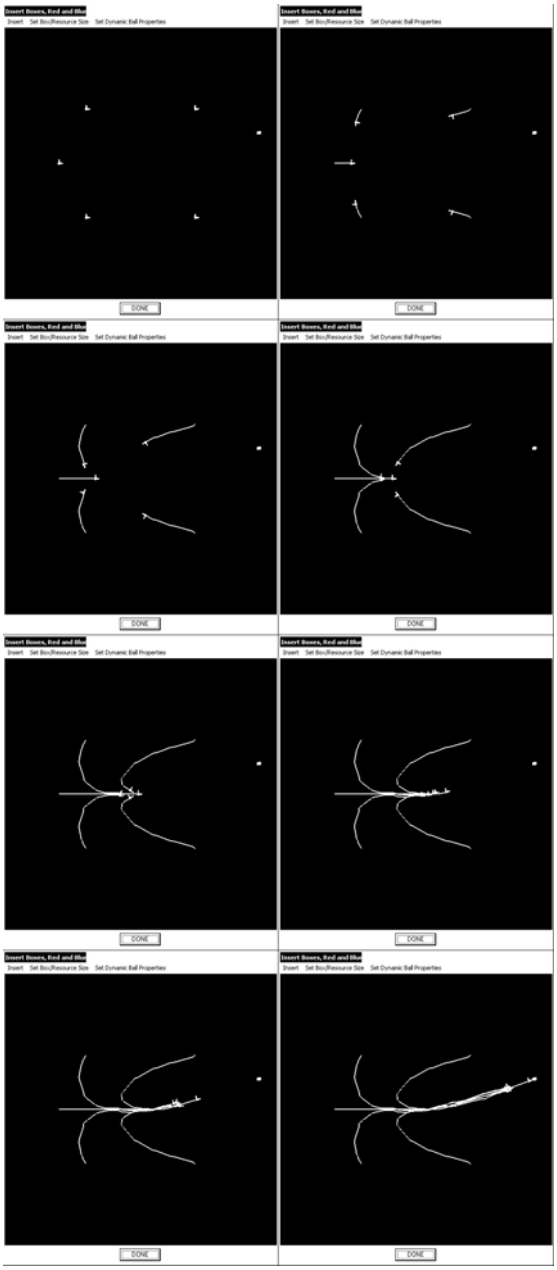

Figure 4.10 Step by Step Group Approach to the Resource

CHAPTER 5

SUMMARY, CONCLUSIONS AND FUTURE WORK

To point out the importance and benefits of behavior based robotics, some parts of this thesis is prepared to present different aspects.

In the study, a simulation environment has been developed in an high level programming language that can easily be expanded to different applications. Various simulations have been carried out to investigate the realizabilility of behavioral control of robots which functions in an unstructured environment. Robot behaviors in these simulations have been implemented in two different forms; arbitration-suppression and fusion-vector summation schemes.
These schemes thus, have been compared effectively based on certain criteria set forth within the study. These criteria are namely;

- the number of hidden layers and number of neurons involved within the learning scheme,
- time taken to reach a reasonable and acceptable pose,
- total distance traveled in reaching to the above referred final pose,
- accomplishment of the task,

We can summarize the findings out of this study as;

- Fusion scheme successfully accomplishes the task in all simulations, arbitration scheme fails sometime or it's successful accomplishments may take longer in time and traveled distance,
- Increasing the number of hidden layers and number of neurons in the learning process enhances the performance of the fusion algorithm but such

a positive impact is not seen for the arbitration scheme,

- The software developed in this study for the simulation environment depending on open-source libraries proves to be useful, besides proves to be quite flexible to construct satisfactory software for demanding applications.

During the literature survey of this thesis work, no software capable of employing such detail and showing such extendibility property is found. The final product of this thesis work shows the following distinctive properties:

- Parallel evaluation of sequentially executed behaviors using different control architectures as behavior control mechanism.
- Integrated OpenCV functionality lets the user not only to have the vision module output, but also process it within the software with no additional effort.
- Integrated ODE makes the software comparable with other physics based dynamic simulation environments.
- Integrated Fuzzy Logic Library lets the user not only to define fuzzy control parameters but also visualize them.
- Object-oriented programming approach makes the software easy to understand and easy to add additional properties for developers.

Examining the aims of this thesis work and the final software release, shows that the targeted points to be covered is reached, successfully. The user interface built over these core modules lets the user to interact with the experimental environment, manipulate it by adding robots equipped with different sensory configuration.

It should be useful in the future to add different properties to the simulation environment. At the present state, user opens the source codes and adds new header and implementation files for a new behavior. This can be enhanced by means of a run-time compiler support to the executable binaries. Such a property gives user the freedom not to deal with the problems of compiler variations or versions. This

property is also evaluated to reduce the mean time to develop some behavior and add them to the desired robot agent. Together with the codes running behind, the user interfaces can also be upgraded. However, these works can be specified as purely a computer programming experience. In order to extend the scope to robotics and electronics, different algorithms of navigation, behavior fusion mechanisms may also be developed.

The current release is operated under Windows operating systems. This can limit the possible user profile. In order to extend the targeted user profile, an operation system independent version can be developed.

REFERENCES

[1] Mataric, M., J.,"Integration of Representation into Goal-Driven Behavior-Based Robots", IEEE Transactions on Robotics and Automation, vol. 8, no. 3, June 1992.

[2] Connell, J. H., "SSS: A Hybrid Architecture Applied to Robot Navigation", Proceedings of the 1992 IEEE International Conference on Robotics and Automation, May 1992.

[3] Gat, E., Desai, R., Ivlev, R., Loch, J. and Miller, D. P., "Behavior Control for Robotic Exploration of Planetary Surfaces", IEEE Transactions on Robotics and Automation, vol. 10, no. 4, August 1994.

[4] Asama, H., Sato, M., Bogoni, L., Kaetsu, H., Matsumoto, A., Endo, I., "Development of an Omni-Directional Mobile Robot with 3 DOF Decoupling Drive Mechanism", IEEE Interantional Conference on Robotics and Automation 0-78-1965-6/95.

[5] Parker, L. E., "Task Oriented Multi-Robot Learning in Behavior-Based Systems", IEEE Proc. IROS'96 07803-3213-X/96.

[6] Yamaguchi, H., "A Cooperative Hunting Behavior by Mobile Robot Troops", Proceedings of the 1998 IEEE International Conference on Robotics and Automation, May 1998.

[7] Watanabe, K., "Control of an Omni directional Mobile Robot", 1998 Second International Conference on Knowledge-Based Intelligent Electronic Systems, April 1998.

[8] Balch, T., Arkin, R. C., “Behavior-Based Formation Control for Multirobot Teams”, IEEE Transactions on Robotics and Automation, vol. 14, no. 6, December 1998.

[9] Hirata, Y., Kosuge, K., Asama, H., Kaetsu, H. and Kawabata, K., “Decentralized Control of Mobile Robots in Coordination”, Proceedings of the 1999 IEEE International Conference on Control Applications, August 1999.

[10] Fukao, T., Nakagawa, H. and Adachi, N., “Adaptive Tracking Control of a Nonholonomic Mobile Robot”, IEEE Transactions on Robotics and Automation, vol. 16, no. 5, October 2000.

[11] Hirata, Y., Kosuge, K., Asama, H., Kaetsu, H. and Kawabata, K., “Coordinated Transportation of a Single Object by Multiple Mobile Robots without Position Information of Each Robot”, Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems, May 2000.

[12] Hirata, Y., Takagi, T., Kosuge, K., Asama, H., Kaetsu, H. and Kawabata, K., “Map-based Control of Distributed Robot Helpers for Transporting an Object in Cooperation with a Human”, Proceedings of the 2001 IEEE International Conference on Robotics and Automation, May 2001.

[13] Arkin, R. C., “Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior”, IEEE CH2413-3/87.

[14] Matsikis, A., Schulte, F., Broicher, F. and Krais, K. F., “A Behaviour Coordination Manager for a Mobile Manipulator”, Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2003.

[15] Montesano, L., Montano, L., “Identification of Moving Objects by a Team of Robots Based on Kinematic Information”, Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2003.

[16] Mitsunaga, N., Izumi, T., Asada, M., “Cooperative Behavior Based on a Subjective Map with Shared Information in a Dynamic Environment”, Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2003.

[17] Brooks, R. A., “Behavior-Based Humanoid Robotics”, IEEE Proc. IROS’96.

[18] Mataric, M. J., “Reinforcement Learning in the Multi-Robot Domain”, Autonomous Robots 4,73-83, 1997.

[19] Brooks, R. A., “A Robust Layered Control System for a Mobile Robot”, IEEE Journal of Robotics and Automation, vol. ra-2, no. 1, March 1986.

