Behavior Representation and Simulation for Military Operations on Urbanized Terrain

Zhuoqian Shen
Suiping Zhou
Parallel and Distributed Computing Center
School of Computer Engineering
Nanyang Technological University
Singapore 639798
zqshen@pmail.ntu.edu.sg

Generating human-like behaviors in a virtual warfare environment is still a challenging task to date. In this article, the authors present their work on the design of an artificial intelligence (AI) framework for the bots in military operations on urbanized terrain (MOUT) simulations. Their framework is designed 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. A case study has been conducted that shows that the framework is effective and efficient in generating realistic bot behaviors in various combat scenarios.

Keywords: Military operation simulation, game engine, human behavior representation

1. Introduction

Generating human-like behaviors in a virtual warfare environment is still a challenging task to date, which involves various research areas such as multiagent systems, behavior representation, cognitive psychology, and so on. In recent years, the advances in military operations on urbanized terrain (MOUT) simulations have attracted both the military departments and the game community. With MOUT simulations, various tactics and scenarios can be tested in a virtual environment, which could save tremendous cost and time for the military departments as compared to real operations. A key issue in MOUT simulations is how to generate human-like behaviors for the nonplayer characters (NPCs, also known as *bots*) in the virtual environments. As computer graphics achieve amazing enhancement in recent years, game developers are putting more and more effort on game artificial intelligence (AI). Therefore, simulating realistic human behaviors is a common and important issue for both the military departments and the game industry. The AI-driven bots should be able to act as opponents against human players or as team members to cooperate with human players in the virtual environment. In this regard, we have seen some examples on how the training community and the game industry benefit from each other. On one hand, the game industry provides more realistic and modifiable game engines that can be used as simulation platforms for the training community. On the other hand, researchers in the training community are putting much effort in creating more realistic simulation models, which also leads to more interesting games.

To produce immersive, interactive, and real-time simulations, the Institute for Creative Technology (ICT) at the University of Southern California (USC) has developed two simulation systems, called *Full Spectrum Command* and *Full Spectrum Warrior* [1]. The first system is initially for 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 [2]. The key feature of these simulation systems or games is the intelligent and realistic bots that can *sense*, *reason*, and *act* in the virtual environments.

This article describes our work in this trend. We aim to develop a flexible, extensible, integrable, and platform-independent AI framework for the bots for MOUT simulations. To this end, a layered architecture is proposed that reflects the natural pattern of a human's decision-making process. In addition, the architecture allows different decision-making mechanisms and knowledge representations to be used in each layer without affecting the other layers. To

SIMULATION, Vol. 82, Issue 9, September 2006 593-607
©2006 The Society for Modeling and Simulation International
DOI: 10.1177/0037549706072093

demonstrate the effectiveness of the proposed framework, we implemented the framework using Infiltration [3], a modification of the First-Person-Shooter (FPS) game Unreal Tournament (UT), due to its affordability, effectiveness, and high fidelity [4]. The point here is to demonstrate that commercial games such as UT can provide a good platform for AI, and different AI techniques could be integrated into our framework incrementally. Human factor tests are also conducted on various tactical scenarios to evaluate the realism of the generated behaviors.

The remainder of this article is organized as follows. In section 2, we describe various commercial games that may be used as simulation platforms and the benefits of using UT as our platform. In section 3, we describe the design guidelines of our framework. Section 4 describes the implementation details of the framework. Section 5 describes a case study conducted to demonstrate the effectiveness of our framework. Conclusions and future work are given in section 6.

2. Commercial Game Engines for MOUT Simulations

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 implement other features of the game, such as 3-D rendering and generic physics/dynamics for the virtual world [5]. While the first approach is more flexible (i.e., the system can be specifically designed to meet various AI specifications), it may take more time and effort. The reason to take the second approach is that there are some off-the-shelf commercial game engines that are affordable and provide excellent support for high-fidelity 3-D virtual environments. The main limitation to this approach, however, is that the implementation of the AI engine should conform to the APIs of the specific game engine. Currently, there are several commercial FPS games in the market that are possible choices for MOUT simulations.

2.1 Quake, Quake II, and Quake III

Quake series games, especially Quake II [6], have been the popular choices for bot developers as 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 with good source code support available from a wide community. John Laird and the research group at the University of Michigan developed the Soar Quakebot that can play Quake II death matches with or against human players [7]. 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 (Dynamic Link Library) through sockets. Soar is a general cognitive architecture that can be used as an inference engine for intelligent agents [8]. In their recent work, *anticipation* is integrated into Soar's decision-making process so that the Quakebot is able to *predict* (e.g., to set an ambush or to prevent the enemy from obtaining an important weapon or health item) [9]. Alex Champandard has developed a bot for Quake II that can navigate realistically in the game world by measuring weighted rewards/costs [10]. In Waveren [11], detailed design of a Quake III Arena bot has been illustrated that uses a binary space partition (BSP) tree-based area awareness system (AAS) for navigation and routing. Besides these bots, there are some useful tools available in the game community that can facilitate the development of bots for the Quake games. One good example is Ben Swartzlander's Quake 2 Bot Core, which is an interface DLL written for communication with the Quake II server so that bot developers can easily use the DLL to build their own client bots. However, as the first-generation game engine, the Quake engine is implemented using C programming language, which makes it hard to modify and extend compared to some other engines that adopt the object-oriented languages such as C++.

