Behavior representation and simulation for Military Operations on Urbanized Terrain (MOUT)

Shen, Zhuoqian

2006

Shen, Z. (2006). Behavior representation and simulation for Military Operations on Urbanized Terrain (MOUT). Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2477>

<https://doi.org/10.32657/10356/2477>

Nanyang Technological University

Downloaded on 26 Apr 2023 20:04:46 SGT

**BEHAVIOR REPRESENTATION AND SIMULATION
FOR MILITARY OPERATIONS
ON URBANIZED TERRAIN (MOUT)**

SHEN ZHUOQIAN

**SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY**

2006

**Behavior Representation and Simulation
for Military Operations
on Urbanized Terrain (MOUT)**

Shen Zhuoqian

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2006

ACKNOWLEDGEMENTS

First of all, I would like to give my warmest gratitude to my supervisor, Dr. Zhou Suiping, for his openmindedness and enthusiasm on research. This thesis would not have been possible without his help. His approachability and insightful comments have made my research in the Parallel and Distributed Computing Centre (PDCC) a fruitful and memorable experience.

My thanks also go to two final year project students under Dr. Zhou's supervision, Mr. Luo Linbo and Mr. Chin Chee Yung, for their efforts in testing and coding. I would not be able to come this far without their collaboration.

Finally, I thank my parents, to whom I dedicate this thesis, for their endless love, support, and encouragement. They are always the impetus behind me.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

Acknowledgements.....	iii
Table of Contents.....	v
List of Figures.....	ix
List of Tables.....	xi
Summary.....	xiii
Chapter 1: Introduction.....	1
1.1 Background.....	1
1.2 Challenges.....	2
1.3 Objectives.....	3
1.4 Thesis Organization.....	5
Chapter 2: Literature Review.....	7
2.1 Overview.....	7
2.2 Intelligent Agent and Multi-agent Systems.....	7
2.2.1 Action Selection.....	8
2.2.2 Learning.....	15
2.2.3 Multi-agent Systems.....	17
2.3 Bots in Games.....	19
2.3.1 Neuralbot.....	20
2.3.2 Quake III Arena Bot.....	20
2.3.3 Soar QuakeBot.....	21

TABLE OF CONTENTS

2.4	Off-the-shelf Game Engines.....	22
2.4.1	Approaches.....	23
2.4.2	Possible Choices.....	24
Chapter 3: AI Framework Design.....		27
3.1	Overview.....	27
3.2	MOUT Simulation System.....	27
3.2.1	System Organization.....	27
3.2.2	GameBots API.....	28
3.2.3	JavaBot API.....	31
3.3	AI Framework.....	32
3.3.1	Design Guideline.....	32
3.3.2	A Layered Framework.....	33
Chapter 4: Implementation.....		37
4.1	Overview.....	37
4.2	Bot Behaviors Module.....	37
4.3	Path-finding Module.....	40
4.3.1	World Representation.....	40
4.3.2	Algorithm.....	44
4.4	Group Coordination Module.....	48
Chapter 5: Simulation and Empirical Studies.....		55
5.1	Overview.....	55
5.2	System Setup.....	55
5.3	Simulations.....	56

TABLE OF CONTENTS

5.3.1 Scenario 1.....	56
5.3.2 Scenario 2.....	59
Chapter 6: Conclusion and Future Work.....	65
6.1 Conclusion.....	65
6.2 Future Work.....	65
6.2.1 A Full-fledged Task Scheduling Module.....	67
6.2.2 Spatial Reasoning Algorithms.....	67
6.2.3 Learning Capabilities.....	68
6.2.4 Coordination and Cooperation.....	69
6.2.5 GameBots Enhancements.....	70
References.....	71
Publication.....	79

TABLE OF CONTENTS

LIST OF FIGURES

Figure 2.1: The Architecture of the TouringMachine.....	12
Figure 2.2: The Architecture of INTERRAP	14
Figure 3.1: System Organization.....	28
Figure 3.2: JavaBot API.....	31
Figure 3.3: The Conceptual Model of the AI Framework.....	34
Figure 4.1: A Sample FSM.....	38
Figure 4.2: FSM for 'Flee without Covering Fire'.....	40
Figure 4.3: Path Node and Grid Representation.....	41
Figure 4.4: Area-based Grid Representation.....	42
Figure 4.5: Placement of Observing Nodes.....	43
Figure 4.6: Placement of Securing Nodes.....	43
Figure 4.7: Calculation of Following Point.....	50
Figure 4.8: Representation of Obstacles.....	52
Figure 4.9: Calculation of Obstacle Avoidance.....	53
Figure 5.1: System Setup.....	56
Figure 5.2: Test Environment of Scenario 1.....	57
Figure 5.3: Scenario 1.....	58
Figure 5.4: Screenshots of Magnum Docks Deluxe.....	59
Figure 5.5: Obtain Item 1 Illustration.....	61
Figure 5.6: Obtain Item 2 Illustration.....	62-63
Figure 5.7: Obtain Item 3 Illustration.....	64

LIST OF FIGURES

LIST OF TABLES

Table 3.1: Gambots Network Messages and Descriptions.....30

LIST OF TABLES

SUMMARY

Creating human-like behaviors in a virtual warfare environment is still a challenging task to date. It is a cutting edge field that involves a variety of research areas such as intelligent agent & multi-agent systems, behavioral animation, cognitive psychology, and etc. In recent years, advancement in simulating Military Operations on Urbanized Terrain (MOUT) has greatly appealed to both the military departments as well as the game community. Comparing to military training in the real world, it could save tremendous cost for the military departments to carry out training exercises in a virtual environment. As for the game developers, it is also wise to put more efforts on “Game AI” to make a difference since the graphics is reaching the paramount after achieving amazing enhancement in recent years.

This thesis presents our research on behavior representation and simulation for MOUT. First of all, a layered AI framework for the artificial players (bots) has been proposed. The framework is designed to be flexible, extensible, integrable and independent of simulation platforms. The functionality of each module and the interfaces between each other have been specified. Secondly, for fast prototyping, the AI framework is implemented on the Unreal Tournament (UT) game engine which provides high-fidelity whilst affordable 3D virtual environments. In this stage, various algorithms and representations, such as the area-based grid representation, the hierarchical finite state machines, and the steering behaviors, etc., have been successfully employed to integrate all the modules into an efficient and consistent system. Finally, a number of simulations have been conducted and two scenarios have been illustrated and analyzed accordingly.

SUMMARY

Overall, the AI framework is effective and efficient in creating realistic and robust bot behaviors in various combat scenarios. The bots exhibit goal directed behaviors to achieve certain task while react to immediate surroundings in different situations. In addition, various warfare tactics, such as securing an entrance, are successfully employed by the bots in different scenarios. Although modeling human behaviors in MOUT simulations involves a number of open questions that are still unsolved, our framework and the resulting implementation have been verified to be a solid basis for further efforts.

CHAPTER 1: INTRODUCTION

1.1 Background

The history of emulating human thinking patterns and simulating human-like behaviors can be traced back to half a century ago. The term, “Artificial Intelligence”, was coined by John McCarthy in 1956. A variety of algorithms and methodologies have been developed since then, such as planning, machine learning, pattern recognition and so on. A more recent trend in AI research is to build intelligent agent and multi-agent systems. An intelligent agent is usually embodied either physically as a machine or virtually as a computer program which can automatically perceive its environment, reason upon its perception, and act according to its decisions. Software agents are widely used in the areas such as controlling manufacturing process, transportation management, air traffic control, telecommunications systems, etc.

In the early 1990s, a new research trend thrived that is to simulate human behaviors in Military Operations on Urbanized Terrain (MOUT). It is a cutting edge field that involves a variety of research areas such as intelligent agent & multi-agent systems, behavioral animation, cognitive psychology, and etc. In recent years, both the researchers from the military training community and the developers from the game industry have made great contributions in conducting MOUT simulations. They share the same objective that is to simulate more realistic and robust behaviors for autonomous or semi-autonomous characters. These characters can either act as opponents against human players or act as team members that can cooperate with human players in the virtual environment.

Comparing to military training in the real world, it could save tremendous cost for the military departments to carry out training exercises in a virtual environment. As for the game

developers, it is also wise to put more efforts on “Game AI” to make a difference since the graphics is reaching the paramount after achieving amazing enhancement in recent years. In fact, we have seen quite a number of examples in how the training community and the game industry benefit from each other. On the one hand, the game industry is providing more realistic and modifiable game engines that can be used as simulation platforms. On the other hand, researchers are devoting efforts to create more intelligent characters by applying different AI techniques, which leads to more interesting and playable games.

The Institute for Creative Technology (ICT) at USC has developed two simulation systems from scratch, called Full Spectrum Command and Full Spectrum Warrior, to produce immersive, interactive and real time simulations [ICT, 2003]. The first system is initially for the U.S. Army training, and later for both the U.S. Army and the Singapore Armed Force training. The second system is actually a game developed for Microsoft Xbox. In contrast to ICT, the MOVES Institute at the Naval Postgraduate School (NPS) has used the off-the-shelf Unreal engine to develop the ‘America’s Army’ game [MOVES, 2004]. The key feature of these simulation systems or games is the intelligent and realistic Non-Player Characters (NPCs) or bots that can sense, reason, and act in the virtual environment.

1.2 Challenges

In the current computer games, especially the First Person Shooter (FPS) games, there do exist some NPCs (or bots) that seems to behave like human players. These bots usually act as opponents that can patrol around in the map and initiate an attack when they see the players or hear sounds from the players. However, these seemed intelligent behaviors are actually implemented using simple and straightforward mechanisms. A typical example is the event triggering system. Each time an important event happened in the game, a signal

will be sent to the corresponding bots and certain script will be triggered, such as shooting the player or ducking to avoid the incoming missile. Although such mechanisms are computationally efficient and give human players an illusion of intelligence, they do not exhibit really smart behaviors. First of all, these bots usually cannot pursue a long term goal as a task like rescuing some hostages in a building from the terrorists. In other words, these bots only react to immediate surroundings by executing predefined routines. Secondly, as the bots repeat their behaviors from time to time, human players can quickly make precise predictions on the bot's behaviors after a few rounds of game play. In computer games, such experience reduces the replayability. In military trainings, such experiences will lead to negative training effects. Thirdly, these bots basically do not exhibit group behaviors, i.e. they do not coordinate or cooperate between each individual's behaviors through interactions. Therefore, tasks that require teamwork cannot be carried out by the bots.

In order to make the bots more intelligent, we need to employ more sophisticated AI algorithms and methodologies. Although algorithms such as planning, reasoning, inference have been developed for decades, it is still difficult to design an integral AI system that exhibits human-level intelligence. Additionally, compared to the agents for the purpose of industrial control, these bots face real life problems that are more complicated. Moreover, in an MOUT simulation system, only limited computational power can be provided to run the AI algorithms. Thus, the implementation of the AI framework should be as efficient as possible.

1.3 Objectives

First of all, we aim to design an AI framework as the brain of the bots that engage in the urban warfare. Each bot should be able to achieve certain tasks either independently or in cooperation with his/her teammates. In the MOUT simulation, the virtual environment is very

dynamic and competitive. The bot needs to balance his/her long term goals and short term goals according to the situation. In addition, as the tasks assigned to the bots may vary in different MOUT scenarios, the framework should be extensible to allow changes in task specifications. Moreover, in order to achieve a task, the bot needs to exhibit a variety of capabilities, such as finding a path to a specific location, sniping the enemies tactically, or setting an ambush with his/her teammates. The framework should facilitate such variability. The key issue here is to identify the functionality of each module and clearly define the interfaces between each other so as to make sure that all the modules work effectively and efficiently as an integral system. Furthermore, the framework should be independent of the simulation platforms, which means that the framework could be implemented on different platforms as long as the specifications of its sensory input and action output can be satisfied.