[20] Gu, D., Hu, H, Spacek, L., “Learning Fuzzy Logic Controller for Reactive Robot Behaviours”, Proceedings of the 2003 IEEE/ASME Intl. Conference on Advance Intelligent Mechatronics (AIM 2003).

[21] Dassanayake, P., Watanabe, K., Kiguchi, K., Izumi, K., “Fuzzy Behavior-Based Motion Planning for the PUMA Robot”, Proceedings of the 2000 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, May 2000.

[22] Thrun, S., Burgard, W., Fox, D., “A Real-Time Algorithm for Mobile Robot Mapping with Applications to Multi-Robot and 3D Mapping”, Best Conference Paper Award, IEEE International Conference on Robotics and Automation, April 2000.

[23] Tunstel, E., “Mobile Robot Autonomy via Hierarchical Fuzzy Behavior Control”, Manuscript appears in Proc. of the 6th Intl. Symp. On Robotics and Manuf., 2nd World Automation Congress, May 1996.

[24] Ojala, J., Inoue, K., Sasaki K., and Takano, M., “Interactive Graphical Mobile Robot Programming”, IEEE/RSJ International Workshop on Intelligent Robots and Systems IROS’91, November 1991.

[25] Anderson, T. L., Donath, M., “Autonomous Robots and Emergent Behavior: A Set of Primitive Behaviors for Mobile Robot Control”, IEEE International Workshop on Intelligent Robots and Systems IROS’90.

[26] Beom, H. R., Cho, H. S., “A Sensor-Based Navigation for a Mobile Robot Using Fuzzy Logic and Reinforcement Learning”, IEEE Transactions on Systems, Man, and Cybernetics, vol. 25, no.3, March 1995.

[27] Li, W., “Perception-Action Behavior Control of a Mobile Robot in Uncertain Environments Using Fuzzy Logic”, Department of Computer Science and Technology National Laboratory of Intelligent Technology and Systems, Tsinghua University, Beijing, China.

[28] Shibata, T., Ohkawa, K., Tanie, K., “Spontaneous Behavior of Robots for Cooperation - Emotionally Intelligent Robot System”, Proceedings of the 1996 IEEE International Conference on Robotics and Automation, April 1996.

[29] Fukuda, T., Iritani, G., “Construction Mechanism of Behavior with Cooperation”, 0-8186-7108-4/95.

[30] Uchibe, E., Asada, M., Hosoda, K., "Behavior Coordination for a Mobile Robot Using Modular Reinforcement Learning", IEEE Proc. IROS'96, 0-7803-3213-X/96.

[31] Murphy, R. R., Hawkins, D. K., "Behavioral Speed Control Based on Tactical Information", IEEE Proc. IROS'96, 0-7803-3213-X/96.

[32] Wnag, Z., Nakano, E., Matsukawa, T., "Realizing Cooperative Object Manipulation using Multiple Behavior-Based Robots", IEEE Proc. IROS'96, 0-7803-3213-X/96.

[33] Lee, W., Hallam, J., Lund, H. H., "Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots", IEEE 0-7803-3949-5/97.

[34] Mali, A. D., "Tradeoffs in Making the Behavior-Based Robotic Systems Goal-Directed", Proceedings of the 1998 IEEE International Conference on Robotics & Automation, May 1998.

[35] Uchibe, E., Asada, M., Hosoda, K., "Cooperative Behavior Acquisition in Multi Mobile Robots Environment by Reinforcement Learning Based on State Vector Estimation", Proceedings of the 1998 IEEE International Conference on Robotics & Automation, May 1998.

[36] Brussel, H. V., Moreas, R., Zaatri, A., Nuttin, M., "A Behaviour-Based Blackboard Architecture for Mobile Robots", IEEE 0-7803-4503-7/98.

[37] Chen, T. M., Luo, R. C., "Integrated Multi-Behavior Mobile Robot Navigation Using Decentralized Control", Proceedings of the 1998 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 1998.

[38] Fukuda, T., Mizoguchi, H., Sekiyama, K., Arai, F., "Group Behavior Control for MARS (Micro Autonomous Robotic System)", Proceedings of the 1999 IEEE International Conference on Robotics & Automation, May 1999.

[39] Hasegawa, Y., Ito, Y., Fukuda, T., "Behavior Coordination and its Modification on Brachiation-type Mobile Robot", Proceedings of the 2000 IEEE International Conference on Robotics & Automation, April 2000.

[40] Balch, T., Hybinette, M., "Behavior-Based Coordination of Large-Scale Robot Formations" IEEE 0-7695-0625-9/00.

[41] Kawabata, K., Ishikawa, T., Fujii, T., Asama, H., Endo, I., "A Behavior Learning Method of a Mobile Robot using View Information", Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems, May 2000.

[42] Xu, L., Sabatto, S., Sekmen, A., "Development of Intelligent Behaviors for a Mobile Robot", IEEE 0-7803-6661-1/01.

[43] Suzuki, M., Scholl, K., Dillmann, R., "A Method for Learning Complex and Dexterous Behaviors through Knowledge Array Network", Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, November 2001.

[44] Seraji, H., Howard, A., "Behavior-Based Robot Navigation on Challenging Terrain: A Fuzzy Logic Approach", IEEE Transactions on Robotics and Automation, vol. 18, no 3, June 2002.

[45] Polvichai, J., Khosla, P., “An Evolutionary Behavior Programming System with Dynamic Networks for Mobile Robots in Dynamic Environments”, Proceedings of the 2002 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2002.

[46] Suzuki, M., “A Study on Quantitative Evaluation of Robot Behaviors”, SICE August 2002.

[47] Li, H., Yang, S. X., “A Behavior-Based Mobile Robot With a Visual Landmark-Recognition System”, IEEE/ASME Transactions on Mechatronics, vol. 8, no. 3, September 2003.

[48] D’Angelo, A., Ota, J., “How Intelligent Behavior Can Emerge from a Group of Roboticles Moving Around”, Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2003.

[49] Gu, D., Hu, H., “Teaching Robots to Coordinate its Behaviours”, Proceedings of the 2004 IEEE International Conference on Robotics & Automation, April 2004.

[50] Vadakkepat, P., Miin, O. C., Peng, X., Lee, T. H., “Fuzzy Behavior-Based Control of Mobile Robots”, IEEE Transactions on Fuzzy Systems, vol. 12, no. 4, August 2004.

[51] Parker, L. E., Kannan, B., Tang F., Bailey M., “Tightly-Coupled Navigation Assistance in Heterogeneous Multi-Robot Teams”, Proceedings of the 2004 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2004.

[52] Jing, X., Tan, L., Wang, C., “Behavior Dynamics of Collision-Avoidance in Motion Planning of Mobile Robots”, Proceedings of the 2004 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, October 2004.

[53] Li; S., Li, Y., “Compound Zero Behavior-Based Autonomous Mobile Robot Obstacle Avoidance Algorithm in Unknown Environment”, Manuscript received October 2005.

[54] Carreras, M., Yuh, J. and Batile, J., “An Hybrid Methodology for RL-based Behavior Coordination in a Target Following Mission with an UAV”, MTS 0-933957-28-9.

[55] Thongchai, S. et al, “Sonar Behavior-Based Fuzzy Control for a Mobile Robot“, IEEE 0-7803-6583-6/00, June 2000.

[56] www.k-team.com

[57] Iske, B., Rückert, U., “A Methodology for Behavior Design of Autonomous Systems”, Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Nov. 2001.

[58] Kitamura, T., Abe, M., “Emotional Intelligence for Linking Symbolic Behaviors”, Proceedings of the 2002 IEEE International Conference on Robotics and Automation, May. 2002.

[59] www.ai.mit.edu/projects/humanoid-robotics-group/

[60] Okada, K., Kino, Y., et al, “Rapid Development System for Humanoid Vision-based Behaviors with Real-Virtual Common Interface”, Proceedings of the2002 IEEE/RSJ Conference on Intelligent Robots and Systems, October 2002.

[61] Mali, A. D., “On the Behavior-Based Architectures of Autonomous Agency”, IEEE Transactions on Systems, Man, and Cybernetics- Part C: Applications and Reviews, vol. 32, no. 3, August 2002.

[62] Suksakulchai, S., Kawamura, K., “A Behavior Monitoring Architecture for a Mobile Robot: Enhancing Intelligence to Behavior-Based Robotics”, Proceedings 2003 IEEE International Symposium on Computational Intelligence in Robotics and Automation, July 2003.

[63] www.cyberbotics.com

[64] www.ode.org

[65] Arkin, R., C., “Behavior-Based Robotics”. MIT Press, Cambridge, Ma, 1998.