2.2 Half-Life

Half-Life is an FPS game developed by Valve Software based on the code of the original Quake engine with a number of new features added. A research group at the Department of Computer Science at Northwestern University has developed a sensor and actuator interface DLL for the Half-Life server called *FlexBot* [12]. Moreover, they developed some behavior-based bots for demonstrations that can generate a set of prioritized behaviors according to different situations.

2.3 Unreal Tournament

Unreal Tournament (UT) is another FPS game released by Epic that is getting more and more recognition due to its great modifiability and expandability [13, 14]. It is able to generate a completely new game world using its shipped editor or even to design a completely new game type other than FPS games, such as UnrealSpeed, a racing modification of UT 2003 [15]. The designers of Unreal Tournament have developed UnrealScript [16], a built-in object-oriented scripting language. UnrealScript provides a high-level interface to the UT server that 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 bots as well as more interesting games [17]. The game they created, called Haunt 2, is an adventure-style game instead of a warfare game.

Our simulations are based on the UT engine and one of its modifications, called *Infiltration*. Infiltration replaces the futuristic players/bots in the original UT with modern soldiers that can walk, jog, sprint, crouch, prone, and lean around corners, and it uses a number of more realistic modern weapons. Its enhanced realism meets our requirements of MOUT simulations quite well. Various tools have been developed for the development of AI bots using the UT engine. A research infrastructure, called *Gamebots*, has been jointly developed by USC and Carnegie Mellon University (CMU) [18]. As UT is a multiplayer 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, while bots' action commands, such as "run to a specific position" or "change the weapon," are sent back to the server from the clients. Furthermore, Andrew Marshall at USC-ISI created a higher-level interface based on the Gamebots protocol, called *JavaBot* API [19]. It handles the specific Gamebots protocol, network socket programming, message passing, and other related issues, which makes the development of bot AI neater and simpler. Figure 1 shows how the different parts are organized in our simulation system.

3. Design Guidelines

In general, the design of bot AI can be regarded as a process of applying various techniques and methodologies in designing intelligent agent and multiagent systems to meet the requirements of human-like behavioral simulation. More specifically, this includes several different issues such as action selection, learning, adaptation, multiagent coordination, and cooperation. Based on different action selection mechanisms, these agent systems can be classified into reactive systems, deliberative systems, and hybrid systems. Basically, reactive systems do not have a complicated internal representation and select actions directly based on the sensory input (e.g., Rodney Brooks's subsumption architecture [20, 21]). Reactive systems intend to produce prompt and robust actions in response to the dynamic environment. However, the major problem with a reactive system is the poor scalability of the system as well as the difficulty in understanding and predicting its behaviors. On the contrary, a deliberative system usually has a symbolic representation of the world and selects actions through reasoning and planning, which is the trend of mainstream AI. Generally speaking, Soar, the underlying AI engine of the TacAir-Soar system [22] and the Quakebot, is a planning system. Such systems are good at producing goal-directed behaviors. However, high computational cost of these systems usually leads to difficulty in producing real-time responses. Therefore, a natural idea is to design a hybrid system as a combination of a reactive system and a deliberative system. Typically, such an idea leads to layered architectures [23, 24].

Learning and adaptation are essential for MOUT bots to cope with unpredictable situations in a complex environment. An action selection mechanism, which is considered as optimal at the design time, may turn out to be suboptimal because of some changes in the environment. Therefore, another essential characteristic of an intelligent agent is the ability to learn or adapt from experiences so that its action selection mechanism can improve over time. In military trainings, predictable behaviors of the opponents may lead to negative training effects. As for computer games, adaptable opponents will also result in more interesting and replayable experiences. There are a variety of learning algorithms such as decision tree learning, artificial neural networks, reinforcement learning, genetic algorithms, Bayesian learning, and so forth. Blumberg [25] has developed a virtual dog that can learn through a set of user's gestures and postures. Stone [26] also gave a good example of integrating learning algorithms with different levels of the decision-making process in constructing robotic agents playing soccer games.

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

We believe that the development of an AI framework for MOUT bot is an incremental process. A full-fledged brain may consist of a number of modules in a complex architecture. Therefore, we shall start the development from a relatively small scale and gradually expand the framework by adding variant modules.