Secondly, taking the AI framework as the guideline, we need to implement the framework. At this stage, different algorithms need to be implemented with regard to respective functionality of the framework, and different knowledge representations need to be employed for the warfare tactics and strategies. The evaluation of the implementation can be based on two criteria. On the one hand, the behaviors generated should be realistic and robust. Realistic behaviors mean that it is difficult for observers to differentiate the bots from the human players. Robust behaviors mean the bots may not be easily stuck in an unexpected situation. On the other hand, the implementation should not incur heavy computational cost so that the simulations can be run in real-time on commonly-used PCs.

Finally, extensive experiments should be conducted to verify the effectiveness of our proposed framework. For fast prototyping, we prefer using an off-the-shelf commercial FPS game as our simulation platform. Such games provide high-fidelity 3D virtual environments whilst they are affordable to us. Several possible choices need to be investigated and

compared for our purpose. Building up the simulation system may also need a lot of work to configure different parts so as to meet our AI specifications.

1.4 Thesis Organization

Chapter 2 overviews some work in the relevant areas. Firstly, we investigate issues in designing intelligent agent and multi-agent systems. We have been inspired a lot from the ideas and methodologies in this area. Secondly, we review some prominent work in developing MOUTbots which is similar to our research aims. Lastly, discussions are given on some off-the-shelf commercial game engines which can be used as our simulation platform.

Chapter 3 illustrates the organization of the MOUT simulation system and our AI framework design. A layered framework is proposed and detailed description of each module is given respectively. This is the key contribution of our research work.

Chapter 4 focuses on the implementation of our framework with regard to different algorithms and representations. The major benefits of our implementation have been identified.

Chapter 5 gives illustration on the simulations. Overall, realistic and robust bot behaviors have been generated through various simulations, among which two scenarios are presented.

Finally, Chapter 6 draws conclusions of our research and discusses some potential directions for future work. Though modeling human behaviors in MOUT simulations involves

a number of open questions that are still unsolved, our framework and the resulting implementation can be used as a solid basis for further efforts.

CHAPTER 2: LITERATURE REVIEW

2.1 Overview

Research on designing intelligent agent and multi-agent systems has been around since the early 1970s which is mainly focusing on theoretical proving of variant tools, techniques, and algorithms. As an application area of intelligent agent system, MOUT simulation is focusing on generating robust, efficient, and realistic bot behaviors in the urban warfare. Each bot in the simulation can be modeled as an agent that can sense, think, and act in the virtual world. Therefore, we believe that a lot of ideas can be inspired from techniques and methodologies developed in designing intelligent agent and multi-agent systems. Moreover, in recent years more and more commercial game engines as well as various tools for developing bots have been made available to bot designers, which greatly promotes the research on MOUT simulations. In Section 2.2, we investigate some key issues in designing intelligent agent and multi-agent systems. Section 2.3 reviews some prominent work on developing bots in the games. In Section 2.4, we discuss some off-the-shelf commercial game engines that could be used as our simulation platform.

2.2 Intelligent Agent and Multi-agent Systems

The goal of designing intelligent agents “is to better understand the principles and organizations that underlie adaptive, robust, effective behavior and to develop tools, techniques and algorithms for constructing autonomous agents that embody these principles and organizations” [Maes, 1995]. Typically, an agent should be able to sense the surroundings, reason upon its perception, and act according to its decisions, which

comprises a loop between the agent and the environment it resides. Broadly speaking, it has two kinds of application fields, building autonomous robots and constructing software agents. A well-known application of agent in robots is robotic soccer. Robotic agents are organized into teams that play soccer against each other. Software agents are widely used in areas such as controlling manufacturing process, transportation management, air traffic control, telecommunications systems, etc. The key issues of designing intelligent agent are action selection, learning, as well as cooperation and coordination with respect to multi-agent systems.

2.2.1 Action Selection

The problem of action selection is to make the decision, i.e. what to do next, based on the agent's knowledge. Generally, the process of action selection may be tightly coupled with deliberation, reasoning and planning. Tyrrell [Tyrrell, 1993] investigated some different action selection mechanisms and evaluated in terms of their respective computational properties. However, his evaluation is based on a particular simulated environment and a set of specific behaviors. In this section we discuss three typical action selection mechanisms, called deliberative model, reactive model, and hybrid model, respectively. The agents designed based on these models are deliberative agents, reactive agents, and hybrid agents, respectively.

2.2.1.1 Deliberative Agents

The classical approach to constructing intelligent agents is to view them as a particular type of knowledge based systems. This approach is based on the traditional AI techniques, such as planning and reasoning. A deliberative agent often has a symbolically represented model

of the world, and it selects actions based on the model by logical reasoning or searching its plan space.

The first well-known planning system STRIPS (STanford Research Institute Problem Solver) was developed by Richard E. Fikes and Nils J. Nilsson in the early 1970s [Fikes and Nilsson, 1971]. STRIPS describes the problem space as the initial world model, the set of available operators, and the goal statement. Each of the operators has an add list, a delete list, and a list of preconditions. The add list and the delete list specify those propositions that will become true or be no longer true respectively in the resulting world state by executing the operator. The list of preconditions specifies the conditions under which the operator is applicable. The planning process can thus be seemed as the attempt to find a series of operators that can transform the initial world state into the given goal state.

Although STRIPS is proved to be ineffective on problems of even moderate complexity because of its simple planning algorithm, it inspired researchers and designers in the following years and much effort has been devoted to developing deliberative agents in this trend. One of the most promising research directions that have been explored since the late 1980s is the modeling of agents based on beliefs, desires and intentions, called BDI models [Bratman, 1987]. The basic idea behind the BDI model is to emulate the mental processing procedure of human beings by a set of mental attitudes. More details of BDI models can be found in [Bratman et al, 1988; Rao and Georgeff, 1991; Rao and Georgeff, 1992; Rao and Georgeff, 1995].

2.2.1.2 Reactive Agents

Although the approach to developing deliberative agents has thrived since the early 1970s, researchers encountered a lot of problems in practical applications. “Only poor results have been obtained in its application to research on autonomous agents. The few systems built

show deficiencies such as brittleness, inflexibility, no real time operation, and so on. They also spawned a number of theoretical problems such as the frame problem and the problem of non-monotonic reasoning which remain unsolved (at least within realistic time constraints).” [Maes, 1990]

A number of researchers began to believe that agents constructed by the deliberative model would not be able to deal with urgent situations in real time, thus radically different models should be developed to solve the problem. Several new models have been proposed since mid-1980s. Typically, they do not hold a symbolic world model and do not use symbolic reasoning to select actions. On the contrary, they have a very limited amount of the world information and their action selection is directly based on the sensory input. Such agents intend to produce prompt and robust actions in response to the dynamic environment, whereas do not guarantee optimal or correct actions. Moreover, the seemingly goal-directed activity is actually modeled as an emergent property of the interaction among the internal modules and between the modules and the environment. The idea behind such models is that the complexity of the behavior produced by an agent may be a reflection of the complexity of the environment in which it operates rather than a reflection of the agent's complex internal design [Simon, 1981]. [Maes, 1990] collected a number of papers by the authors who had put a lot of effort in this approach.

A very well-known reactive model is the *subsumption* architecture proposed by Rodney A. Brooks in 1986 [Brooks, 1986]. In contrast to the traditional decomposition of a mobile robot control system into functional modules such as perception, modeling, planning, task execution and motor control, the *subsumption* architecture decomposes the control system based on task-achieving behaviors such as avoiding objects, wandering, exploring, building maps, etc. The modules are organized into a series of incremental layers and each layer connects perception to action, which means that they can be active in parallel. In addition,

each layer has a level of competence and a higher level of competence implies a more specific desired class of behaviors, i.e. higher levels of competence provide additional constraints on the lower ones.

Modules that compose each layer are finite state machines augmented with some instance variables. Each of them has some input lines and output lines that connect the modules with each other. Control is based on the substitution (*suppression*) signals on the input lines and removal (*inhibition*) signals on the output lines of the lower level modules by the higher level modules.

In a later paper [Brooks, 1990], the author introduces some robots built up on the *subsumption* architecture. These robots exhibit successful abilities to achieve simple tasks such as wandering around while avoiding collisions, exploring the world while building the map and path planning, etc. The importance is that these seemingly goal-directed behaviors are actually emerged from modules' direct interaction with the environment in parallel with no central repository and no explicit reasoning. However, the *subsumption* architecture has some disadvantages. The obvious one is it cannot produce complex behaviors by an architecture with many layers because the interaction between these layers will be too complex to understand. Moreover, the architecture is application specific. Once the application is changed, the model should be re-designed.

2.2.1.3 Hybrid Agents

As stated above, both deliberative agents and reactive agents have their own advantages and disadvantages. The pure deliberative agents are elegant in logical proving and their foundation AI has been developed for decades, hence they are feasible for achieving complex tasks. But once they are put in the complex and highly dynamic environment, they are often inept at dealing with the problems in an acceptable time period. On the contrary,

while the pure reactive agents can produce prompt and robust actions, they are difficult to understand and construct, and can only achieve some simple tasks. Therefore, a natural idea is to build hybrid agents based on the combination of these two approaches. They should have a reactive subsystem to deal with urgent situations and also a deliberative subsystem to produce goal directed and long term behaviors. Typically, such an idea leads to layered architectures.

In this section, we investigate two typical layered architectures called TouringMachines and **IntERRAP**. TouringMachines is a horizontally layered architecture which is developed to construct rational, autonomous, resource-bounded mobile agents that can carry out complex and time-constrained navigational tasks in a dynamic environment [Ferguson, 1992]. The architecture is illustrated in Figure 2.1. It consists of three control layers: a *reactive layer*, a *planning layer*, and a *modeling layer*. Each of the layers operates concurrently, receives sensory information from the perception subsystem and sends action commands to the action subsystem independently.

Figure 2.1: The Architecture of the TouringMachine

The reactive layer is designed to compute hardwired, domain-specific action responses to particular sets of environmental stimuli. Therefore it can deal with the unexpected and urgent events promptly. The planning layer is designed to generate and execute plans to achieve long-term domain tasks that is relocating to a given target location within certain pre-specified spatio-temporal constraints. Plans are stored as partially elaborated, procedural and hierarchical structures in the plan library. The planner of the planning layer makes use of the library to generate linear route plans based on the topological map and the path junctions of the world. In addition, the plan formation and execution is interleaved so that it can cope with certain unexpected failures. The modeling layer is designed to solve the intra- and inter-agent conflicts by goal coordination. The purpose of the modeling layer is to detect and foresee such potential situations that cannot be solved by the planning layer and propose suitable courses of actions to deal with. The three control layers are embedded in a control framework that consists of the *perception subsystem*, the *action subsystem*, a facility for *inter-layer message passing*, and a set of *context-activated control rules*. Especially, the control rules act as filters to resolve inter-layer conflicts both between the perception subsystem and the control layers, and between the control layers and the action subsystem. The function of the control framework is to mediate the interactions between the layers, thus ensure that the TouringMachines can behave appropriately in different world situations.

In comparison with Ferguson's horizontally layered architecture, Müller takes a different approach to construct agents with a vertically layered architecture called **INTERRAP** (Integration of Reactive Behavior and Rational Planning) [Müller and Pischel, 1993; Müller, 1996]. The architecture is illustrated in Figure 2.2.

Figure 2.2: The Architecture of **INTERRAP**. SG: situation recognition and goal activation. PS: planning, scheduling, and execution.