[66] Saffiotti, A., Ruspini, E., Kokolige, K., “Using Fuzzy Logic for Mobile Robot Control in International Handbokk of Fuzzy Sets”. Kluwer Academic Publishing, 1999.

APPENDIX A

ON KINEMATICS OF WHEELED MOBILE ROBOTS

Many different configurations for locomotion of a wheeled robot can be seen in the literature. Some of them are listed as follows:

A.1 DIFFERENTIAL DRIVE

The differential drive is a two-wheeled drive system with independent actuators for each wheel. The name refers to the fact that the motion vector of the robot is sum of the independent wheel motions, something that is also true of the mechanical differential (however, this drive system does not use a mechanical differential). The drive wheels are usually placed on each side of the robot and toward the front:

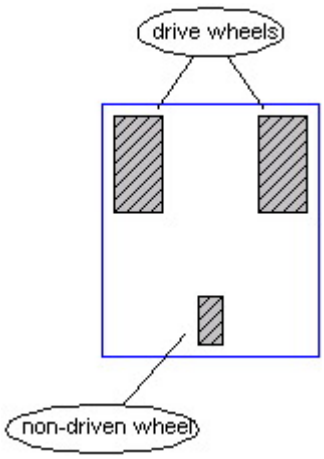

Figure A.1 Differential Drive System

In the above diagram, the large gray crosshatched rectangles are the drive wheels. The small gray crosshatched rectangle is a non-driven wheel which forms a tripod-

like support structure for the body of the robot. Often, the non-driven wheel is a caster wheel, a small swiveled wheel used on office furniture

Unfortunately, caster wheels can cause problems if the robot reverses its direction. Then the caster wheel must turn 180 degrees and, in the process, the offset swivel can impart an undesired motion vector to the robot. This may result in a translational heading error. However, if the robot always changes direction by moving forward and turning, a caster wheel may be okay. Another alternative to a caster wheel is a captive ball which does not use a swivel mechanism. In the case of small robots, a rolling device is not strictly necessary if the floor is smooth--some robots have used fixed rounded Lego parts in place of captive balls. The only drawback is the increased friction component as the Lego piece must slide along instead of rolling.

Straight-line motion is accomplished by turning the drive wheels at the same rate in the same direction, although that's not as easy as it sounds. In-place (zero turning radius) rotation is done by turning the drive wheels at the same rate in the opposite direction. Arbitrary motion paths can be implemented by dynamically modifying the angular velocity and/or direction of the drive wheels. In practice, however, complexity is reduced by implementing motion paths as alternating sequences of straight-line translations and in-place rotations. Odometry is easier to do using this method.

Motors:

Two - One for each drive wheel.

Pros:

Simplicity - The differential drive system is very simple, often the drive wheel is directly connected to the motor (usually a gear motor--a motor with internal gear reduction--because most motors do not have enough torque to drive a wheel directly).

Cons:

Control - It can be difficult to make a differential drive robot move in a straight

line. Since the drive wheels are independent, if they are not turning at exactly the same rate the robot will veer to one side. Making the drive motors turn at the same rate is a challenge due to slight differences in the motors, friction differences in the drive trains, and friction differences in the wheel-ground interface. To ensure that the robot is traveling in a straight line, it may be necessary to adjust the motor RPM very often (many times per second). This may require interrupt-based software and assembly language programming. It is also very important to have accurate information on wheel position. This usually comes from the odometry sensors

A.2 SYNCHRO DRIVE

The synchro drive system is a two motor, three/four wheeled drive configuration where one motor rotates all wheels to produce motion and the other motor turns all wheels to change direction:

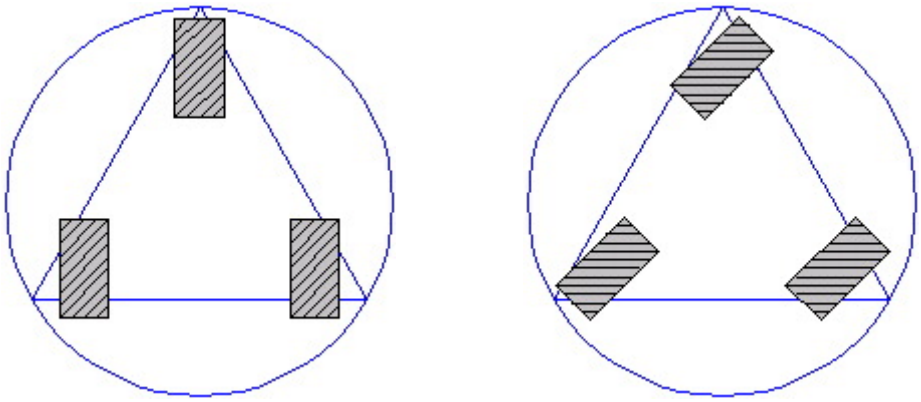

Figure A.2 Synchro Drive System

The left figure shows the wheels in the 0 degree position--in this position the robot will move forward/backward. The right figure shows the wheels turned -45 degrees. Note that all wheels have turned an equal amount. Using separate motors for translation and wheel rotation guarantees straight-line translation when the rotation motor is not actuated. This mechanical guarantee of straight-line motion is a big advantage over the differential drive method where two motors must be dynamically controlled to produce straight-line motion. Arbitrary motion

paths can of course be done by actuating both motors simultaneously. Wheel alignment is critical in this drive system, if not all wheels are parallel; the robot will not translate in a straight line.

Motors:

Two: One to rotate all the wheels, and one to turn all the wheels.

Pros:

Control - Separate motors for translation and rotation make control much easier. Straight-line motion is guaranteed mechanically. There is no need for interrupt-based control as in the case of the differential drive method.

Cons:

Complexity - The mechanism which permits all wheels to be driven by one motor and turned by another motor is complex. It is an open problem whether it can be implemented using Lego components (it probably can, but it might not be easy or practical). In addition, wheel alignment is critical.

A.3 CAR-TYPE DRIVE

Car-type locomotion is very common in the "real world," but not as common in the "robot world." Car-type locomotion (and its cousin, tricycle locomotion) is characterized by a pair of driving wheels and a separate pair of steering wheels (only a single steering wheel in tricycle locomotion):

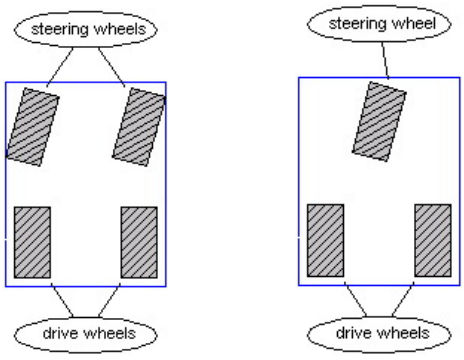

Figure A.3 Car-Type Drive System

Although these rear-wheel drive configurations are the most common, there are also front-wheel drive versions as well in which the steering wheel(s) are also the drive wheel(s). An advantage of front-wheel drive is a smaller turning radius. Consider the figure below:

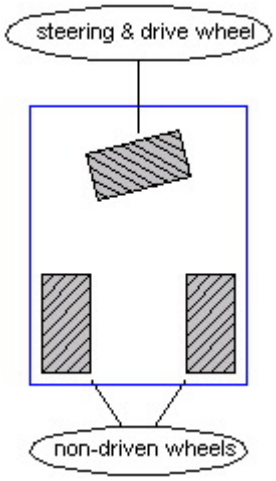

Figure A.4 Front Wheel Drive Car-Type Drive System

If this were a rear-wheel drive system it is unlikely that such a sharp turn could be accomplished since the majority of the forward force produced by the rear wheels cannot be used for motion (a 90 degree turn of the front wheel would be impossible in a rear-drive system). However, if the front wheel is driven there is no problem with this amount of turn; indeed, even a 90 degree turn of the front wheel could be performed.

The placement of odometry sensors is an issue in any car-type locomotion system where more than one wheel is providing thrust. As in a car, if the rear wheels are driven by a solid axle and a differential is not used the rear wheels will slip when turning because they must travel different distances (the wheel on the outside travels a farther distance that the wheel on the inside). Since any wheel slip will reduce odometry accuracy, placing the shaft encoder on a wheel which does not slip is advantageous.