Furthermore, all of the modules should ultimately work as an integral system. The whole system should also run as efficiently as possible (i.e., its computational cost should not be too high). Thus, the functionality of each module needs to be carefully determined, and the interfaces between different modules need to be clearly defined.

Based on the above observations, the AI framework should meet the following requirements:

- Flexible: the bots should be able to take different actions according to different surroundings; they can respond promptly to the emergent situations and, in the meantime, carry out a task smartly to achieve a goal either individually or in collaboration with other teammates.

- Extensible: various new features could be added into

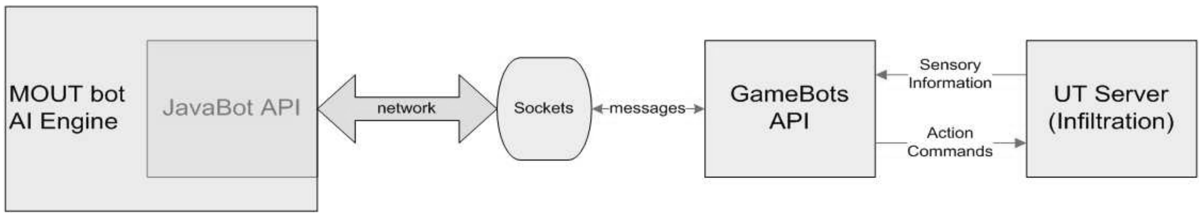


Figure 1. System organization

the framework without much difficulty.

- Integrable: different behaviors and different levels of the reasoning process should be integrated into a consistent hierarchy.

To meet these requirements, we propose an AI framework as shown in Figure 2.

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

- The architecture reflects the natural pattern of a 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 the immediate handling of urgent situations. High-level decisions, such as scheduling different tasks in a mission, correspond to deliberative behaviors and are responsible for achieving some long-term goals.

- The layered architecture is flexible and extensible, provided that the function of each layer as well as their interfaces to neighboring layers has been clearly defined. Thus, different decision-making mechanisms and knowledge representations can be employed in each layer without affecting the other layers.

Specifically, there are four layers in the proposed framework. The first layer (the bottom layer) is the Group Coordination Module, which coordinates the bots' movement in the same squad. The members of the squad are supposed to move as a group (i.e., within some immediate distance to each other). Thus, each bot's movement should take the movements of other group members into consideration. The input of this module is a desired path for the group, and the output is the specific locations in the world that a bot should move to. The calculation is based on the desired path, the group formation, and the positions of the group members. To this end, a bot should regularly update the information of its teammates as the *World Status*.

The second layer is the Path-Finding Module, which aims to find a path in the virtual world leading to a desired location. The Terrain Database stores the 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 3-D bounded hulls, as used in the Quake III Arena game. In addition, the path calculation may also include some tactical information, such as the positions of the enemies or threats, sniping spots, covering spots, and so on. Therefore, the output path may not be the shortest one or fastest one. In fact, it may be desirable for a 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, including movement actions and other warfare actions, such as shooting. Bot behaviors are the building blocks of MOUT simulations. Each behavior achieves a certain objective in a task reflecting certain warfare tactics. For example, when a squad plans to enter a room, one or two bots may need to secure the entrance of the room first. Here "secure_room" is a behavior. The input to 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 calculations. To make the behaviors reusable in different scenarios, a set of parameters is used to represent different situations. Thus, the same behavior may generate different actions according to different parameter settings.

The Task-Scheduling Module is responsible for scheduling bot behaviors appropriately, and it accomplishes the highest level of reasoning or planning that reflects various warfare strategies. The input to this module is the description of a specific task, and the output of this module is the desirable behaviors to achieve the task.

Besides the 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 resulting actions. A simple approach is to construct a table denoting the degrees of freedom (DOFs) of each action. If two actions have overlapping DOFs, they should not be carried out concurrently. For example, a bot can shoot at the enemies while moving toward a desired location. However, a bot cannot shoot at the enemies while picking up something on the ground.