The architecture consists of a World Interface, a Knowledge Base, and a Control Unit. The World Interface provides perceptual, communicative and motoric links to the environment. Both the Control Unit and the Knowledge Base are layered into three levels: the Behavior-based Layer, the Local Planning Layer and the Cooperative Planning Layer corresponds to the World Model, the Mental Model and the Social Model respectively. The Behavior-based Layer is designed to achieve reactivity, the Local Planning Layer uses means-ends reasoning to produce goal-directed behaviors to achieve its tasks, and the Cooperative Planning Layer is responsible for social interaction. Correspondingly, the World Model holds its beliefs about the environment, the Mental Model holds the beliefs about itself, and the Social Model holds the beliefs about the other agents. In addition, each of the control layers can access its corresponding layer's as well as its lower layers' knowledge base, but it cannot access the higher layers' knowledge base. Furthermore, each control layer is partitioned into two processing units called SG (situation recognition and goal activation)

and PS (planning, scheduling and execution) according to their respective functions. The characteristic of the architecture is that control is activated in a bottom-up fashion whereas execution is activated in a top-down manner. As illustrated in Figure 2.2, control is triggered by the SG of the Behavior-based Layer and shifted upward by the PSs of the Local Planning Layer and the Cooperative Planning Layer once they find they are not competent to deal with the situation. On the other hand, execution is activated directly from the PS of a higher control layer to the PS of a lower control layer.

The main difference between the TouringMachines and the **INTERRAP** is the control methods of interaction between the layers. In TouringMachines, each of the layers is directly linked with the perception subsystem and the action subsystem, and they can act independently. Therefore a global control framework should be employed to mediate conflicts among the layers. However, it is difficult and tedious to consider all of the possible ways of their interaction at the design time. The **INTERRAP** architecture partially solved the problem by the constraint that only the Behavior-based Layer can interact with the World Interface, but it is still problematic when the control should be shifted to a higher layer. Moreover, another problem associated with the **INTERRAP** is that passing of control and execution between the layers may induce additional computational costs and potential error transmission.

2.2.2 Learning

In the previous section, we have discussed the action selection problem, the goal of which is to autonomously generate actions to achieve goals and in response to the dynamic environment. However, the hard-wired action selection mechanism is not sufficient for an intelligent agent due to the unpredictable changes of the environment. In other words, it is impossible to design an action selection mechanism that can cope with all unpredictable situations, especially in a competitive environment. Moreover, an action selection

mechanism considered as optimal at the design time may turn out to be sub-optimal or even non-optimal because of some changes of the environment. Therefore, another essential characteristic for an intelligent agent is the ability to learn or adapt from experiences so that its action selection mechanism can improve over time.

Machine learning is a sub-field of AI and a number of learning techniques and methodologies have been developed for decades including decision tree learning, artificial neural networks, Bayesian learning, instance-based learning, genetic algorithms, reinforcement learning, analytical learning, etc. These methods can be classified as online learning methods and offline training methods. For example, decision tree learning and artificial neural networks are often used as offline training methods, while reinforcement learning is getting more and more recognition as an online learning method.

Blumberg [Blumberg, 1997] developed a computational model to build and control animated interactive creatures. Silas, a virtual dog, has been implemented based on this model, which can respond to the user by a set of gestures and postures through learning. Specifically, the dog can learn in the following three ways: learn associations between an existing behavior and some previously unassociated goals in a novel context; learn associations between a new releasing mechanism and an existing behavior; and learn more appropriated association between a behavior and its releasing mechanisms. The learning process is triggered by a radical change of the internal variables, i.e. the feedback of its actions, and each internal variable is responsible for its own learning. Basically, it is a kind of reinforcement learning using temporal difference model. As an improvement, the approach proposed is not state based so as to make it suitable for real life problems.

Robotic soccer has provided a fertile platform to model learning agents in the recent years [Tambe et al, 1999; Stone et al, 2000; Stone and Sutton, 2001; Yao et al, 2001].

Researchers construct robotic agents or software agents to take part in the RoboCup competition. The ability to learn and adapt is very critical for the agents because of the highly dynamic and unpredictable environment. Stone [Stone, 2000] proposed a model for layered learning. The basic idea is that direct learning from the input to the output is not tractable in such a complex domain. However, the whole task can be easily decomposed into a bottom-up hierarchy. Therefore, it is desirable to perform learning in each of the levels and feed the results of the lower level learning into the higher level as its input. More specifically, in the lowest level, each agent learns an individual skill to intercept the ball using neural network. Since such a behavior is hard to be coded by hand, but once it is learned it need not be refined during the play, neural network is suitable to train the skill offline. The second level is learning a multi-agent behavior to evaluate whether the ball can be passed to a particular teammate. Because such a behavior involves numerous attributes, using decision trees constructed by the C4.5 algorithm [Quinlan, 1993] to train the skill offline is suitable. Ultimately, a Q-learning-like reinforcement learning method called TPOT-RL (Team-Partitioned, Opaque-Transition Reinforcement Learning) is developed to learn a team behavior online to select a specific teammate, to whom the agent can pass the ball. While the input space is too huge to directly employ Q-learning, TPOT-RL successfully uses previous learned decision tree to drastically reduce the input space. The layered approach gracefully integrates several learning methods into the agent model and achieves good results in the RoboCup competition.

2.2.3 Multi-agent Systems

Action selection and learning are two key issues in designing self-contained intelligent agents. In this section, we briefly discuss the issues in multi-agent systems and review some work in this area.

The development in designing multi-agent systems is closely related to the progress of distributed artificial intelligence (DAI) that is incepted in the late 1970s and has been prosperous in recent years. Generally, there are two problems that multi-agent systems should solve:

- 1 How to coordinate the knowledge and behaviors among the agents, and how to reason about the coordination (especially cooperation and competition).
- 2 How to decompose the whole task into sub-tasks and how to synthesize the results, which is traditionally seemed as the issue of distributed problem solving.

The solution of the above problems is based on interaction that takes place directly among the agents by a shared language or indirectly through the world they are embedded. Thus it leads to a more specific question that is how interaction affects the process of deliberation, planning, reasoning, action selection, and learning of each individual agent. [Weiss, 2000] has much more detailed description of these issues.

A simple but well-known example is Reynolds' boids model [Reynolds, 1987]. Boids are an aggregation of simulated birds navigating in the virtual world. In contrast to the traditional animation that the animator should script the paths for each of the boids individually, Reynolds endowed each of the boids with independent ability to find its own way in the navigation. Specifically, each boid has its own local perception, the laws of simulated physics and a set of behavioral rules to obey. The simulated perception is not intended to emulate the real animals, instead it is implemented as a spherical zone of sensitivity centered at the boid's local origin and any objects in the zone can be detected by the boid. There are three behavioral rules: Collision Avoidance is to avoid collisions with nearby flockmates, Velocity Matching is to match velocity with nearby flockmates, and Flock Centering is to stay close to nearby flockmates. These rules are strictly prioritized so that it

is easy to arbitrate conflicting behaviors. Therefore, once the scene is set, the boids can simply animate themselves without the need for animator's intervention and the flocking behavior is emerged by interaction of the simple behavioral rules of each individual boid. In the view of designing intelligent agent, the novel ideas presented by Reynolds are distribution and reaction. Distribution means that there is no central control and each of the actors can think about for itself. Reaction means that each actor can promptly take actions in response to the dynamic environment by its local perception. A later version of the boids model is the steering behaviors [Reynolds, 1999].

As for more complicated multi-agent model, Haddadi presents a formal yet pragmatic theory for reasoning about actions and interactions. Especially, interactions are regarded as communicative actions, which are incorporated into the general process of reasoning about actions. The formalization of the theory is based on the BDI logical framework as illustrated in Section 2.2.1.1. To enable agents to interact, the formalism is extended to facilitate forming joint commitments. In order to make it more practical, a convention is introduced to specify the process that includes three reasoning states: *potential for cooperation*, *pre-commitment* and *commitment*. In particular, the sets of rules for forming goals to communicate are specified. These goals are actually sub-goals formed dynamically in the context of task delegation and task adoption, and then fed back into the reasoning process to find appropriate plans to satisfy them. More details can be found in [Haddadi, 1996].

2.3 Bots in Games

Thanks to the advent of more powerful PCs and more modifiable FPS games, quite a number of bots have been developed both by academia and by bot amateurs in the past several years. In this section, we review some prominent work in this area. The design of

these bots varies from very simple ideas to fairly complicated architectures. However, the bottom line is that these bots can behave realistically in the virtual world in real time.

2.3.1 Neuralbot

The Neuralbot is designed as an automated Quake2 deathmatch opponent in an almost pure bottom-up manner, i.e. there is no hard-coded knowledge or rules embedded for planning or reasoning purposes [Chapman, 1999]. Instead, a fully connected artificial neural network is used to control its actions and a genetic algorithm is used to evolve the neural network. The input of the neural network is the sensory information including environmental information and enemy status, as well as the status of the bot itself. The output of the neural network is the actions the bot takes corresponding to the input, such as selecting a specific weapon, looking at a specific direction, etc. Each Neuralbot has a neural network with the same topology which will not be changed through learning. However, different bots have a different set of synapse weights encoded. The genetic algorithm is used to adjust the weights of the neural network so as to increase its fitness, i.e. the ability to get flags.

The Neuralbot is a good example on its bottom-up approach. However, such a bot is unable to carry out a more complicated task in an MOUT scenario. Therefore, it is necessary to incorporate some top-down structures in order to simulate more realistic bot behaviors.

2.3.2 Quake III Arena Bot

The Quake III Arena bot is an artificial player emulating a human player in the Quake III Arena environment [Waveren, 2001]. The bot can either act as a single player opposing to other players or act in a team play cooperating with other players.

The Quake III Arena bot is designed on a layered architecture. There are four layers altogether. The first layer, i.e. the bottom layer, gets sensory information from the game world and outputs bot actions. The second layer is responsible for making low level decisions, such as selecting goals, navigating towards a goal, interpreting chats, etc. The third layer provides higher level reasoning and decision-making functionality, such as fighting against opponents or navigating through mazes. The fourth layer, i.e. the top layer, is exclusively for the team leader to organize the team so as to achieve some tasks. Basically, knowledge of the game world is formed and collected in the lower layers and propagated to higher layers. On the contrary, decisions are made in the higher levels and propagated to the lower layers.

A feature of the Quake III Arena bot is its Area Awareness System (AAS) which provides all of the sensory information of the virtual world for navigation and routing purpose. AAS represents the game world using 3D bounded hulls, called areas. The reachabilities of these areas are pre-calculated to minimize the computational cost at runtime. A reachability between two areas means that a player can travel from one area to the other area. Therefore, routes between areas can be calculated by using the areas' descriptions and reachabilities.

2.3.3 Soar Quakebot

John Laird and the research group at the Michigan University developed the Soar Quakebot that can play Quake II deathmatches with or against human players [Laird and Lent, 1999; Laird, 2001]. The Soar Quakebot uses the Soar architecture as its underlying AI engine and interacts with the Quake II game engine using the Quake II interface DLL through sockets. Soar is a general cognitive architecture that can be used as an inference engine for intelligent agents [Laird et al., 1987]. In the Quakebot, all the knowledge for playing the game

is encoded in Soar rules. Soar receives sensory information of the game world ten times a second and makes decisions for execution asynchronously. Basically, Soar organizes its operators into three levels. The top level operators represent the Quakebot's goals or modes of behavior, such as "Attack the enemy" or "Explore the map". The middle level operators represent the high level tactics the Quakebot uses to achieve the top level goals, such as "Circle-Strafe", "Snipe", or "Chase". The bottom level operators are the steps and sub-steps, called behaviors such as "Shoot target" or "Find hidden location", used by the agent to implement the tactics. In each decision cycle, Soar elaborates a high level goal to specific Quakebot actions.

In [Laird, 2001], anticipation is integrated into Soar's decision making process. First of all, the Quakebot creates an internal representation that mimics what it thinks the enemy's internal state is, based on its own observation. It then uses its own knowledge of tactics to predict the enemy's behavior. The Quakebot determines whether the prediction is useful or not based on certain rules. Finally, the prediction is used to set an ambush or prevent the enemy from obtaining an important weapon or health item through Soar's decision making cycle.