A car-type drive is one of the simplest locomotion systems in which separate motors control translation and turning (a big advantage compared to the differential drive system). This is why it is popular for human-driven vehicles. However, the simplicity comes at a price: the car-type drive is a non-holonomic actuation system. A non-holonomic system is one in which the actuators do not directly control one or more of the degrees-of-freedom of the system, but instead are coupled such that orientation becomes much more complicated than in a holonomic system. For example, planar motion requires two degrees of freedom: x and y. Any robot architecture that allows for direct and, possibly simultaneous, motion along the x and y axes would be holonomic. Most robots do not have orthogonal actuators, although there are some that do. However, any x and y movement can be expressed in polar coordinates, combinations or rotations and translations. Therefore, any robot that can perform arbitrary rotations and translations is also holonomic. However, a car-type drive cannot perform rotations without translating. Thus, the degrees-of-freedom are linked and the system is non-holonomic. The obvious example of this is parallel parking a car. If a car used a synchro drive system (which is holonomic), then the car could position itself next to an empty parking space, rotate its wheels so that the direction of motion is into the parking space, and then translate into the space. Instead, a car must perform many forward/backward motions in order to accomplish a sideway translation because there is no actuator that can directly perform sideways movement. The result is a system for which path planning is much more difficult.

Motors:

Two - One for translation and one for rotating the turning wheel(s).

Pros:

Simplicity - One of the simplest locomotion systems to implement with one caveat: the turning mechanism must be precisely controlled. A small position error in the turning mechanism can cause large odometry errors.

Cons:

Planning - Planning is difficult because the system is non-holonomic. Note that the difficulty of a non-holonomic system is relative to the environment. On a

highway, path planning is easy because the motion is mostly forward with no absolute movement in the direction for which there is no direct actuation (sideways). However, if the environment requires motion in the direction for which there is no direct actuation, path planning is very hard.

A.4 SKID-STEER DRIVE

Skid-steer locomotion is commonly used on tracked vehicles such as tanks and bulldozers, but is also used on some four- and six-wheeled vehicles. On these vehicles, the wheels (or tracks) on each side can be driven at various speeds in forward and backward (all wheels on a side are driven at the same rate). There is no explicit steering mechanism--as the name implies steering is accomplished by actuating each side at a different rate or in a different direction, causing the wheels or tracks to slip, or skid, on the ground.

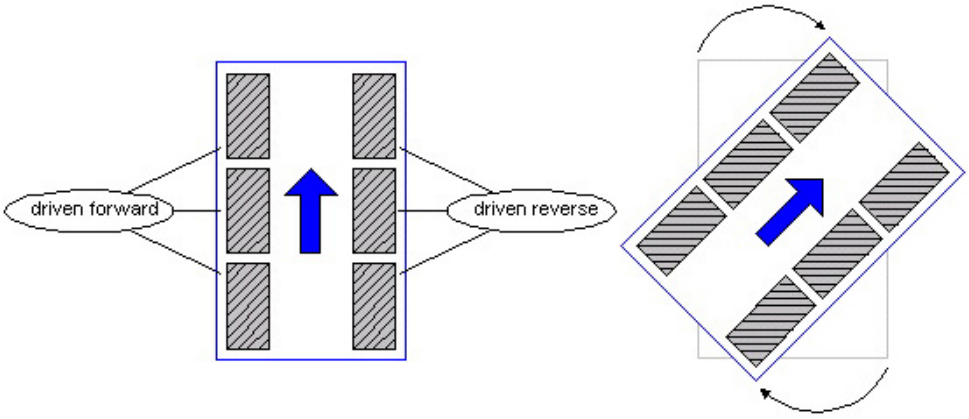

Figure A.5 Skid-Steer Drive System

In the above left figure, the wheels on the left side are driven forward and the wheels on the right side are driven in reverse at the same rate. The result is a clockwise zero radius turn about the center of the vehicle shown in the right figure. Note that throughout the turn the wheels are required to skid on the ground, with the front and rear pair of wheels skidding more than the center pair. Skidding has some disadvantages including tire/track wear but for tracked vehicles there is no alternative. (Vehicles that use skid-steer usually are off-road types such as

construction equipment and tanks--the reduced friction of a non-paved surface helps to reduce tire/track wear.) In the real world, these disadvantages are offset by the simplicity of the drive system. However, in the robot world skidding is a severe disadvantage because of the negative effect it has on odometry: wheels that are skidding are not tracking the exact movement of the robot. Since odometry is a very important sensor for position determination, skid-steer is not commonly used on robots with sparse sensing (no video cameras or sonar) that require accurate position determination.

Skid-steer is closely related to the differential drive system, replacing the caster wheel with extra drive wheels. It has the same disadvantage: moving in a straight line requires the wheels on each side to be turning at the same speed, which can be difficult to achieve. The advantage of skid-steer is increased traction and no caster wheel effect.

Motors:

Two - One for each side of the robot.

Pros:

Simplicity - No explicit steering mechanism.

Traction - Multiple drive wheels on each side gives greatly increased traction, especially on rough terrain (even greater for tracked vehicles).

Cons:

Control - Straight-line travel can be difficult to achieve.

Odometry - Skidding cause wheels to lose contact with the ground which means odometry sensors cannot accurately track the position of the vehicle

A.5 ARTICULATED DRIVE

Articulated drive is similar to the car-type drive except the turning mechanism is a deformation in the chassis of the vehicle, not pivoting of wheels:

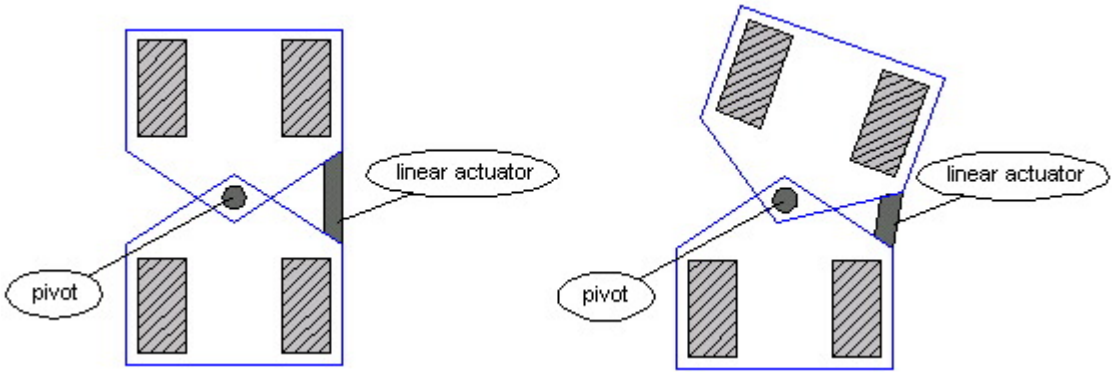

Figure A.6 Articulated Drive System

This design has the same disadvantages of the car-type drive: if multiple wheels are driven and a differential is not used, wheel slippage will occur. This design is commonly used in construction equipment where wheel slippage is not an issue (speeds are slow and the coefficient of friction with the ground is low).

Motors:

Two - One to drive the wheels and one to change the pivot angle of the chassis.

Pros:

Simplicity - One of the simplest two-wheel drive locomotion systems to implement with one caveat: the turning mechanism must be precisely controlled. A small position error in the turning mechanism can cause large odometry errors. A four-wheel drive system must use a universal joint to couple power across the pivot.

Cons:

Planning - Planning is difficult because the system is non-holonomic.

A.6 PIVOT DRIVE

Pivot drive is a unique type of locomotion. The pivot drive system is composed of a two parts: 1) a four-wheeled chassis with non-pivoting wheels and, 2) a rotating platform which can be raised or lowered:

The wheels and the platform are driven by the same motor, although the platform is geared to rotate slowly.

When the platform is raised, the wheels will translate the robot in a straight line, the platform will spin, but as it is not touching the ground, it has no effect.

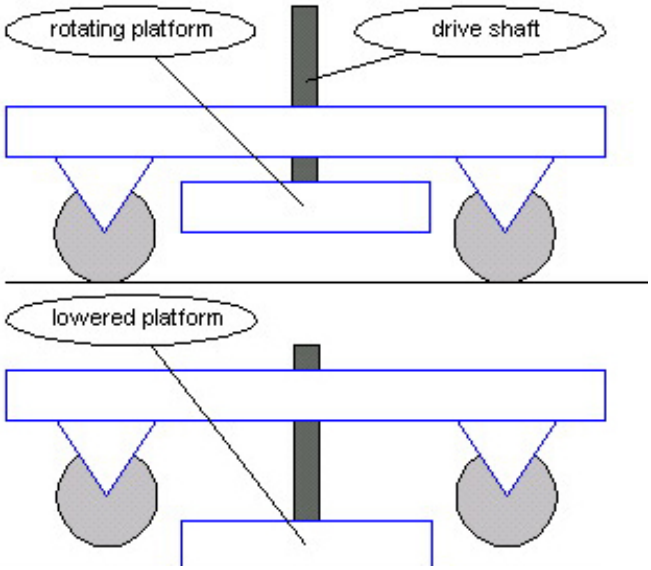

Figure A.7 Pivot Drive System

When a turn is required, the robot stops the drive motor and activates the motor which lowers the platform. Once the platform is in the down position, the drive motor is activated. Now the drive motor spins the robot since the wheels are off the ground. When the robot has rotated to the desired heading, the drive motor is stopped and the platform is raised. Now the robot can translate again using the drive motor. This design produces mechanically guaranteed straight-line motion which is a real advantage for odometry. However, the cost is the complex design necessary to raise and lower the platform while coupling power to the platform drive shaft. A simpler design (but requiring more control hardware) would use three motors: one to drive the wheels, one to rotate the platform, and one to raise/lower the platform. Fortunately, the raising/lowering of the platform would not require complicated position sensing, contact switches at the end-points of travel would suffice. The motor drive circuitry could be a couple of relays or transistors, as speed control is not necessary.