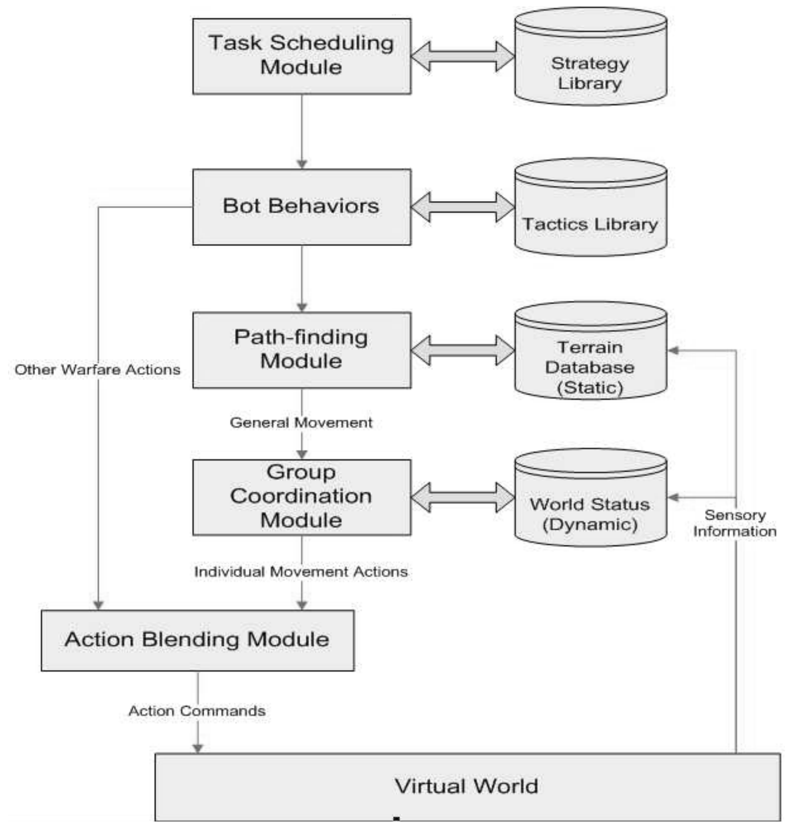


Figure 2. The conceptual model of the AI framework

4. Implementation Details

Bot Behaviors. This module can be regarded as a behavior reservoir. Each behavior is an independent program. Once its parameters are set, it generates action commands correspondingly. Different behaviors should be able to be added into or deleted from the reservoir easily. In addition, as the behaviors are the building blocks of bot AI, the implementation should be as efficient as possible. We implemented this module using a set of hierarchical finite state machines (FSMs). The FSM is widely used both in commercial games and in the bot community because of its simplicity and efficiency. Figure 3 shows a sample FSM that we have implemented. Each rectangle represents a state in the FSM. The arrows connecting different states denote transitions. For example, in Figure 3, when the squad forms a group (a state transition is triggered), each bot in the squad moves following their leader.

However, the problem with the FSM is its poor scalability. On one hand, the transitions between the states will become very complicated if there are many states in a single FSM. On the other hand, 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 this problem, we take the following two steps. First, the FSMs are organized into a hierarchy with no more than three levels, where a higher-level state corresponds to a lower-level FSM. Those states that can not be further decomposed into a lower-level FSM will generate action commands. Second, the behaviors implemented using FSMs are strictly limited to some low-level behaviors (i.e., the transition logic should not be too complicated). These two steps are to ensure that there are only a few states in each level and that the transitions among the states will not be mixed up. In addition, as we keep the sizes of the FSMs in each level to be small, it is not difficult to maintain or extend them.

Take the FSM in Figure 3 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 at the lower level. Figure 4 shows the FSM representation of the “Flee with Covering Fire” state.

Path-Finding Module. Path-finding algorithms depend on the world representation. In this project, we use a grid