2.4 Off-the-shelf Game Engines

A computer game usually includes the game engine, the virtual game worlds, the game code, and the networking code. The game engine is the core of the whole game. It handles 3D rendering and generic physics/dynamics for the virtual game world. Normally the code of the game engine is not open to any users. The virtual game world can be shipped with the game or authored by a third party using the game's editing tool. A popular game engine usually has a variety of game worlds available. The game code specifies the basic

mechanisms of the game, like the game rules, the basic actions of each character, etc. The networking code supports server-clients architecture, i.e. one computer hosts the game and the others connect to the same game as clients through the network.

Generally, there are two approaches to building up an MOUT simulation system. One approach is to design and implement the whole system completely. The other approach is to implement a separate AI engine while using an off-the-shelf game engine to handle other features of the game [Biddle et al, 2003]. While the first approach may take more time and effort, it is more flexible, i.e. the system can be specifically designed to meet various AI specifications. The reason to take the second approach is that there are quite some off-the-shelf commercial game engines which are affordable and provide excellent support for high-fidelity 3D virtual environments. The main limitation of this approach, however, is that the implementation of the AI engine should conform to the API of the specific game engine. In this section, we review three possible approaches to using an off-the-shelf commercial game engine as the simulation platform as well as some prominent FPS games which are the possible choices for MOUT simulations.

2.4.1 Approaches

Alex J. Champandard has an article discussing the issues of using off-the-shelf game engines in detail [Champandard, 2003]. Basically, there are three approaches to using an off-the-shelf game engine as the simulation platform and each approach has its own pros and cons. The first approach is the server side approach. In order to take this approach, we need to have the source code of the target game. We can modify the source code, as we like, to create our own game DLL that is loaded by the game engine. The advantage of this approach is that we have great flexibility to modify the game, such as adding our own bots, in variant aspects. The disadvantages of this approach are that, on the one hand, we need

to understand the overwhelming source code in detail, which may take quite some time, and on the other hand, the approach lacks fault tolerance that means any bug or mistake may cause the server to crash.

The second approach is the game interface approach. We can take this approach when the game source code is not available but we want to modify the game at a game DLL level as well. The approach is to build an intermediate interface DLL sitting between the game engine and the game DLL so that we can run our AI algorithms coded in the interface DLL by intercepting the calls between the game engine and the game DLL. An efficient way to take this approach is to find an existing skeleton code for the interface DLL, and insert our AI code where needed.

The third approach is the client side approach. For some games, there exist interface programs responsible for communication between the game server and the clients. In this approach, we can write a stand-alone bot program as a client communicating with the server through the interface. The difference between this approach and the above two approaches is that we can run the bot code independently on another machine and connect it to the server remotely over the network. This approach also simplifies our task which means we only need to understand the interface between the server and the clients instead of investigating the game source code. However, the disadvantage is that it provides less flexibility for us to make game modifications and we cannot get some specific information in the game engine level except those provided by the interface.

2.4.2 Possible Choices

- **Quake Series (Quake, Quake II, and Quake III Arena)**

Quake series games, especially Quake II, have been the popular choices for bot developers due to the fact that id Software has made the full source code freely

available for game modifications. Especially, there are quite a number of bots built for Quake II and great source code support available from a wide community as well. We can either take the client side approach or the server side approach. There are interface DLLs written for communication with the Quake II server available. One good example is Ben Swartzlander's Quake 2 Bot Core which can be utilized for building bots in a client side approach [Swartzlander, 1999]. Quake III is also a good choice. One advantage of Quake III is that it has an Area Awareness System (AAS) developed based on Binary Space Partition (BSP) tree which is quite good for navigation purpose compared to waypoints systems used in other FPS games, although we can only access it by an interface since the code is wrapped up inside a library. [Waveren, 2001] has a detailed illustration of the Quake III AAS. However, the problem of the Quake series games is that the engines are written in C, which greatly increases the complexity of developing bot projects on a large scale.

- **Half-Life**

Half-Life is a FPS game developed by Valve Software based on the code of the original Quake with a number of new features added. A research group at the department of computer science, Northwestern University, has developed a sensor & actuator interface DLL for the Half-Life server called FlexBot [Khoo et al., 2001]. Moreover, they developed some behavior-based bots for demonstration which can generate a set of prioritized behaviors according to different situations. Besides, there is a bot-framework available by Botman based on the game interface approach [Broome, 2002].

- **Unreal Tournament**

Unreal Tournament (UT) is another FPS game released by Epic which is getting more and more recognition because of its great modifiability and expandability [Muñoz-Avila, 2004; Wray et al, 2004]. It is viable either to generate a completely new game world using its shipped editor or even design a completely new game type other than FPS

games, such as UnrealSpeed, a racing modification of UT 2003 [UnrealSpeed Team, 2003]. The designers of Unreal Tournament have developed UnrealScript [Sweeney, 1998], a built-in object-oriented scripting language. UnrealScript provides a high level interface to the UT server which makes authoring game modifications much easier. The MOVES Institute has used the Unreal engine to develop the 'America's Army' game. John Laird and the research group at the University of Michigan are also shifting to UT for more intelligent characters as well as more interesting games [Magerko et al, 2004]. The game they created, called Haunt 2, is an adventure-style game instead of a warfare game.

Various tools have been developed for the development of AI bots using UT engine. A research infrastructure, called Gamebots, has been jointly developed by USC and CMU [Adobbati et al., 2002]. As UT is a multi-player game with the generalized client-server architecture, Gamebots acts as an interface between the server and the clients. Sensory information such as the location and rotation of a player in the game world or a message received from a teammate is sent from the server to the clients by synchronous messages and asynchronous messages, whilst bot action commands such as run to a specific position or change the weapon are sent back to the server from the clients. Furthermore, in addition to Gamebots interface Andrew Marshall at USC-ISI created a higher-level interface based on the Gamebots protocol called JavaBot API [Marshall et al., 2002]. It handles the specific Gamebots protocol, network socket programming, message passing and other related issues, which makes the work to develop bots AI neater and simpler. We will give more details in the Chapter 3.

CHAPTER 3: AI FRAMEWORK DESIGN

3.1 Overview

An MOUT simulation system comprises two major parts, a simulation engine and a bot AI engine. The focus of this research project is to design an AI framework for the MOUTbot which is supposed to be flexible, extensible, integrable, and independent of simulation platforms. For fast prototyping, the AI framework is implemented on the Unreal Tournament (UT) game engine. We introduce the organization of our simulation system in Section 3.2 and illustrate our AI framework design in Section 3.3.

3.2 MOUT Simulation System

3.2.1 System Organization

Although our AI framework is designed to be independent of simulation platforms, we have implemented the framework on Infiltration [Sentry Studios, 2003], a modification of the Unreal Tournament game. The benefits of using Infiltration are as follows:

- The Unreal Tournament game provides excellent support for high-fidelity 3D virtual environments. A variety of maps are freely available and we can also easily design our own maps using the Unreal Editor shipped with the game.
- Various tools have been developed for the development of AI bots using the UT engine, such as the GameBots API and the JavaBot API, which simplify the implementation of AI algorithms.

- Infiltration replaces futuristic players/bots in the original UT with modern soldiers that can walk, jog, sprint, crouch, prone, lean around corners, and use a number of more realistic modern weapons. Its enhanced realism meets our requirements of MOUT simulations quite well.

The organization of our simulation system is illustrated in Figure 3.1. Basically, there are four parts. The UT Server runs the game engine. The GameBots API [Kaminka et al, 2002; Adobbati et al, 2001] and the JavaBot API [Marshall et al, 2002] are interfaces through which our MOUT bot AI engine communicates with the UT server. They are freely available on the web. Description of GameBots and JavaBot will be given in Section 3.2.2 and Section 3.2.3 respectively. MOUT bot AI engine is the brain of the bot that is the focus of this research project.

Figure 3.1: System Organization

The GameBots API provides socket-based message-passing interface to the UT server. Therefore, the whole system can be run on a client-server architecture. Multiple bots can run on different computers as clients and connect to the same game world.

3.2.2 GameBots API

The GameBots API was jointly developed by a group of researchers from the University of Southern California and Carnegie Mellon University. It provides a platform that allows creation and evaluation of bots in the UT environment. The main purpose of developing the

GameBots infrastructure is to facilitate research in the areas of multi-agent systems and artificial intelligence using the multi-player FPS game, Unreal Tournament, as the simulation platform.

The core of the GameBots API is the functionality of communicating with the UT server via network sockets. It receives sensory information from the UT server and sends it to the bot clients. Meanwhile, it receives action commands and player communication messages from the bot clients and sends them back to the UT server for execution. The sensory information includes synchronous messages and asynchronous messages. Synchronous messages indicate the state of the game world at a specific moment like the bot's own state, the list of items and players within the bot's view, and the status of the game. Asynchronous messages indicate the events happened between the synchronous messages, like bumping into a wall, hearing a noise, receiving a message from a teammate, etc. Action commands include bot actions like running to a specific location, firing the bot's weapon, stopping all movement or rotation, etc. Table 3.1 gives the summary of the network messages that have been used in our project.

The GameBots API was written in UnrealScript, an integrated scripting language that handles all game logic and object interactions under Unreal Tournament. The source code is freely available on the GameBots website. Therefore, it is possible for us to modify the original code according to our application specifications, which provides sufficient room for variant bot applications. For example, the clients can easily get additional information that can be extracted from the game world via the GameBots API as required.

can be faced with different tasks in variant scenarios, which facilitates continuous, long-term research.

Network Messages	Description
Synchronous	
BEG	Begin a synchronous block
SLF	Information on the agent's state (Situational awareness)
INF	Information on enemy or teammates (Situational awareness)
PLR	Another visible player in the game seen by the agent
NAV	A path node in the game used for navigation purposes
MOV	A "Mover". Can be doors, windows or switches
INV	Items on the ground that can be picked up (e.g weapons)
END	End of a synchronous block
Asynchronous	
NFO	Information about the game after connecting to the server
AIN	Added inventory.
VMG	Received personal message from another player
CWP	Changed weapon
WAL	Collided with a wall
FAL	Hit a ledge, about to fall
BMP	Bumped into another player
HRN	Heard noise like footsteps or shooting
KIL	A player killed
DIE	This agent died
HIT	Hurt another player
DAM	Took damage
Commands	
INIT	Used to spawn an agent
STOP	Stop all movement
JUMP	Jump
CHANGE WEAPON	Change the weapon
SHOOT	Shoot at target
RUNTO	Run to a certain location
TURNTO	Turn to a direction or player
STRAFE	Move sideways while facing another

Table 3.1: GameBots Network Messages and Descriptions

In addition to the communication protocol, there are some tools developed as extensions such as visualization tools and data collection tools. The visualization tools provide a bird's eye views of the map and display the real-time movements of the bots. The data collection tools are used to generate parsable text logs at the server side as well as at the client side.

3.2.3 JavaBot API

The JavaBot API is a higher level interface to the GameBots API to further facilitate bot development. It is initially created by Andrew Marshall at the University of Southern California and later extended by a group of student from the University of Texas A&M. Currently, it is a open source project maintained at Sourceforge and could be freely downloaded from the website.

The JavaBot API handles the specific GameBot protocol, network socket programming, message passing and other pragmatic issues, which makes the work to develop bots AI neater and simpler. In other words not much knowledge of the specifics of the GameBots protocol is necessary to implement a bot, what we need to do is to implement the bot interface in the JavaBot package. As for the other Java based applications such as visualization tools or analyzing tools, they can also be built on top of the JavaBot platform conveniently. Therefore, JavaBot API is suitable for us to develop our bot AI. The diagram in Figure 3.2 illustrates how different parts fit together.