Motors:

Two - One to drive the wheels & rotating platform and one to raise/lower the platform.

Three - One to drive the wheels, one to rotate the platform, and one to raise/lower the platform.

Pros:

Control - Separate actuation of translation and rotation make control much easier. Straight-line motion is guaranteed mechanically--there is no need for interrupt-based control as in the case of the differential drive method.

Cons:

Complexity - The 2-motor system is quite complex, the 3-motor system is easier to build but would require extra position sensing electronics.

Versatility - In this system, translation and rotation are mutually exclusive, unlike the synchro drive or the dual differential drive. The inability to make arbitrary radius turns could be constraining.

A.7 DUAL DIFFERENTIAL DRIVE

Straight-line motion is important because it simplifies odometry sensing and eliminates time-critical processing. The synchro drives does give a mechanical guarantee of straight-line motion (assuming the wheels are properly aligned) but it would be difficult to build. The dual differential drive, given its name because it utilizes two mechanical differentials, also guarantees straight-line motion and it is relatively simple to construct in even Lego parts. Unlike the use of the differential in a car-type drive, where it distributes input force to two output shafts, the dual differential drive, or DDD, uses its differentials to combine the forces from two input shafts and uses the resulting sum to drive a wheel (each drive wheel has its own differential):

In the following above figure, the left and right differentials have their output shafts, **C** & **C'** attached to the drive wheels. The B and B' shafts are linked by a 3-gear train--if force is applied to any of the yellow, green, or cyan gears, the B & B'

shafts will rotate in the same direction. The A & A' shafts are linked by a 2-gear train--if force is applied to the red or magenta gears, the A and A' shafts will rotate in opposite directions.

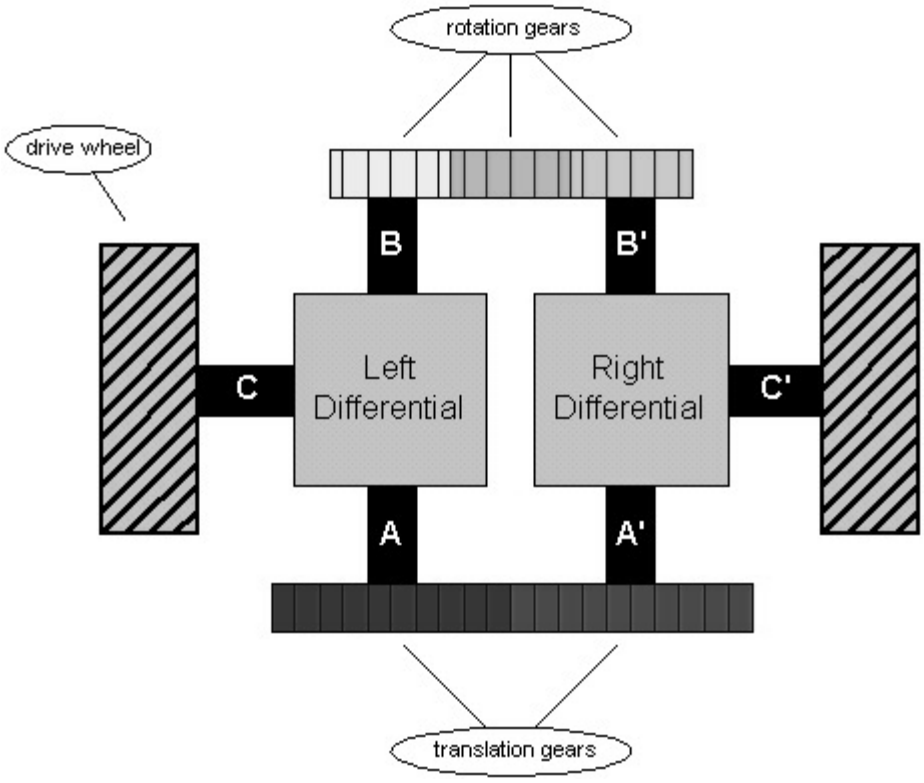

Figure A.8 Dual Differential Drive System

Now consider what happens if the B & B' shafts are prevented from rotating and force is applied to the A & A' shafts from a motor whose output gear is interfaced with one of the translation gears. A & A' will rotate in opposite directions and, since the B & B' shafts are fixed, the C shaft will rotate with the force of the A shaft and the C' shaft will rotate with the force of the A' shaft. Furthermore, since A & A' are rotating in opposite directions, C & C' will rotate in the same direction (because the differentials are facing in opposite directions). The result will be a translation of the robot and, because the entire system is mechanically linked, the wheels must rotate at the same rate producing straight-line motion. Conversely, if the A & A' shafts are prevented from rotating and force is applied to the B

& B' shafts, the C & C' shafts will rotate in opposite directions resulting in a zero radius turn about a point midway between the wheels, since the wheels are still constrained to turn at the same rate. From the preceding discussion, it follows that if a motor is connected to the translation gear train and another motor is connected to the rotation gear train, by actuating the motors in a mutually exclusive manner the robot can be made to perform straight-line translation or in-place rotation. This assumes that the motors are non-back drivable, meaning that when a motor is off it cannot be turned by an outside force acting on its output shaft. If the motors are back drivable, then some of the force can move through the differential, back driving the idle motor and possibly causing unequal force to be applied to the drive wheels. Motors without internal gear reduction are always back drivable, motors with internal gear reduction (gear motors) may or not be back drivable depending on the type of gear train. Below is a photograph of the dual differential drive made from Lego parts:

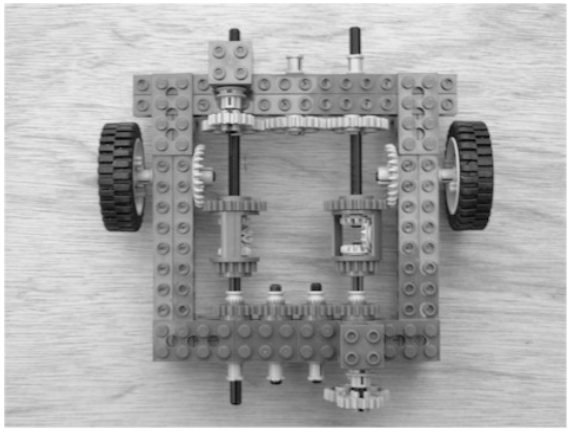

Figure A.9 Dual Differential Drive System with LEGO Blocks

There are other variations of this design-different placement of the differentials and/or using a different shaft for input/output-but the principle is the same.

Motors:

Two - One to drive the wheels in the same direction resulting in straight-line translation, one to drive the wheels in the opposite directions. Note that it is possible to actuate both motors at the same time to generate arbitrary motion paths.

Pros:

Control - Separate actuation of translation and rotation make control much easier. Straight-line motion is guaranteed mechanically--there is no need for interrupt-based control as in the case of the differential drive method.

Simplicity - Easily implemented in Lego, relatively compact design.

Cons:

Efficiency - The many gears in this system make it somewhat less efficient that a differential drive system, as there are frictional losses in the gear shafts. A heavy robot may require care in choosing the gear ratios in the system, since frictional losses rob the system of power. However, the benefits of this system outweigh the negatives.

APPENDIX B

CONTROLLABILITY OF WHEELED MOBILE ROBOTS

The controllability of wheeled mobile robots is very important for the analysis of underlying trajectory-tracking control systems. Without an adequate understanding of these control systems and in the absence of a reliable tracking system, high level planning cannot easily be performed successfully. In recent years, significant progress has been made in the application of differential geometric methods to nonlinear control systems. Part of differential geometric theory which is directly applicable to mobile robot control problems is used here. These problems have proved to be difficult to solve by other methods.