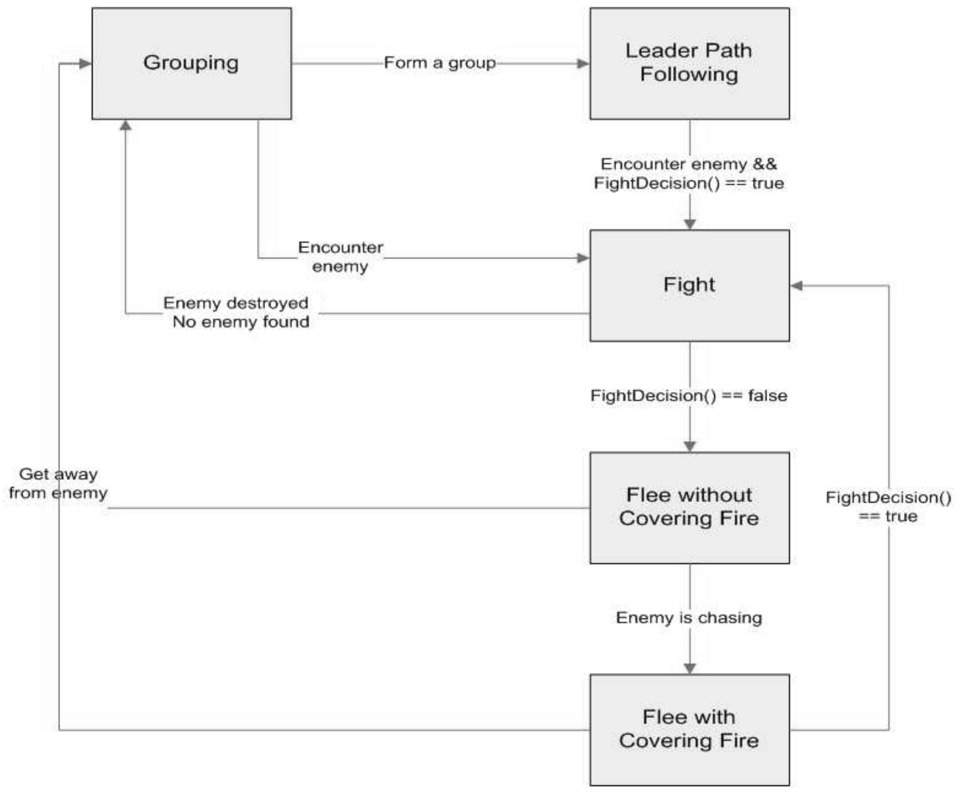


Figure 3. A sample finite state machine (FSM)

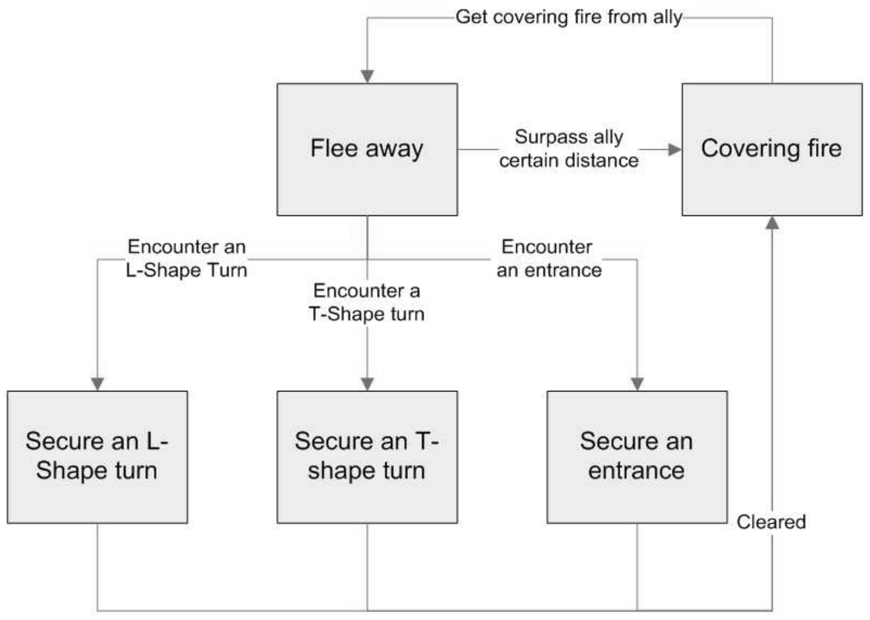


Figure 4. An finite state machine (FSM) example: “Flee with Covering Fire”

system to inform the bot about the 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 5 shows the path nodes in a rendered scene as well as in the Unreal Editor.

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, and so on. 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 some *transition nodes* that are put at the doors, stairs, and entrances. These transition nodes are special path nodes. Figure 6 gives a sample area-based grid representation. Consequently, path finding is achieved at two levels. At the higher level, the bot finds a rough path from the current area to the target area through a set of transition nodes that connect the areas it needs to get through. As shown in Figure 6, the rough path from the “Start” to the “End” consists of transition nodes 7, 6, and 5 consecutively. Then, at the lower level, the bot finds a final path in the grid of each area. In our implement, the A* algorithm is used at the two levels to find out the desired path.

The above approach has 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. The time complexity of the best-known A* implementation is $O(N \times \log(N))$ [27], where N denotes the number of path nodes. When N is large (large map with a lot of path nodes), the computational overhead is still very high for real-time tactical path planning. Therefore, if we could divide the map into many small areas and apply the A* algorithm on these small areas rather than the whole map, the computational overhead could be greatly reduced. To this end, a rough path is computed first based on the graph of all the transition nodes. The computation is very fast because we have already stored the cost between any two transition nodes in a table. Then, the detailed path is determined using the A* algorithm in each of the small areas along the rough path rather than on the whole map. In this way, the A* algorithm works well even with very large maps.

- *Increased realism*. When a soldier figures out a path to a destination in the real world, he or she will not determine the whole path from the start to the end in detail at the beginning. Instead, he or she finds out a viable rough path first and determines his or her final movement during execution of the rough path. Moreover, usually a soldier only determines his or her final path within the range of his or her sight. Our area-based path-finding algorithm naturally reflects the pattern of a human’s reasoning process.

- *More flexibility to deploy warfare tactics*. For example, when opponents are found on the way and the squad decides to find another path 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. Furthermore, grids on different areas may have different resolutions. For example, in outdoor streets, we may have lower resolution grids, while in some indoor rooms, where the bot may deploy a variety of tactics, we may have higher resolution grids. In addition, we can denote some special information on each grid to notice the bot. For example, we could denote two nodes at the entrance of a room as securing nodes, so that the bot will secure the entrance first before entering the room. Figure 7 depicts an *observing node* and some *securing nodes*.