Figure 3.2: JavaBot API

3.3 AI Framework

3.3.1 Design Guideline

The AI framework is the brain of the MOUTbot which senses the world and decides what to do next, while the game server controls the body of the bot to execute any action commands from the brain. In a MOUT simulation, usually a bot should play two roles. On the one hand, it should act as a self-contained soldier. That is to say, each bot should be able to carry out a task independently. It should behave in a goal-directed way so as to achieve the task. Meanwhile, it should react to urgent situations immediately so as to avoid any injuries or damages. More importantly, all its behaviors should make sense and be as realistic as possible. On the other hand, in MOUT simulations tasks are usually carried out by squads. Each squad may consist of several, say three or four, soldiers. A well trained soldier should move coordinately and act cooperatively with his/her teammates through appropriate interaction. Thus, the AI framework should also facilitate such interactions.

Moreover, we believe developing an AI framework for MOUTbot is an incremental process. A full-fledged brain may consist of a number of modules in a very complex architecture. Therefore, it is hard to determine the framework at the very start. We shall start the project with a solid foundation in a small scale and increasingly expand the framework by adding variant modules.

Furthermore, ultimately all of the modules should work as an integral system. We need to carefully determine the functionality of each module and clearly define the interfaces between each other. The whole system should run as efficiently as possible, i.e. its computational cost should not be too high.

Overall, the AI framework should meet the following requirements:

- Flexible: the bot can take different actions according to different surroundings; it can not only respond promptly to the urgent situations but also carry out a task smartly so as to achieve a goal alone or in collaboration with the teammates.
- Extensible: various new features could be added into the framework without much difficulty.
- Integrable: different behaviors, different levels of reasoning process should be integrated into a consistent hierarchy.

3.3.2 A Layered Framework

Our AI framework has been designed to meet the above requirements. Figure 3.3 shows the conceptual model of the framework.

Basically, the framework has a layered architecture which has the following major benefits:

- The architecture reflects the natural pattern of human's decision making process. Low level decisions, such as coordinating movement among the bots in the same group, correspond to reactive behaviors and are responsible for immediate handling of urgent situations. High-level decisions, such as scheduling different tasks in the mission while taking warfare strategies into consideration, correspond to deliberative behaviors and are responsible for achieving long term goals.
- The layered architecture is very flexible and extensible provided that the function of each layer as well as their interfaces to neighbouring layers has been clearly defined. In this case, different decision-making algorithms and knowledge representations can be employed in each layer without affecting the other layers.

Figure 3.3: The Conceptual Model of the AI Framework

Specifically, there are four layers in the hierarchy. The first layer or the bottom layer is the Group Coordination Module which coordinates the bots movement in the same squad. They are supposed to move as a group within immediate distance to each other so that each soldier's movement should take the other teammates into consideration. The input of this module is a desired path, and the output is specific locations in the world that a bot moves to. The calculation is based on the path, the group formation, and the positions of its teammates. The bot regularly updates the information of its teammates as the World Status.

The second layer is the Path-finding Module which is to find a path in the virtual world leading to a desired location. This layer basically makes use of the Terrain Database to find

an appropriate path. Terrain Database stores static information of the terrain. The representation may vary from one model to another. For example, the terrain can be represented as grids, polygons, waypoints, or even 3D bounded hulls as used in the Quake III Arena game. In addition, the path calculation may also include some tactical information, like positions of enemies or threats, sniping spots, covering spots, etc. Therefore, the output path may not be the shortest one or fastest one. Instead, it may be desirable for the bot to take a detour to avoid the enemies.

The third layer is the Bot Behaviors Module. Each behavior can be regarded as a sequence of basic actions. An action, in turn, is an animation sequence that controls the body movements of the bot. Actions include movement actions and other warfare actions, such as shooting or throwing a grenade. Bot behaviors schedule these actions smoothly so as to achieve certain objectives in a task reflecting certain warfare tactics. For example, when a squad plans to enter a room, one bot or two among them may need to secure the entrance of the room first. Here “secure_room” is a behavior. Therefore, the set of bot behaviors can be seemed as the building blocks of MOUT simulations. The input of this module is the objective of a behavior, and the output is the specific actions representing the behavior. The module refers to the Tactics Library for appropriate calculation. In order to make the behaviors reusable in different scenarios, they could be generic by use of a set of parameters indicating different situations. Therefore, each behavior may generate different actions according to the specific parameter settings.

The highest layer in the hierarchy, the Task Scheduling Module, is responsible for scheduling bot behaviors appropriately so as to achieve certain task. The input of this module is the description of a specific task, and the output of this module is the desirable behaviors to achieve the task. The module represents the highest level reasoning or planning, and it takes warfare strategies into consideration.

Besides these four layers, there is a module called the Action Blending Module which blends different actions into an executable sequence. Each bot in the MOUT simulation may have more than one behavior running at the same time, and each behavior schedules a sequence of actions to take. The module is used to blend these actions into an executable sequence, i.e. some actions can be executed concurrently. The blending guideline is to ensure that there are no conflicts in the ultimate actions. A simple approach is to construct a table denoting the degrees of freedom (DOFs) of each action. Specifically, if two actions have overlapping DOFs, they should not be carried out concurrently and vice versa. For example, the bot can shoot at enemies while moving to a desired location. However, the bot cannot shoot at enemies while picking up something on the ground.

The next chapter gives implementation details of these modules.

CHAPTER 4: IMPLEMENTATION

4.1 Overview

This chapter gives details of our implementation of the proposed AI framework. As it is pointed out in Chapter 3, our framework has been designed to be flexible, extensible, and integrable, which means that various algorithms and representations could be employed to implement each module as long as their interfaces conform to our specifications. Therefore, it should not be troublesome to change the implementation of any module due to the advancement of this research project. Section 4.2, Section 4.3, and Section 4.4 illustrate the Bot Behaviors Module, the Path-finding Module, and the Group Coordination Module respectively.

4.2 Bot Behaviors Module

This Module can be regarded as a behavior reservoir. Each behavior is an independent program. Once its parameters are set, it outputs action commands correspondingly. Behaviors should be able to be added into or deleted from the reservoir easily. In addition, as the collection of the behaviors is the building blocks of the bot AI, the implementation should be as efficient as possible. We implemented the module using a set of hierarchical Finite State Machines (FSMs) which meets the requirements of the module's functionality.

FSM is a model of computation which consists of a set of states, a set of actions, an initial state, an input alphabet, and a transition function that maps input signals and current states to a next state. Computation starts in the initial state with an input signal. It transits to new

states according to the input signals and the transition function. These input signals can be triggered from the environment or from the actions produced in the states or in the transitions. Normally an FSM whose actions are produced from the states is called a Moore machine, while an FSM whose actions are produced from the transitions between the states is called a Mearly machine. Figure 4.1 shows a sample FSM we have implemented. Each rectangle represents a state in the FSM. It is a kind of Moore machine, i.e. actions are produced from the states. The arrows connecting the states in the figure denote transitions. For example in Figure 4.1, when the squad forms a group (a state transition triggered), each bot in the squad moves following their leader.

Figure 4.1: A Sample FSM

FSM is widely used both in commercial games and in the bot community. It has the following advantages:

- It is straightforward to design, i.e. to convert a logical concept to a specific FSM.
- It is simple to implement and there are various versions available that we can learn from.
- It is efficient in execution. Only the code for the current state needs to be executed, and normally the logic to determine the current state is simple.
- It is flexible to integrate different techniques such as fuzzy logic.

However, the problem with FSMs is its poor scalability. Firstly, the transitions between the states will be very complicated if there are many states in a single FSM. Secondly, once there is a change in the design it will be difficult to modify the FSM structure. Adding or deleting a state may lead to a significant change in all state transitions.

To address the problem, we take the following two steps. Firstly, the FSMs are organized into a hierarchy with no more than three levels. A higher level state corresponds to a lower level FSM. Those states that can not be further decomposed into a lower level FSM generate action commands. Secondly, the behaviors implemented using FSMs are strictly limited to low level behaviors which means the transition logic should not be too complicated. These two steps are to ensure that there are only a few states in each level so that the transitions among the states will not be mixed up. In addition, as we keep all of the FSMs in each level small it is not difficult to maintain or extend them.

Take the FSM in Figure 4.1 as an example, among all the five states, the 'Fight' state, the 'Flee without Covering Fire' state, and the 'Flee with Covering Fire' state can be further decomposed into FSMs in the lower level. Figure 4.2 shows the 'Flee with Covering Fire' FSM.

Figure 4.2: FSM for 'Flee without Covering Fire'

4.3 Path-finding Module

4.3.1 World Representation

Path-finding algorithms depend on the world representation. An ideal representation model should be able to give answers to any geographical queries of the virtual environment from the bot. The basic requirement is to differentiate between walkable/accessible areas and unwalkable/unaccessible areas. A more advanced requirement is to tell the bot the location and the dimension of every object in the virtual world, such as walls, corridors, doors, lifts, stairs, windows, etc. In addition, the more detailed information the representation model can provide, the better it is. However, practically we are not able to have such an ideal representation model. Instead, we need to evaluate the following issues to balance the pros and cons of different representations:

- Computational cost and memory cost: Neither too high computational cost or too high memory cost is acceptable. Some representation models may allow pre-

computation to significantly reduce the computational cost at run time, while others may be able to reduce computational cost by increasing its memory cost.

- Implementation simplicity: Different representation models vary a lot in the implementation simplicity. Some models may have tools available to automate the generation process, while some models may need designers to do a lot of manual work.
- Tactical feasibility: Good world representation should facilitate deployment of variant tactics. For example, the geographical information retrieved can be used to find spots for fire-covering or sniping.

In this project, we use grids to represent the virtual world. A grid consists of some evenly spaced points that the bot could reach. We call these points path nodes. Normally, grids can be square, triangular, or hexangular. We use square grids due to its simplicity. Currently all the path nodes are manually placed onto the map using the Unreal Editor, an editing tool shipped with UT. Figure 4.3 shows the path nodes in a rendered scene as well as in the Unreal Editor.

Figure 4.3: Path Node and Grid Representation

Usually an urban environment consists of a number of buildings and streets, and each building is composed of rooms, corridors, and stairs. Thus, the whole environment can be thought of as a spatial organization of different areas. Each area is connected to its neighboring areas by doors, stairs, entrances, etc. In the same way, our grid system is built up based on areas. Each area is represented by a grid, and different grids are connected by Transition Nodes put at the doors, stairs, and entrances. These Transition Nodes are special path nodes. Figure 4.4 gives a sample area-based grid representation.

Figure 4.4: Area-based Grid Representation

This area-based approach gives flexibility to distribute grids in each area with different resolutions as shown in Figure 4.4. Grid resolution can be used to reflect the terrain complexity. High resolution grids convey more detailed geographical information whilst result in higher computational cost and memory cost. For example, in out-door streets we may have lower resolution grids while in some in-door rooms where the bot may deploy a variety of tactics we may have higher resolution grids.

Grid representation also facilitates deploying warfare tactics. The simplest approach is to attach information onto the grids so as to notice the bot. Currently we have implemented Observing Nodes and Securing Nodes as shown in Figure 4.5 and Figure 4.6 respectively.

Figure 4.5: Placement of Observing Nodes

Figure 4.6: Placement of Securing Nodes

Like Transition Nodes, Observing Nodes and Securing Nodes are also special path nodes that the bot can reach. Basically Observing Nodes are the points from which the bot may spot potential threats. When the bot takes a path to go through an Observing Node, it will slow down and wait for a while to check out whether there are any threats. The observing action can help to avoid unexpected losses.

The Securing Nodes are similar to the Observing Nodes except that they are placed at L-Shape turns, T-Shape turns, and entrances. When a group of bots find that there is a turn (L-Shape turn or T-Shape turn) ahead, one of them will execute a securing action sequence to check out the existence of enemies. If there are any enemies, they will be prepared and well-positioned to launch an attack.

4.3.2 Algorithm

The task of path-finding in MOUT simulation is to find a best path from a start location to an end location as shown in Figure 4.4. Based on the area-based grid representation illustrated in Section 4.3.1, the A* algorithm is the best searching algorithm for our purpose. A* is a graph search algorithm that is extremely efficient compared to other searching algorithms, and yet intuitively easy to understand and simple to implement.