Since we are interested in describing the mobile robot with respect to the world coordinate system, the kinematics equations of the mobile robot are given as follows:

$$\dot{p} = J(p)\dot{q}, \tag{B.1}$$

where p $\in R^n$ is a generalized coordinate vector, $\dot{q} \in R^m$ is an input vector to the system, and n > m . Rewriting the above equation to the conventional form for nonlinear control, we get

$$\dot{p} = f(p) + \sum_{i=1}^{m} g(p)_i u_i, \tag{B.2}$$

where p $\in R^n, u = [u_1, u_2, ..., u_m]^t = \dot{q}$, t denotes a transpose, and f and g_i are analytic vector fields on R^m. For the mobile robot system, n-m nonholonomic constraints are written in the form

$$\sum_{i=1}^{n} a_{ji}\dot{p}_i + a_j = 0; (j = 1,2,...,m) \tag{B.3}$$

where the a_{ji} is in general a function of p and time and a_j equals zero.

Therom 1. Given a wheeled mobile-robot system, the system is locally accessible (weakly controllable) around a point $p_c \in R^n$ if it satisfies the accessibility rank condition at p_c, that is, its accessibility distribution spans R^n at the point p_c.

Proofs for Theorems 1, 2 and 4 and the six corollaries that follow can be found in [19] , Chapter II, along with related definitions. The proof of Theorem 3 is self-contained.

Using the above theorem, we have proved that the six commonly used 2-DOF or 3-DOF wheeled mobile robots are locally accessible. These mobile robots with different wheel and axle configurations are shown in Figure B.1. The coordinates (x, y) indicate the location of the robot with respect to the world coordinate system. The angle θ gives the orientation of the vehicle or the wheels with respect to a line parallel to the x axis of the reference coordinate frame. In the following kinematics equations, we assume that these vehicles roll on a plane surface without slipping.

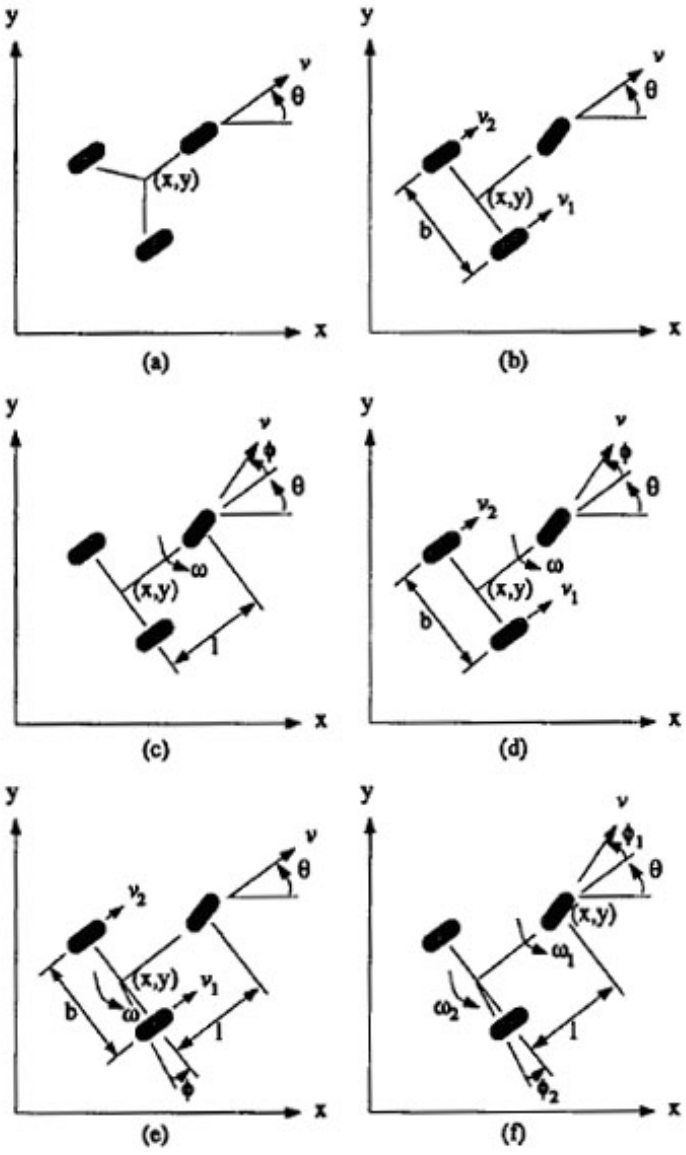

Figure B.1 2-DOF and 3-DOF Mobile Robots

Corollary 1: The synchro-drive-steering-wheel vehicle (2-DOF, Equation (4), Figure 1a) is locally accessible.

$$\dot{p} = g_1 u_1 + g_2 u_2 = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ 0 \end{bmatrix} \nu + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} w. \qquad \text{(B.4)}$$

Corollary 2: The two-rear-drive-wheel vehicle (2-DOF, Equation (5), Figure 1b) is locally accessible.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} [\cos\theta]/2 \\ [\sin\theta]/2 \\ 1/b \end{bmatrix} v_1 + \begin{bmatrix} [\cos\theta]/2 \\ [\sin\theta]/2 \\ -1/b \end{bmatrix} v_2. \tag{B.5}$$

Corollary 3: The one-front-drive-and-steering-wheel vehicle (2-DOF, Equation (6), Equation (6), Figure 1c) is locally accessible.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta\cos\phi \\ \sin\theta\cos\phi \\ [\sin\phi]/l \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} w. \tag{B.6}$$

Corollary 4: The two-rear-drive-wheel and one-front-steering-wheel vehicle (3-DOF, Equation (7), Figure 1d) is locally accessible.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} [\cos(\theta+\phi)]/2 & [\cos(\theta+\phi)]/2 & 0 \\ [\sin(\theta+\phi)]/2 & [\sin(\theta+\phi)]/2 & 0 \\ 1/b & -1/b & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ w \end{bmatrix}. \tag{B.7}$$

Corollary 5: The two-rear-drive-wheel and rear-steering vehicle (3-DOF, Equation (8), Figure 1c) is locally accessible

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \left[\cos\theta\right]/2 & \left[\cos\theta\right]/2 & 0 \\ \left[\sin\theta\right]/2 & \left[\sin\theta\right]/2 & 0 \\ 1/b+\left[\sin\phi\right]/l & -1/b-\left[\sin\phi\right]/l & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ w \end{bmatrix}. \qquad \text{(B.8)}$$

Corollary 6: The one-front-drive-and-steering-wheel and rear-steering vehicle (3-DOF, Equation (9), Figure 1f) is locally accessible.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi}_1 \\ \dot{\phi}_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta+\phi_1) & 0 & 0 \\ \sin(\theta+\phi_1) & 0 & 0 \\ \left[\sin(\phi_1-\phi_2)\right]/\left[l\cos(\phi_1)\right] & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w_1 \\ w_2 \end{bmatrix}. \qquad \text{(B.9)}$$

For the above systems, accessibility implies controllability, as stated below.

Theorem 2. The systems (4), (5), (6), (7), (8), and (9) are controllable.

The above conditions and findings cover those mobile robots most commonly used in laboratories and industry. Most existing wheeled mobile robots can be characterized by the above equations or simple variations, although some mobile robots may have more than three wheels or more than one or two caster wheels. Furthermore, although the above proofs indicate that mobile robot systems are controllable, they are not stabilizable by using continuous feedback.

It may not be true that the corresponding linearized system can be stabilized using linear feedback.

APPENDIX C

BACKPROPAGATION ALGORITHM FOR MULTILAYERED FEEDFORWARD NEURAL NETWORK

Step 0. Initialize weights: to small random values;

Step 1. Apply a sample: apply to the input a sample u^k vector having desired output y^k;

Step 2. Forward Phase:

Starting from the first hidden layer and propagating towards the output layer:

2.1. Calculate the activation values for the units at layer L as:

2.1.1. If L-1 is the input layer

$$a_{h_L}^k = \sum_{j=1}^{N} w_{jh_L} u_j^k$$

2.1.2. If L-1 is a hidden layer

$$a_{h_L}^k = \sum_{J_{L-1}=1}^{N_{L-1}} w_{j_{(L-1)}h_L} x_{j_{(L-1)}}^k$$

2.2. Calculate the output values for the units at layer L as:

$$x_{h_L}^k = f_L(a_{h_L}^k)$$

in which, use i_0 instead of h_L if it is an output layer

Step 3. Output errors: Calculate the error terms at the output layer as:

$$\delta_{i_0}^k = \left(y_{i_0}^k - x_{i_0}^k\right) f_0'\left(a_{i_o}^k\right)$$

Step 4. Backward Phase: Propagate error backward to the input layer through each layer L using the error term

$$\delta_{h_L}^k = f_L'\left(a_{h_L}^k\right)\sum_{i_{L+1}=1}^{N_{L+1}} \delta_{i_{(L+1)}}^k w_{h_L i_{(L+1)}}^k$$

in which, use i_0 instead of $i_{(L+1)}$ İF L+1 is an output layer;

Step 5. Weight update: Update weights according to the formula

$$w_{j_{(L-1)}h_L}(t+1) = w_{j_{(L-1)}h_L}(t) + \eta\delta_{h_L}^k x_{j_{(L-1)}}^k$$

Step 6. Repeat steps 1-5 until the stop criterion is satisfied, which may be chosen as the mean of the total errors (MSE- mean square error)

$$< e^k >=< \frac{1}{2}\sum_{i_0=1}^{M}(y_{i_0}^k - x_{i_0}^k)^2 >$$

is sufficiently small.