Group Coordination Module. For a squad of bots that moves coordinately in the game world, it is often not appropriate to calculate a path for each of them. On one hand, if the start node and the end node are the same for different bots, the A* algorithm may generate overlapping paths for different bots. 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 determined 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 accordingly along the path. The movement of each bot should take its teammates’ positions into consideration while following the path for the group. We use a modified version of Reynolds’s *steering behaviors* [28] to coordinate the movements of a group of bots. Generally, the basic steering behaviors are a set of primitive behavior patterns such as *separation*, *cohesion*, *alignment*, *flee*, *seek*, *path following*, and so on. More complicated behaviors, such as *flocking*, can be generated 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 may switch to 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* (simulated flocking creatures) in the same flock. The behaviorally determined steering forces are applied on the boids to produce accelerations. Thus, the velocities of the boids are changed according to the accelerations applied. The steering behaviors mod-

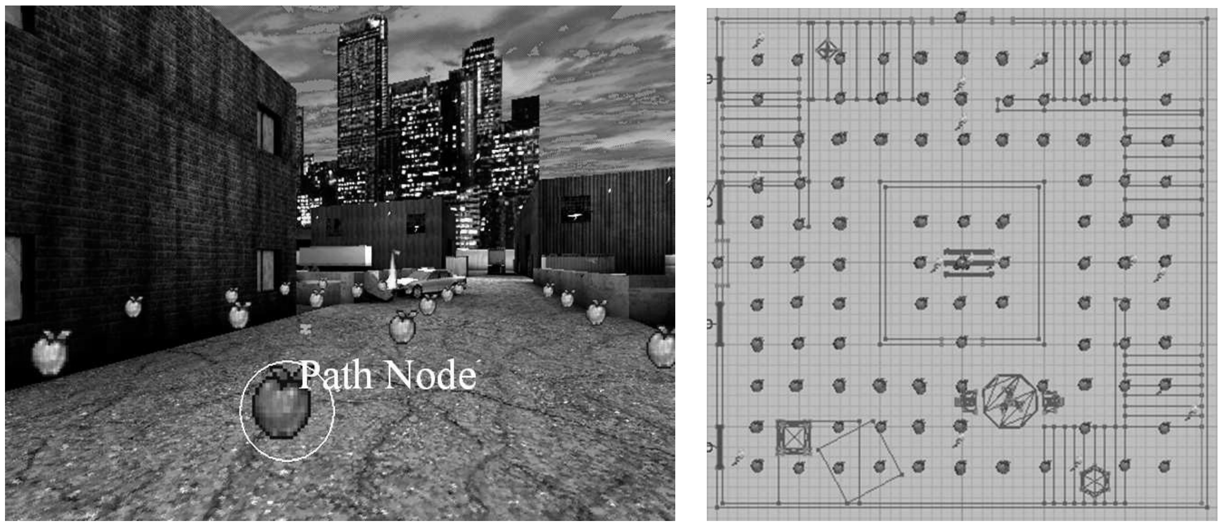


Figure 5. Path node and grid representation

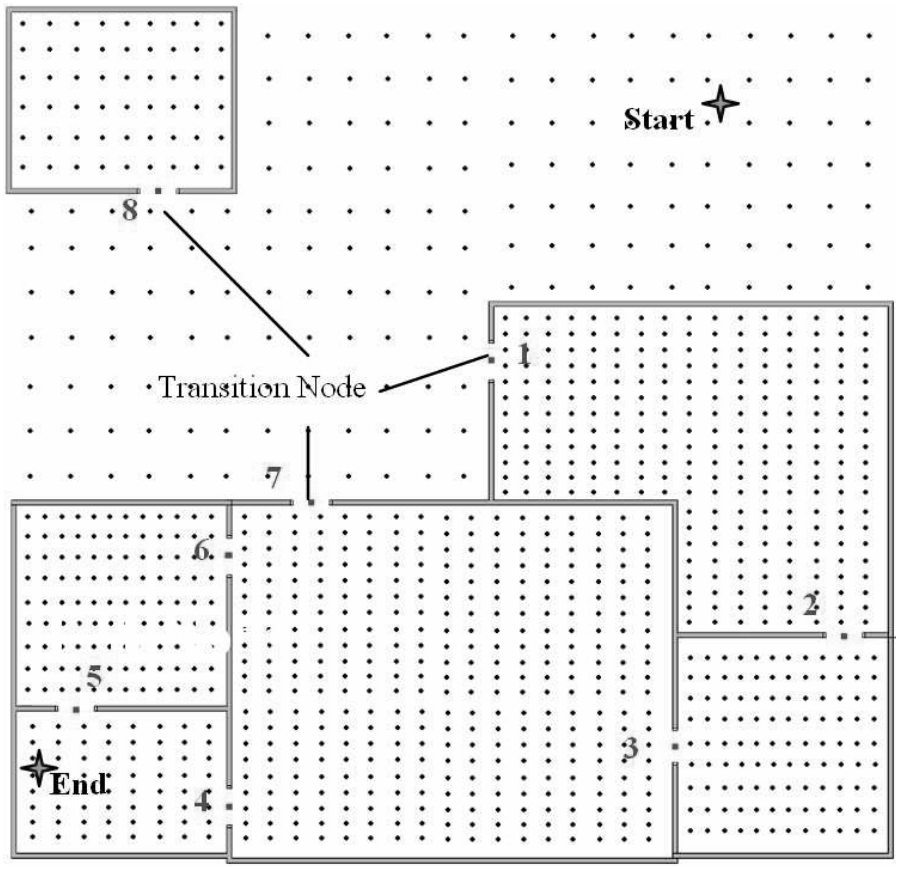


Figure 6. Area-based grid representation

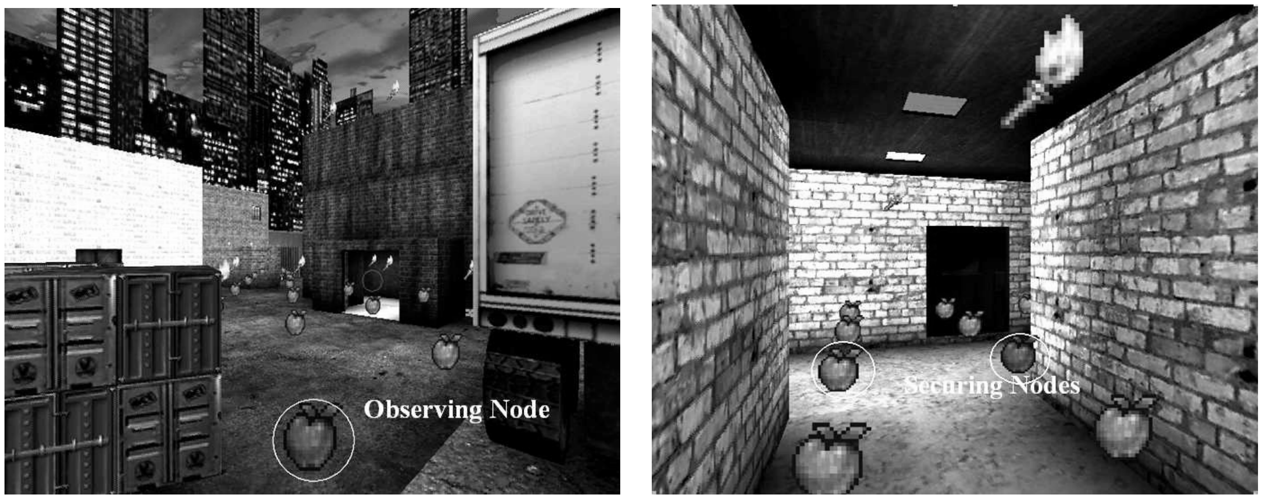


Figure 7. Placement of observing node and securing nodes

els are good at simulating fast-moving objects such as a flock of birds, where the speed of each boid is much larger than the magnitude of its turning acceleration. In this case, directly adjusting the velocity is desirable. However, in MOUT simulations, the bots usually move slowly, but the moving directions are changed frequently, which means their normal speeds may be similar to or even smaller than the magnitude of its turning accelerations. Therefore, it is not appropriate to use the original flocking algorithm directly in our implementation. We have developed some position-based steering behaviors to simulate bot movements, which are proved to be very efficient and effective for MOUT simulations.

With our position-based steering behaviors, instead of velocities, the desired positions of the bots are calculated 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. In each simulation step, a bot just moves to the desired position.

5. A Case Study

In this section, we describe some empirical studies to demonstrate the effectiveness of our framework in producing realistic and robust bot 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 bots. The second map is much larger to facilitate the employment of various warfare tactics.

5.1 Small Map

In this scenario, we focus on testing the movement capabilities of the bots. A testing environment has been created, as shown in Figure 8. It is a two-story building with two entrances denoted as *entrance 1* and *entrance 2*. The two stories are connected by three stairs. The map is divided into three areas, and grids are placed onto these 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 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.

The mission is started after a "task" command is received from the commander. First, the squad finds the 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 by the squad. 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 turning 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 capture the flag successfully. Figure 9(a) shows a screenshot of the squad moving in a leader-following pattern. Figure 9(b) illustrates the path that the squad takes (the ceiling of the top story and the front wall have been removed for clearness in the figure).