Generally, A* starts from the initial path node and searches its neighboring path nodes as its successors. The criterion of selecting its successor is based on certain heuristics. It repeats the same searching action till reaching the goal path node. Therefore, the path node sequence from the initial node to the goal node is the resulting path A* gets.

More specifically, each path node in the A* algorithm has three attribute denoted as G, H, and F. G is the cost of getting from the initial path node to this path node. H is the estimated cost of getting from this path node to the goal path node. The estimate is based on a heuristic function. F is the sum of G and H, i.e. $F = G + H$, representing the estimate of best path that goes through this node. Normally, the algorithm maintains two lists, the OPEN list and the CLOSED list. The OPEN list keeps track of those nodes that have not been explored, while the CLOSED list keeps track of those nodes that have already been explored. Here 'an explored path node' means that the algorithm has looked at every path

node connected to this one, calculated their G, H, and F values, and placed them on the OPEN list to be explored in the future. At the beginning, the OPEN list contains just the initial path node, and the CLOSED list is empty. The following pseudo-code describes the A* algorithm.

Step1. Initialize the OPEN list.
Step2. Initialize the CLOSED list.
Step3. Add the initial path node to the OPEN list.
Step4. Repeat the following steps while the OPEN list is not empty:
 Select the path node (x) with the lowest F value from the OPEN list.
 Add the path node to the CLOSED list.
 -If it is the goal path node, then quit (a path has been found).
 -If the OPEN list is empty, then quit (no path can be found).
 For every path node (x') neighboring to this path node (x):
 -Assign F, G, and H values to x'.
 -If x' is on the OPEN or CLOSED list and the existing one has a lower F value, then discard x' and continue.
 -Add x' to the OPEN list.
Step5. The ultimate path is found by backtracking from the goal path node to the initial path node.

In the A* algorithm illustrated above, the heuristic function to calculate the H value is the essential factor in regard to its performance. A good heuristic function allows the algorithm to run quickly and find the optimal path. Ideally, the heuristic function should take all the factors that contribute to the cost of getting to the goal into consideration which includes the distance to the goal as well as the difficulty to reach the goal such as the enemies on the path and the terrain geography.

With regard to the area-based grid representation model illustrated above, we implement the A* algorithm at the two levels respectively to find out the desired path. At the higher level, the bot finds a rough path from the current area to the target area through a set of Transition Nodes which connect the areas it needs to get through. As shown in Figure 4.4, the rough path from the start to the end consists of Transition Node 7, 6, 5 consequently. At the lower level, the bot finds a final path in the grid of each area. In Figure 4.4 the final path is determined from the start to Transition Node 7, from Transition Node 7 to Transition Node 6, from Transition Node 6 to Transition Node 5, and from Transition Node 5 to the end node consequently.

The heuristic functions used at the two levels are different. At the lower level, we use the Euclidean distance to calculate the H value, i.e. the cost from the current path node to the goal path node:

$$H = \text{sqrt}((\text{currentX} - \text{goalX})^2 + (\text{currentY} - \text{goalY})^2 + (\text{currentZ} - \text{goalZ})^2).$$

Every time the bot finds a path in a single grid, i.e. the same area, A* returns a sequence of path nodes from the start to the end. The path-finding at the lower level is achieved at runtime. At the higher level, the heuristic function includes two cost factors, the DISTANCE factor and the ENEMY factor. Distances are calculated between the start/goal path node and any one of the Transition Nodes as well as between any two Transition Nodes. The latter can be pre-computed using the A* algorithm implemented at the lower level as they are two fixed nodes in the same area. We build up a table to store the result for fast reference. Moreover, each section of the rough path either between the start/goal node and a Transition Node or between any two Transition Nodes has an enemy cost attached. Therefore, the total cost H is the sum of distance cost D and enemy cost E, i.e. $H = D + E$.

To summarize, our area-based grid representation and path-finding algorithm have the following benefits:

- Less computational cost. Path-finding algorithms are executed frequently at runtime. Therefore, its computational cost is a critical factor to the ultimate performance. As the time complexity of the best known A* implementation is $O(N \times \log(N))$ [Jönsson, 1997], where N denotes the size of the grid system, the computational cost increases very fast as the grid size becomes larger. Our approach solves the problem to a great extent. Every time a rough path is computed first based on the node graph of all the Transition Nodes. The computation is very fast because we have stored the cost between any two Transition Nodes in a table already. Then at runtime, the final path is computed in only one specific area with a limited grid size. In this way, the A* algorithm works well even for very large maps.
- Increased realism. When a soldier figures out a path to a destination in the real world, he/she will not determine the whole path from the start to the end in detail at the beginning. Instead, he/she finds out a viable rough path first and determines his/her final movement during execution of the rough path. Moreover, usually a soldier only determines his/her final path within the range of his/her sight. Our area-based path-finding algorithm naturally reflects the pattern of human's reasoning process.
- More flexibility to deploy warfare tactics. As mentioned above, we can adjust the grid resolution to reflect the terrain complexity. We can also denote some special information on the grid to notice the bot so that he/she can take appropriate actions such as observing or securing, corresponding to variant circumstances. Furthermore, enemy cost can be easily included into the path-finding consideration. For example, when opponents are found on the way and the squad decides to find another path so as to get around the enemies, only a new rough path needs to be generated. Therefore, the computational cost of calculating the final paths can be saved.

4.4 Group Coordination Module

In MOUT simulations, usually a squad comprises a leader and several followers. The Group Coordination Module is to coordinate the movement of the bots in the same squad. As they move within immediate distances in the game world, it is often not appropriate to calculate a path for each of them. On the one hand, if the start node and the end node are the same, the A* algorithm will generate paths overlapping with each other. On the other hand, the A* algorithm only takes static terrain data into consideration, so the bots may bump into each other if the paths are found independently. Therefore, a desirable solution is to calculate a path for the group and each bot in the group makes its own decision to move individually. The movement of each bot should take its teammates' positions into consideration while the whole squad still follows the path. In addition, each bot in the squad should have sufficient flexibility to move around which means their movement patterns are dynamically adjusted according to different situations. Therefore, statically mapping to a position around the leader is not desirable.

We use a modified version of Reynolds' steering behaviors [Reynolds, 1999] to coordinate the movements of a group of bots. Steering behaviors are developed based on the boids model which has been introduced in Chapter 2, and it has been extended to have more varieties. Generally, the basic steering behaviors are a set of primitive behavior patterns such as separation, cohesion, alignment, flee, seek, path following, etc. More complicated behaviors, such as flocking, can be modeled as combinations of the basic behaviors. For instance, when the squad moves to a desired location in a leader-following pattern, each follower combines two basic steering behaviors, i.e. seeking the leader and keeping separated from the other followers. In the MOUT simulation, the squad switches different steering behaviors to adjust the movement pattern from time to time.

Originally, the steering behaviors use a physically-based model to simulate the movements that are dynamically balanced between each boid in the same flock. Behaviorally determined steering forces are applied on the boids to produce accelerations, which in turn may change the velocities. The steering behaviors are good at simulating fast moving objects such as a flock of birds, which means typically the velocity of each object in the simulation is much larger than its turning acceleration. In this case, directly adjusting the velocity is desirable. However, in MOUT simulations bots usually moves slowly but the moving directions are changed frequently, which means their normal velocities maybe similar to or even smaller than the turning accelerations. Therefore, it is not appropriate to use the original algorithm directly in our implementation. We have developed position-based steering behaviors to simulate bot movements which are proved to be very efficient and effective.

With our position-based steering behaviors, instead of velocities, desired positions are calculated for each bot directly. More specifically, each primitive behavior calculates a desirable position and the ultimate position is the vector summation of the positions produced by all primitive behaviors involved. Therefore, in each simulation step the bot just moves to the pointed position.

In MOUT simulations, a squad usually moves in a leader-following pattern, i.e. the leader moves following a path and the rest follows the leader while keeping separated from each other. Each following bot combines leader-following, separation, and obstacle avoidance to determine a desired position in every simulation step. We give the details of these behaviors respectively as follows.

Leader-Following

In the leader-following behavior, we want the followers to keep close to the leader while keep away from the leader's path. Figure 4.7 illustrates the calculation of the desired following point.

Figure 4.7: Calculation of Following Point

First of all, we define a circle around the leader. The calculation only takes place when the bot is out of the circle, i.e. the bot needs to get closer to the leader. The desired following point is the intersection between the circle and the line connecting the bot and the leader. In addition, if the bot is within the 120° spanning of the leader's vision in front, he/she needs to move horizontally within a defined rectangular boundary as shown in Figure 4.7 using the broken line in order to avoid blocking the leader's path. In this case, the moving distance is inversely proportional to his/her horizontal distance to the leader.

Separation

The separation behavior is used to prevent the bots from crowding together. The idea behind is to repulse the bots that are too close to each other. Each bot maintains a matrix storing the distances between himself/herself and any other bots in the squad. This could be easily achieved since the GameBots protocol sends in all the bots' locations periodically. Based on the distance matrix, each bot (b_n) that is closer than a predefined threshold applies a repulsive force onto this bot (b_1). The result is a vector subtracting the location of b_n from the location of b_1 :

$$\vec{l}_n = \begin{pmatrix} X_{b_1} - X_{b_n} \\ Y_{b_1} - Y_{b_n} \end{pmatrix}$$

The vector is normalized and scaled by $1/D$, where D is the distance between the bot b_1 and the bot b_n . Therefore, the repulsive force becomes stronger if the two bots get closer. Finally all of these repulsive forces are added together to produce the ultimate steering vector:

$$\vec{L} = \sum \vec{l}_n$$

The total steering force for separation is regarded as a location vector originated from the bot's location and will be added to the following point vector to adjust the position of the bot.

Obstacle Avoidance

In the leader-following behavior pattern, the followers do not move on the grid. Instead, they try to move to desired positions calculated by the steering behaviors which may blindly lead to collisions with surrounding objects. Therefore, the obstacle avoidance behavior is used to avoid the bot bumping into any obstacles.

Currently, all the obstacles are represented using 2D straight segments in the simulation maps. Figure 4.8 shows the representation of the obstacles for both regular and irregular shapes.

Figure 4.8: Representation of Obstacles

After a desired position is calculated based on the combination of leader-following and separation behaviors, the bot first checks whether the position is reachable. If it is unreachable, i.e. the desired position is blocked by an obstacle, the intersections between the segment from the bot's current location to the desired location and all the segments of the obstacle will be found. If there are multiple intersection points found, the nearest one to the bot's location should be selected. After the intersection point is computed, a vector which is normal to the obstacle segment and starts from the intersection point is used to calculate the target position as shown in Figure 4.9.

Figure 4.9: Calculation of Obstacle Avoidance

From Figure 4.9, it can be seen that the target position is determined by the addition of the two vectors. One is the vector starting from the bot's position to the intersection point. The other vector is the normal vector of the obstacle segment.

Overall, our experimentations have verified that the implementation of the framework is efficient and effective in creating realistic bot behaviors. We have been running eight bots (a leader, three followers, and four bot opponents), a human commander, and a camera at the same time on a single Pentium 4 PC smoothly, and without any problems. Our testers acting as opponents also recognized very realistic behaviors generated by the bots that are difficult to be differentiated from human players' in various combat scenarios. Detailed experimental results are given in the following chapter.

CHAPTER 5: SIMULATIONS AND EMPIRICAL STUDIES

5.1 Overview

In this chapter, we give the experimental results with regard to our implementation of the AI framework. Section 5.2 illustrates different modules of our MOUT simulation system. Section 5.3 describes the simulations and empirical studies conducted to evaluate the proposed AI framework and its implementation.

5.2 System Setup

The Unreal Tournament game is a multi-player game supporting the client-server architecture. Therefore multiple MOUTbots and multiple human players can play in the same game concurrently by connecting to the server through network.