Two C++ files are written for this utility, BackProg.h and BackProp.cpp. They are given as follows:

<BackProp.h>

```
////////////////////////////////////////////////////////////////////////////
//          Fully connected multilayered feed                 //
//          forward artificial neural network using           //
//          Backpropogation algorithm for training.           //
////////////////////////////////////////////////////////////////////////////
#ifndef backprop_h
#define backprop_h
#include<assert.h>
#include<iostream.h>
#include<stdio.h>
#include<math.h>

class CBackProp{
//          output of each neuron
            double **out;
//          delta error value for each neuron
            double **delta;
//          vector of weights for each neuron
            double ***weight;
//          no of layers in net
//          including input layer
            int numl;
//          vector of numl elements for size
//          of each layer
            int *lsize;
//          learning rate
            double beta;
```

```
//          momentum parameter
            double alpha;
//          storage for weight-change made
//          in previous epoch
            double ***prevDwt;
//          squashing function
            double sigmoid(double in);

public:
            ~CBackProp();
//          initializes and allocates memory
            CBackProp(int nl,int *sz,double b,double a);
//          backpropogates error for one set of input
            void bpgt(double *in,double *tgt);
//          feed forwards activations for one set of inputs
            void ffwd(double *in);
//          returns mean square error of the net
            double mse(double *tgt) const;
//          returns i'th output of the net
            double Out(int i) const;
};
#endif
```

<BackProp.cpp>

```
#include "backprop.h"
#include <time.h>
#include <stdlib.h>

//          initializes and allocates memory on heap
CBackProp::CBackProp(int nl,int *sz,double b,double a):beta(b),alpha(a)
{
            //          set no of layers and their sizes
            numl=nl;
            lsize=new int[numl];

            for(int i=0;i<numl;i++){
                        lsize[i]=sz[i];
            }
            //          allocate memory for output of each neuron
            out = new double*[numl];

            for( i=0;i<numl;i++){
                        out[i]=new double[lsize[i]];
            }
            //          allocate memory for delta
            delta = new double*[numl];

            for(i=1;i<numl;i++){
                        delta[i]=new double[lsize[i]];
            }
            //          allocate memory for weights
            weight = new double**[numl];
```

```
	for(i=1;i<numl;i++){
		weight[i]=new double*[lsize[i]];
	}
	for(i=1;i<numl;i++){
		for(int j=0;j<lsize[i];j++){
			weight[i][j]=new double[lsize[i-1]+1];
		}
	}
	//	allocate memory for previous weights
	prevDwt = new double**[numl];

	for(i=1;i<numl;i++){
		prevDwt[i]=new double*[lsize[i]];
	}
	for(i=1;i<numl;i++){
		for(int j=0;j<lsize[i];j++){
			prevDwt[i][j]=new double[lsize[i-1]+1];
		}
	}
	//	seed and assign random weights
	srand((unsigned)(time(NULL)));
	for(i=1;i<numl;i++)
		for(int j=0;j<lsize[i];j++)
			for(int k=0;k<lsize[i-1]+1;k++)
				weight[i][j][k]=(double)(rand())/(RAND_MAX/2) - 1;//32767

	//	initialize previous weights to 0 for first iteration
	for(i=1;i<numl;i++)
		for(int j=0;j<lsize[i];j++)
			for(int k=0;k<lsize[i-1]+1;k++)
				prevDwt[i][j][k]=(double)0.0;

// Note that the following variables are unused,
//
// delta[0]
// weight[0]
// prevDwt[0]

//  This is done intentionally to maintain consistency in numbering the layers.
//  Since for a net having n layers, input layer is refered to as 0th layer,
//  first hidden layer as 1st layer and the nth layer as output layer. And
//  first (0th) layer just stores the inputs hence there is no delta or weight
//  values corresponding to it.
}

CBackProp::~CBackProp()
{
	//	free out
	for(int i=0;i<numl;i++)
		delete[] out[i];
	delete[] out;
	//	free delta
	for(i=1;i<numl;i++)
		delete[] delta[i];
	delete[] delta;
	//	free weight
```

```
	for(i=1;i<numl;i++)
		for(int j=0;j<lsize[i];j++)
			delete[] weight[i][j];
	for(i=1;i<numl;i++)
		delete[] weight[i];
	delete[] weight;
	//	free prevDwt
	for(i=1;i<numl;i++)
		for(int j=0;j<lsize[i];j++)
			delete[] prevDwt[i][j];
	for(i=1;i<numl;i++)
		delete[] prevDwt[i];
	delete[] prevDwt;
	//	free layer info
	delete[] lsize;
}
//	sigmoid function
double CBackProp::sigmoid(double in)
{
		return (double)(1/(1+exp(-in)));
}
//	mean square error
double CBackProp::mse(double *tgt) const
{
	double mse=0;
	for(int i=0;i<lsize[numl-1];i++){
		mse+=(tgt[i]-out[numl-1][i])*(tgt[i]-out[numl-1][i]);
	}
	return mse/2;
}
//	returns i'th output of the net
double CBackProp::Out(int i) const
{
	return out[numl-1][i];
}
// feed forward one set of input
void CBackProp::ffwd(double *in)
{
	double sum;

	//	assign content to input layer
	for(int i=0;i<lsize[0];i++)
		out[0][i]=in[i];  // output_from_neuron(i,j) J'th neuron in I'th Layer

	//	assign output(activation) value
	//	to each neuron using sigmoid function
	for(i=1;i<numl;i++){				// For each layer
		for(int j=0;j<lsize[i];j++){		// For each neuron in current layer
			sum=0.0;
			for(int k=0;k<lsize[i-1];k++){
				// For input from each neuron in preceeding layer
		sum+= out[i-1][k]*weight[i][j][k]; // Apply weight to inputs and add to sum
			}

			sum+=weight[i][j][lsize[i-1]];	// Apply bias
			out[i][j]=sigmoid(sum);	// Apply sigmoid function
		}
```

```
        }
}
//      back propagate errors from output
//      layer until the first hidden layer
void CBackProp::bpgt(double *in,double *tgt)
{
        double sum;

        //      update output values for each neuron
        ffwd(in);

        //      find delta for output layer
        for(int i=0;i<lsize[numl-1];i++){
                delta[numl-1][i]=out[numl-1][i]*
                (1-out[numl-1][i])*(tgt[i]-out[numl-1][i]);
        }

        //      find delta for hidden layers
        for(i=numl-2;i>0;i--){
                for(int j=0;j<lsize[i];j++){
                        sum=0.0;
                        for(int k=0;k<lsize[i+1];k++){
                                sum+=delta[i+1][k]*weight[i+1][k][j];
                        }
                        delta[i][j]=out[i][j]*(1-out[i][j])*sum;
                }
        }

        //      apply momentum ( does nothing if alpha=0 )
        for(i=1;i<numl;i++){
                for(int j=0;j<lsize[i];j++){
                        for(int k=0;k<lsize[i-1];k++){
                                weight[i][j][k]+=alpha*prevDwt[i][j][k];
                        }
                        weight[i][j][lsize[i-1]]+=alpha*prevDwt[i][j][lsize[i-1]];
                }
        }

        //      adjust weights using steepest descent (uses beta - learning rate)
        for(i=1;i<numl;i++){
                for(int j=0;j<lsize[i];j++){
                        for(int k=0;k<lsize[i-1];k++){
                                prevDwt[i][j][k]=beta*delta[i][j]*out[i-1][k];
                                weight[i][j][k]+=prevDwt[i][j][k];
                        }
                        prevDwt[i][j][lsize[i-1]]=beta*delta[i][j];
                        weight[i][j][lsize[i-1]]+=prevDwt[i][j][lsize[i-1]];
                }
        }
}
```

APPENDIX D

REFERENCE GUIDE FOR DEVELOPERS

This document contains the functions that are used to construct "Behavior-Based Mobile Robot Control Simulation Environment". These functions are given to help developers to extend the scope of the simulation environment. The presence of these functions can also give some inspiration to other developers also helping what difficulties wait and how they can be solved to construct other similar simulation environments.

The very first thing to do is the planning phase in building simulations. Deciding on what simulation type will be used is a challenging problem. One needs to search for different simulation types which best suits the problem. Once the simulation type, such as discreet time, fixed step simulation, is decided, the other necessities shape themselves accordingly.

The "Behavior-Based Mobile Robot Control Simulation Environment" is physics based, fixed step, discreet time simulation. The problem itself, robot control of wheeled mobile robots, brings the physics dependency, where as fixed step discreet time option is a design choice of the developer.

The whole code is written in C++. The program uses some dependencies of open source. These dependencies are selected, as they are free-easily available and well documented. The dependencies are as follows:

- **OpenCV**: Open Computer Vision Library is an open source project widely used in computer science as image and video processing tool. The projects' official web site is: "www.opencv.org."
- **ODE** : Open Dynamics Engine is an open source project which started as a PhD. work. The engine is so successful in computation time means

that it is widely used in physics based simulations that require modest accuracy and high timing performance. Its modifications are also used in many professional simulators and computer game physics cores. The projects website is: "www.ode.org".

Both of these open source projects are distributed in C/C++ library form which can be compiled in many common compilers including Visual Studio 6.0, .NET, Borland, etc. Precompiled versions are also available for direct injection to your application.

The "Behavior-Based Mobile Robot Control Simulation Environment" is coded entirely in C++ using Microsoft Visual Studio 6.0. The project is tested also in .NET 2003 environment and succeeded with minor modifications in some header files. Advantages of using C++ is used whenever possible. The units that should be individual in the simulation such as boxes, robots, and sensors are designed as classes. The usage of classes brings the availability of multiple usages of instances of that class, namely multiple robots, multiple sensors, multiple of everything in class.

This document tries to give the available functions for the implemented classes in subsections. Some assumptions made on the proper function of the methods are given in corresponding method information.

D.1 CLASSES

D.1.1 CWORLD

Every other object is registered as a child to this parent class.

void Draw (bool)

This is the function used for drawing in OpenGL contents. If parameter is passed as "false," then everything registered is drawn, otherwise the view is considered as module view, and some auxiliary objects such as sensors are not drawn.

void Update ()

This function updates every object before simulation enhances in time. Sets the robots' sensors properly, adds virtual robot boxes to be used in collision algorithms, etc.

void SetGravity (float)

Sets the gravity scalar used in ODE physical calculations.

void SetERP (float)

Sets the ODE ERP constant.

void SetCFM (float)

Sets the ODE CFM constant.

void SetStep (float)

Sets the simulation time step.

void SetSizeX (float)

Sets the size along x-axis.

void SetSizeY (float)

Sets the size along y-axis.

void SetSizeZ (float)

Sets the size along z-axis.

float GetGravity ()

Returns the gravity scalar used in ODE physical calculations.

float GetERP ()

Returns the ODE ERP constant.

float GetCFM ()

Returns the ODE CFM constant.

float GetStep ()

Returns the simulation time step.

float GetSizeX ()

Returns the World length along the main x-axis.

float GetSizeY ()

Returns the World width along the main y-axis.

float GetSizeZ ()

Returns the World height along the main z-axis.

void RemoveVirtualRobotBoxes ()

Removes the virtual boxes that should be inserted for robot-robot collisions. This function should be called after Collision in each step.

void MyCollisionCallback (void*, dGeomID, dGeomID)

This method handles the collisions of ODE geom. primitives. It is an overridden method that invokes proper functions in ODE library.

int AnyStaticBox (dGeomID Geom)

This function is used in collision handling for physics. Returns true if the Geom. is any static box.

int AnyVirtualRobotBox (dGeomID Geom)

This function is used in collision handling for physics. Returns true if the Geom. is any virtual robot box.(Temporary box primitive that is inserted at the beginning of each step in the location of each robot and erased at the end of the step for a fresh start for new step)

void Box (GLfloat x1, GLfloat y1, GLfloat z1, GLfloat x2, GLfloat y2, GLfloat z2)

Draws the surrounding boxes of the world. The two 3 dimensional diagonal endpoints are passed in world coordinates as parameters.

D.1.2 CBOX

This the primitive class used to represent boxes of various kinds. Boxes are used as static walls, static obstacles, and resources for the red and the blue team as a power supply.

void Draw ()

Draws the box in 3D space according to its settings.

void SetEnergy (float val)

Sets the energy level that the resource holds.

float GetEnergy ()

Returns the current energy level of the resource.

float GetLength ()

Returns the length along x-axis.

float GetWidth ()

Returns the width along y-axis

float GetHeight ()

Returns the height along z-axis.

int GetType ()

Returns the type of the box. ‘1’ stands for red resource, ‘2’ stands for blue resource, ‘3’ stands for static box object.

D.1.3 CROBOT

This class is the main simulated object. The class encapsulates all the geometric, dynamic and auxiliary properties to form a 2-wheel differential drive mobile robot.

void SetLeader (bool)

Sets the robot as the leader robot. This property can be used in behaviors

such as following a leader robot within a group.

void SetPosition ()

Sets the position using an internal Position vector variable.

CVertex* GetPosition ()

Sets the position of the center of mass of the mobile robot chassis.

void SetSpeed (float)

Sets the speed of the robot

float GetSpeed ()

Returns the speed of the robot

void SetMaxSpeed (float)

Sets the maximum allowable speed of the robot

float GetMaxSpeed ()

Returns the maximum allowable speed of the robot

void SetEnergy (float)

Sets the energy of the robot, battery charge level.

float GetEnergy ()

Returns the energy of the robot, battery charge level.

void SetSizes (float, float,float)

Sets the sizes of the main robot chassis.

void Draw (bool)

Draws the robot according to the passed flag, module view or not, that is penetrated from CWorld.Draw ().

void ApplyKinematics ()

Applies the inverse kinematics to determine wheel rotation speeds for the required turn.(Updated by Dynamics)

void Update (std::vector<CBox> Objects, std::vector<CVertex> Triangles)

Updates the internal states and primitives. Calls from the CWorld.Update().

void ApplyDynamics ()

Applies the dynamics to the robot, including collision based on physical constraints set on CWorld parent.

void Box (GLfloat x1, GLfloat y1, GLfloat z1, GLfloat x2, GLfloat y2, GLfloat z2)

Draws main chassis.

D.1.4 SENSORS

Different sensors are implemented in software. These sensor types are generally do not share common properties so no base class for sensors is implemented. A new class is implemented for each sensor type:

D.1.4.1 CBeamRangeSensor

This sensor class uses array to measure the distance. Sensor has a starting point, a heading in 3D, and an active range. The end of the ray is calculated using other objects faces in 3D. The nearest collision point of all of the surfaces with this ray is returned as the measured distance. If no collision occurs with this ray, then maximum active range is returned.

D.1.4.2 CProximitySensor

This is a volume sensor. It gives logical ‘0’ or ‘1’ values as the measurements. It is used to check whether an obstacle lies within the coverage of the sensor or not. It is composed of 5 points, 1 point for the base location of the sensor. The other 4 points are calculated in 3D according to the 3D heading information of the sensor. These 5 points form a rectangular pyramid as the coverage volume.

D.1.4.3 CVisionModule

Vision module gives 100x100 pixels RGB representation of the environment viewed from a defined location on the mobile robot. It is OpenGL content. Raw

and processed versions can be drawn into OpenGL windows and OpenCV highgui windows respectively.

D.1.5 CBEHAVIORS

CBehavior class is a parent class; different behaviors are based on this class. Every other behavior extended from this base class uses the same methods.

float Apply (std::vector<CBox>, std::vector<CVertex>)

> Applies the behavior algorithm. The passed parameters are the obstacles registered to the CWorld. These parameters are fed into robot sensors for measurements.

float GetPriority ()

> Returns the priority value.

void SetPriority (float)

> Sets the priority value to the passed parameter.

D.1.6 CBEHAVIORCOORDINATORS

Coordinators are used to evaluate a single motor command, using many behaviors commands. This class is also a base class and different Behavior Coordinators applying different policies such as command fusion and arbitration are derived from this base class. It implements only one method.

float Apply (std::vector<CBox>, std::vector<CVertex>)

> Applies the coordination (command fusion, arbitration) policy to the robot. Since this method calls behaviors (one by one or selected behavior), the parameters are passed by this method to behaviors' Apply () method.