In our MOUT simulations, usually there are a squad of MOUTbots, a commander, and some bot enemies. The squad comprises of a leader and several followers. These bots are controlled by our AI engine. The commander is controlled by a human player who is to send specific commands to the squad. The bot enemies can be either controlled by human players or made automatic by executing certain scripts. For convenience, currently the enemies in our simulations just exhibit simple behaviors such as patrolling, or wandering around in the specific area. In addition, there may be several cameras either attached to a character in the game or acting as a spectator that can move freely to watch the game.

Typically we use three PCs to run the simulation. The first PC is used to run the server and a camera. The second one is for all the MOUTbots and bot enemies. The third one is for the human player, i.e. the commander. Figure 5.1 illustrates our system setup.

Figure 5.1: System Setup

5.3 Simulations

In this section, we describe empirical studies to demonstrate the effectiveness of our framework in producing realistic and robust bots behaviors in MOUT simulations. Two maps are used in the simulations. The first one is a small map to testify the movement capabilities of the MOUTbots. The second map is much larger so as to facilitate employment of variant warfare tactics. Detailed discussions are given in the following two sections respectively.

5.3.1 Scenario 1

In this scenario, we focus on testing the movement capabilities of the MOUTbots. A testing environment has been created as shown in Figure 5.2. It is a two-story building with two

entrances denoted as Entrance 1 and Entrance 2. The two stories are connected by three stairs. Accordingly, grids are placed onto the map by three areas separately. Area 1 is the walkable space outside the building. Area 2 is the walkable space of the first floor. Area 3 is the walkable space of the second floor. Correspondingly, Area 1 and Area 2 are connected by the two entrances while Area 2 and Area 3 are connected by the staircases. The squad comprises of three AI bots. One bot acts as the leader and the other two bots are followers. The task of the squad is to get the flag placed on the top story of the building.

Figure 5.2: Test Environment of Scenario 1

Mission is started after a 'task' command is received from the commander. Firstly, the squad finds a shortest rough path to the destination. For each section of the rough path, a final path is calculated just before that section is to be traversed. Then, the three bots move coordinately in a leader-following pattern. Before entering the building, the leader secures the entrance first. The secure sequence is to move forward step by step while turn around to check any threats in front. As no opponents are found, the squad moves into the building and goes up to the second floor following the stair. Similarly, the bot secures at the corner of the staircase. Finally, no opponents are encountered and they captured the flag successfully. Figure 5.3(a) shows a screenshot of the squad moving in a leader-following pattern. Figure

5.3(b) illustrates the path the squad takes (The ceiling of the top story and the front wall has been removed for clearness in the figure).

Figure 5.3: Scenario 1

In the scenario illustrated above, the map is fairly small. However, its complexity is sufficient to test the movement capabilities of the squad. The squad needs to take a sharp 'U' turn at the entrance and moves up following a narrow staircase. Our leader-path following algorithm exhibit quite realistic moving behaviors.

In addition, our path-finding algorithm is also very effective. As shown in Figure 5.3(b), the squad takes the shortest path towards the destination through Entrance 1. As we relocate their start locations to some other places, such as the upper-right side of the map referring to Figure 5.3(b), the squad takes another path through Entrance 2.

5.3.2 Scenario 2

In this scenario, we focus on testing the bots' capability in employing various warfare tactics based on our grid representation. A much larger map called Magnum Docks Deluxe is used, which is developed by David Brown [Brown, 2001]. It is a kind of urban dock/warehouse district which supports 6-16 players in the game. Figure 5.4 shows some screenshots of the virtual environment. All the buildings in the map are enterable by the players which provide great flexibility to employ warfare tactics.

Figure 5.4: Screenshots of Magnum Docks Deluxe

In the scenario, the squad of MOUTbots comprises of a leader and three followers. They are assigned a task to obtain three items in a specified sequence, denoted as Item 1, Item 2 and Item 3. These items are placed at three different locations in the map so that the squad

needs to find a path to each item respectively. Moreover, there are some enemies patrolling around in the map. Therefore, the squad should try to either avoid or shoot down the enemies on the way according to different situations. A human player acts as the commander to start the mission of the squad.

Obtain Item 1

Item 1 is placed on the second floor of a building as shown in Figure 5.5(a). An MOUTbot of the squad is assigned the task to obtain this item. He returns a signal to the rest once he gets the item so that the other team members can move forward to the second item immediately. Figure 5.5 illustrates the scenario in sequence.

In the scenario illustrated in Figure 5.5, an enemy is spotted on the way to Item 1. In this case, the bot tries to wait at the Observing Node until the enemy patrols away. However, other possible choices are fighting against the enemies, finding another path to Item 1 through the other entrance, or even ask for another teammate to set an ambush with him. These choices can be taken into consideration when the enemy is staying at the spotted position.

CHAPTER 5

SIMULATIONS AND EMPIRICAL STUDIES

Figure 5.5: Obtain Item 1 Illustration

Obtain Item 2

The rest of the squad, i.e. the leader and two followers, starts to move to Item 2 after they receive a signal from the bot 1. Figure 5.6 gives illustration of this scenario in detail.

Figure 5.6 (a-d): Obtain Item 2 Illustration

Figure 5.6 (e-f): Obtain Item 2 Illustration

In comparison to scenario 1, this time the quad changes the path in response to the spotted enemy. The mechanism behind is that the cost of the original path is increased and there is another path available that seems to have lower cost. Therefore, they give up the shortest path and take the safest path instead. Furthermore, as they move forward and spot an enemy bot at the L-Shape turn, they manage to shoot down the enemy due to the fact that it is the only path they can take.

Obtain Item 3

Figure 5.7 illustrates the scenario of obtaining Item 3 in sequence. This scenario focuses on coordinated bot behaviors in securing a T-Shape turn.

Figure 5.7: Obtain Item 3 Illustration

Overall, realistic and robust bot behaviors have been generated through a variety of simulations. Different actions are smoothly switched to balance the long-term goals and short-term reactions. The modified steering behaviors successfully coordinate the individual bot movements in the same squad. In addition, our area-based grid representation and the hierarchical A* path-finding algorithm based on it have been proved to be very efficient and effective in finding the best path tactically.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 Conclusion

The objective of this research is to build up a flexible, extensible, and integrable AI framework for MOUT simulations. The thesis illustrates the progress we have made towards this goal. Currently we use the UT engine as our platform which provides high-fidelity 3D graphics with sufficient modifiability. A layered architecture has been proposed and implemented which conforms to the UT/Infiltration specifications.

Overall, realistic and robust bot behaviors have been generated through a variety of simulations. The MOUTbots can not only promptly respond to immediate situational changes but also consistently pursue their goal to the end, which gives the human players a very challenging and replayable experience. Different layers of the AI framework work cooperatively to serve respective purposes. At the lowest level, different steering behaviors are smoothly switched to reflect group coordination and cooperation. On top of the steering behaviors, the Path-finding Module works efficiently to generate the desired path for the group whenever necessary. Our hierarchical path-finding algorithm using A* based on the area-based grid representation has several benefits. Firstly, it significantly lowers down the computational cost so that the Path-finding Module works well even for very large maps. Secondly, it increases simulation realism by emulating the pattern of human's reasoning process. Thirdly, it facilitates employing various warfare tactics. For instance, the A* algorithm takes enemy cost into consideration in finding the best path instead of the shortest path. Above the Path-finding Module, the hierarchical FSMs generate robust behaviors according to different situations. FSMs are simple to implement and efficient to execute,

which makes them the building blocks in the whole hierarchy. At the current stage, we use some simple rules to schedule the behaviors as a primitive planning system which need to be further developed in the future.

Moreover, each layer of the framework has great flexibility to be modified or extended with regard to its representations and algorithms as long as their interfaces to the neighboring layers keep unchanged. We can either compose more steering behaviors or add more FSMs into the Bot Behaviors Module in order to increase the variability of the behaviors. We can also change our grid system to other representations such as navigation meshes.

In addition, UT/Infiltration has been verified to be a very promising platform for MOUT simulations. Its efficient game engine as well as its generalized server-client architecture allows more computational resources for AI algorithms compared to other commercial games. In addition, its game logic is described by the object oriented language UnrealScript, which provides us with great flexibility to develop our bot AI. Furthermore, the wide community on UT/Infiltration modifications is also an additional benefit from which we easily and freely get resources and help such as the GameBots API, the JavaBot API, a variety of Infiltration maps and tools, etc.

6.2 Future Work

Simulating human-like behaviors in a virtual warfare environment is still a challenging task with a number of open problems involved. Although we have got quite promising results from our research work to date, it is nevertheless the first step towards the ultimate goal. The framework proposed provides a solid foundation to make various improvements. In this section, we discuss some potential research directions that worth putting in further efforts.

6.2.1 A Full-fledged Task Scheduling Module

In our current implementation, the Bot Behaviors Module makes most of the decisions of taking various actions. The module is implemented using a set of hierarchical FSMs. Due to the poor scalability of FSM, only simple decision logic could be incorporated into the module. In an urban warfare, however, smart decisions need to take many factors into consideration which can make the decision logic very complicated. These factors include the current situation as well as the warfare tactics and strategies. Therefore, we need a Task Scheduling Module to handle the higher level reasoning and planning mechanism.

Planning algorithms have been developed for decades as introduced in Section 2.2.1. A prominent planning system is the Soar architecture [Laird et al, 1987] which has already been used in the Soar Quakebot. As for reasoning algorithms, the BDI model mentioned in Section 2.2.1 is a good example that we can learn from.

6.2.2 Spatial Reasoning Algorithms

Spatial reasoning capability is very important for the bot in the MOUT simulation because many tactical and strategic decisions should be made based on its understanding of the virtual environment locally as well as globally. For human beings, we can quickly build up an understanding of an unfamiliar environment based on our common knowledge and spatial reasoning ability accumulated through years' of experience, such as the dimensions of the walls and the corridors or the way the rooms are connected. However, it is extremely difficult for the bots to identify the layout features of the virtual environment since the fundamental representation of the virtual world is the polygons.

To some extent, the grid representation tackles the problem by representing all the walkable spaces of the virtual environment. Although the grid system gives a good illustration of the environmental constraints, it does not provide any tactical information itself. In the current stage, the bots are explicitly informed to employ various warfare tactics by denoting specific information onto the grids. For instance, a bot always stops at an Observing Node for a predefined time interval to check out any potential threats. However, soldiers in the real warfare situation do not work in this way. Their behaviors are not predetermined. Instead, they adjust their behaviors according to the specific situation in real time. Therefore, a further enhancement is to develop spatial reasoning algorithms based on the grid representation, i.e. to analyze the grids and find out more details of the virtual environment such as the covers, the sniper positions, pinch points, etc. Several papers such as [Sterren, 2001], [Hedstrom and Lindahl, 2002] and [Lidén, 2001] give good examples in this trend.

6.2.3 Learning Capabilities

According to the current design of the AI framework, the bot makes its decision based on the hard-coded knowledge, i.e. the warfare tactics and strategies, which can be transferred from the military experts. Moreover, the bot's decision logic is also pre-determined once the implementation of the framework is done. For instance, the Bot Behavior Module is implemented using a set of hierarchical FSMs. The structure and the transition logic of each FSM are fixed unless we manually change them. In this aspect, the framework is a 'top-down' structure, i.e. all the knowledge and decision logic are explicitly specified and will not change automatically.

An essential characteristic of human beings is the learning capability, i.e. people can gain knowledge and improve its decision-making through experiences. It is also essential for the

MOUTbots to learn through their interactions with the virtual environment in order to cope with unexpected situations. In contrast to the top-down structures, we call the structures capable of learning ‘bottom-up’ structures, which means their knowledge bases as well as the decision-making mechanism can be evolved automatically.

We believe that the ideal framework should be a hybrid one that integrates both the top-down structures and the bottom-up structures. In our proposed AI framework, learning algorithms are possible to be integrated into different layers. At the lower level, the bot can learn to automatically change the parameters of the steering behaviors. For instance, when the squad moves in the leader-following fashion, they should be separated farther to each other in an open space than in a narrow corridor. At the higher level, the bot may learn to fine tune the decision-making mechanism such as the transition logic of the FSMs. In Chapter 2, we have given several prominent examples in this respect.

6.2.4 Coordination and Cooperation

A good teamwork requires well-trained coordination and cooperation among each teammate. Our current implementation of the framework does incorporate coordination and cooperation to a certain extent. Firstly, the steering behaviors nicely coordinate the movements of each bot in the same squad. Secondly, in the MOUT simulations, the bot uses messages to inform the teammates about the status of a sub-task in order to cooperate in the same squad task. For example, after a bot obtained the target item, he signals the rest to move on to obtain the next item. However, this is only our first step and a lot of work could be done in the aspect.

A potential issue in coordination and cooperation is how to automatically analyze a team task, decompose it into some subtasks, and assign each subtask to the bots or sub-teams.

Usually, to achieve a task requires good synchronization among all the subtasks. In other words, each subtask is required to be executed in a certain sequence. Therefore, the reasoning mechanism should balance various factors of each subtask, such as the difficulty, the approximate execution time, the number of bots needed, etc. In addition, each bot or each sub-team needs to dynamically adjust the plan taking into consideration the execution status of the other teammates.

6.2.5 GameBots Enhancement

As mentioned in Chapter 3, the core functionality of the GameBot API is to communicate with the UT/Infiltration server via network sockets. It receives sensory information from the server and sends action commands as well as player communication messages back to the server. However, the GameBots API does not provide all the sensory information and the action commands that the server has specified. Moreover, the information format may not meet the specification of our AI framework. Therefore, we need to modify the GameBots API to serve our purpose. Since the GameBots API source code is freely open to us, modification on the GameBot API is practicable as long as we fully understand the UnrealScript language.

In addition, the current GameBots API is consistent with the Infiltration v2.86. Recently, the new version Infiltration v2.9 has been published which provides more game play, better graphics, and more realistic bot movements. For instance, Infiltration v2.9 introduces many more vehicles, and even helicopters into the game, and the interface for human players is much friendlier. Therefore, it gives more flexibility to develop our AI framework. Upgrading the GameBots to be consistent with the latest Infiltration needs further understanding of the Infiltration specifications.

REFERENCES

- Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G. A., Schaffer, S., et al. (2001). GameBots: A 3D virtual world test-bed for multi-agent research. In *Proceedings of the International Conference on Autonomous Agents (Agents-2001) Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, Canada.
- Biddle, E. S., Stretton, M. L., & Burns, J. (2003). PC games: a testbed for human behavior representation research and evaluation. In *Proceedings of 2003 Conference on Behavior Representation in Modeling and Simulation*, Scottsdale, AZ.
- Blumberg, B. (1997). *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, Media Lab, MIT.
- Bratman, M.E. (1987). *Intentions, Plans, and Practical Reason*. Harvard University Press: Cambridge, MA.
- Bratman, M. E., Israel, D. J., & Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computation Intelligence*, 4:349-355.
- Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14-23.
- Brooks, R. A. (1990). Elephants don't play chess. In Maes, P. (ed.), *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, 3-16. The MIT Press: Cambridge, MA.

REFERENCES

- Broome, J. (2002). *Botman's bots: Half-Life bot development*. Retrieved Jun. 16, 2005, from <http://www.planethalflife.com/botman/index.shtml>.
- Brown D. (2001). *Darksnipr: Combat Mapping (Infiltration Maps and Resources)*, retrieved Jun. 16, 2005, from http://members.tripod.com/darksnipr_1/id17.htm.
- Champanand, A. J. (2003). Pathematics: routing for autonomous agents. In *Proceedings of Game Developers' Conference 2003*, San Jose, CA.
- Chapman, N. (1999). *Neuralbot*. Retrieved Jun. 16, 2005, from http://www.cyd.liu.se/~bjoli035/botepidemic/neuralbot/nb_about.htm.
- Deloura, M., editor (2001). *Game Programming Gems 2*, Charles River Media, Hingham, MA.
- Epic Games (2003). *Unreal Tournament*. Retrieved Jun. 16, 2005, from <http://www.unrealtournament.com/>.
- Ferguson, I. A. (1992). *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, Clare Hall, University of Cambridge, UK.
- Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189-208.
- ICT (2003). *ICT Games Project*. Retrieved Jun. 16, 2005, from http://www.ict.usc.edu/disp.php?bd=proj_games.
- Haddadi, A. (1996). Communication and Cooperation in Agent Systems: A Pragmatic Theory. *Volume 1056 of Lecture Notes in Artificial Intelligence*. Springer-Verlag, Heidelberg, Germany.

REFERENCES

- Hedström, J., & Lindahl, B. (2002). *Usage of First Person Shooter in Military Simulation of Combat on Urbanized Terrain*. Master thesis, Linköpings University, Sweden.
- Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K., Koss, F., & Kenny, P. (1998). TacAir-Soar: generating autonomous behavior for a distributed military training environment. In *Proceedings of the 15th National Conference on Artificial Intelligence*, Madison, WI.
- Jönsson, F. M. (1997). An optimal pathfinder for vehicles in real-world digital terrain maps, online article retrieved Jun. 16, 2005, from <http://hem.passagen.se/fmj/pathfinder/index.html>.
- Kaminka, G. A., Manuela, M. V., Schaffer, S., Sollitto, C., Adobbati, R., & Marshall, A. N. et al. (2002). GameBots: a flexible test bed for multiagent team research. *Communications of the ACM, Volume 45, Issue 1*, 43-45.
- Khoo, A., Dunham, G., Trienens, N., & Sood, S. (2002). Efficient, realistic NPC control systems using behavior-based techniques, *AAAI tech. report, SS-02-01*, AAAI Press, Menlo Park, CA.
- Laird, J. E. (2000). An exploration into computer games and computer generated forces. In *Proceedings of the 9th Conference on Computer Generated Forces and Behavior Representation*, Orlando, FL.
- Laird, J. E. (2001). It knows what you're going to do: adding anticipation to a Quakebot. In *Proceedings of the 5th international conference on Autonomous Agents*, 385-392, Montreal, Canada.
- Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1-64.

REFERENCES

- Laird, J. E. and van Lent, M. (1999). Developing an artificial intelligence engine. In *Proceedings of the Game Developers Conference 1999*, 577-588, San Jose, CA.
- Lidén, L. (2001). The use of artificial intelligence in the computer game industry. Talk at University of Michigan, Oct. 19, 2001.
- Maes, P., editor (1990). *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*. The MIT Press: Cambridge, MA.
- Maes, P. (1995). Modeling adaptive autonomous agents. In Langton, C. G. editor, *Artificial Life, An Overview*, 135-162. The MIT Press: Cambridge, MA.
- Magerko, B., Laird, J. E., Assanie, M., Kerfoot, A., & Stokes, D. (2004). AI characters and directors for interactive computer games. In *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference*, 877-883, San Jose, CA.
- Marshall, A., Rozich, R., Sims, j., & Vaglia, J. (2002). *Unreal Tournament Java Bot*. Retrieved Jun. 16, 2005, from <http://sourceforge.net/projects/utbot>.
- MOVES (2004). *The official U.S. Army game: America's Army*. Retrieved Jun. 16, 2005, from <http://www.americasarmy.com/>.
- Müller, J. P., & Pischel, M. (1993). The Agent Architecture InteRRaP: Concept and Application. *Technical Report RR-93-26*, DFKI Saarbrücken.
- Müller, J. P. (1996). The Design of Intelligence Agents: A Layered Approach. *Volume 1177 of Lecture Notes in Artificial Intelligence*. Springer-Verlag: Berlin, Heidelberg, Germany.
- Muñoz-Avila, H., & Fisher, T. (2004). Strategic planning for Unreal Tournament bots. In *Proceedings of AAAI-04 Workshop on Challenges in Game AI*. AAAI Press.

REFERENCES

- Quinlan, J. R. (1993). *C4.5: Programming for Machine Learning*. Morgan Kaufmann: San Mateo, CA.
- Rabin, S., editor (2003). *AI Game Programming Wisdom*, Charles River Media, Hingham, MA.
- Rabin, S., editor (2003). *AI Game Programming Wisdom 2*, Charles River Media, Hingham, MA.
- Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. *Technical Report 14*, Australian AI Institute, Carlton, Australia.
- Rao, A. S., & Georgeff, M. P. (1992). An abstract architecture for rational agents. In Rich, C., Swartout, W., & Nebel, B., editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, 439-449, Cambridge, MA.
- Rao, A. S., & Georgeff, M. P. (1995). BDI-agents: from theory to practice. In *proceedings of the First International Conference on Multiagent Systems(ICMAS-95)*, 312-319, Menlo Park, CA.
- Reynolds, C. W. (1987). Flocks, herds, and schools: a distributed behavioral model. *Computer Graphics* 21(4), 25-34.
- Reynolds, C. W. (1999). Steering behaviors for autonomous characters. In *Proceedings of Game Developers Conference 1999*, 763-782, San Jose, CA.
- Sentry Studios (2003). *Infiltration: This is as real as it gets*. Retrieved Jun. 16, 2005, from <http://infiltration.sentrystudios.net/>.
- Simon, H. A. (1981). *The Sciences of the Artificial (2nd edition)*. The MIT Press: Cambridge, MA.

REFERENCES

- Sterren, W. (2001). Terrain reasoning for 3D action games. In *Proceedings of Game Developers Conference 2001*, San Jose, CA.
- Sterren, W., editor (2003). *Game Programming Gems 3*, Charles River Media, Hingham, MA.
- Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. The MIT Press: Cambridge, MA.
- Stone, P., & Sutton, R. S. (2001). Scaling reinforcement learning toward RoboCup soccer. *Journal of Machine Learning Research*, 537-544.
- Stone, P., Sutton, R. S., & Singh, S. P. (2000). Reinforcement learning for 3 vs. 2 Keepaway. *RoboCup 2000*, 249-258.
- Stout, W. B., editor (2000). *Game Programming Gems*, Charles River Media, Hingham, MA.
- Swartzlander, B. (1999). *Quake 2 Bot Core*. Retrieved Jun. 16, 2005, from <http://www.telefragged.com/Q2BotCore/>.
- Sweeney, T. (1998). *UnrealScript Language Reference*. Retrieved Jun. 16, 2005, from <http://unreal.epicgames.com/Unrealscript.htm>.
- Tambe, M. et al (1999). On being a teammate: Experience acquired in the design of RoboCup teams. In *Proceedings of the third International Conference on Autonomous Agents (Agents-99)*, 221-227, Seattle, WA.
- Tyrrell, T. (1993). *Computational Mechanism for Action Selection*. PhD thesis, Centre for Cognitive Science, University of Edinburgh, UK.
- UnrealSpeed Team (2003). *UnrealSpeed*. Retrieved Jun. 16, 2005, from <http://www.planetunreal.com/unrealspeed/>.

REFERENCES

- Waveren, J. P. V. (2001). *The Quake III Arena Bot*. Master thesis, Delft University of Technology, Netherlands.
- Weiss, G., editor (2000). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press: Cambridge, MA.
- Wray, R. E., Laird, J. E., Nuxoll, A., Stokes, D., & Kerfoot, A. (2004). Synthetic adversaries for urban combat training. In *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference*, 923-930, San Jose, CA.
- Yao, J., Chen, J., & Sun, Z. (2001). An application in RoboCup combining Q-learning with adversarial planning. *RoboCup 2001*.

REFERENCES

PUBLICATION

Zhuoqian Shen, Suiping Zhou, Chee Yung Chin, & Linbo Luo. (2005). Design of an AI framework for MOUTbots. In *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation (BRIMS)*, Universal City, CA.

PUBLICATION