In this first scenario, 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. The

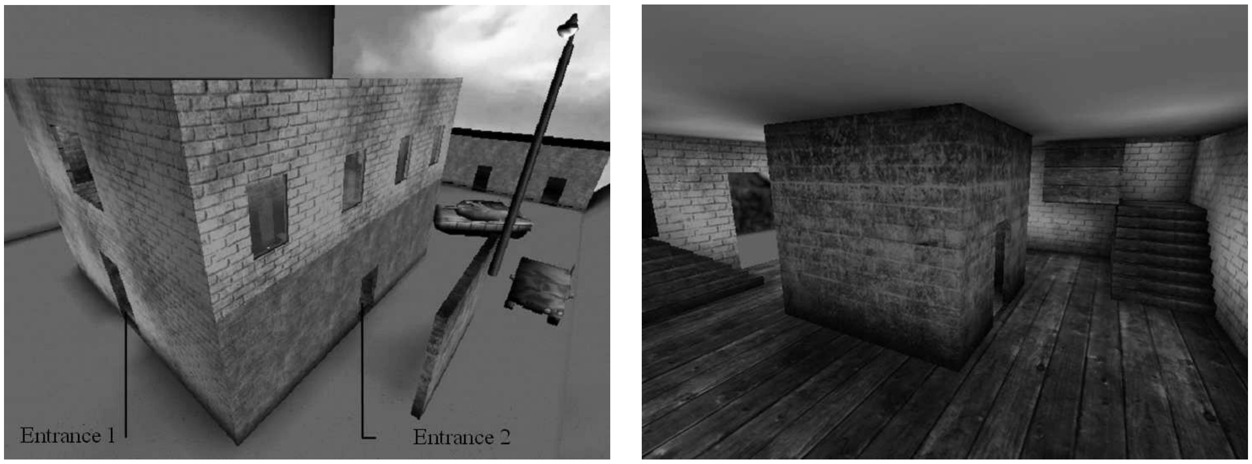


Figure 8. Testing environment

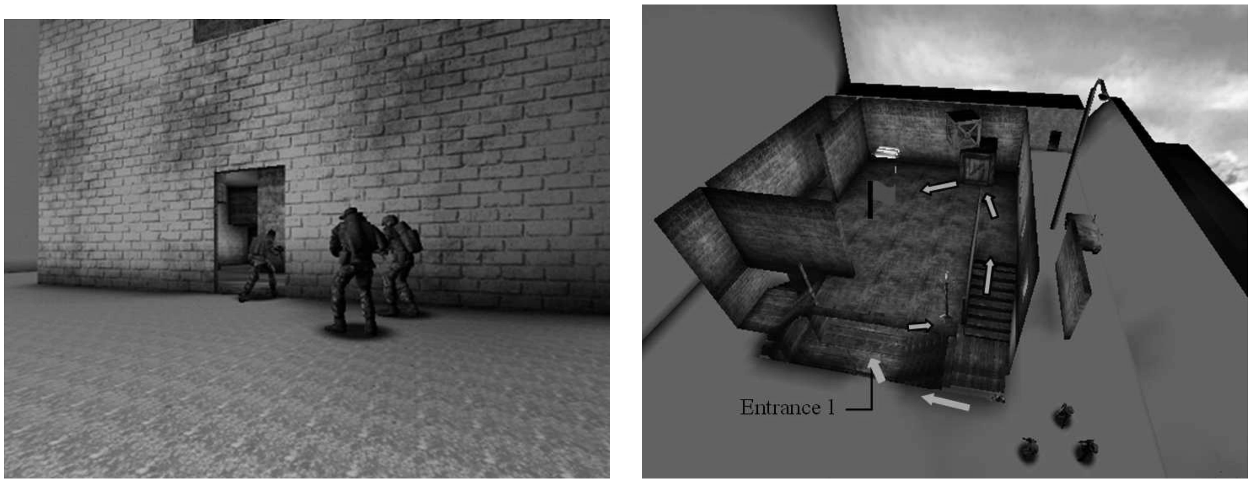


Figure 9. Scenario 1

leader-following algorithm exhibits quite realistic moving behaviors, which are observed by many students in our center.

In addition, our path-finding algorithm is also very effective. As shown in Figure 9(b), the squad takes the shortest path toward 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 9(b), the squad takes another path through entrance 2.

5.2 Larger Map

In this case, we focus on testing the bots' capability in conducting various warfare tactics based on our grid representation. A much larger map, called *Magnum Docks Deluxe*, is used, which is developed by David Brown [29]. It is a kind of urban dock/warehouse district that supports 6 to 16 players in the game. Figure 10 shows two screenshots of the virtual environment.

In the scenario, the squad of bots comprises 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. Thus, the squad should try to either avoid or kill the enemies on the way according to different situations. A human player acts as the commander to start

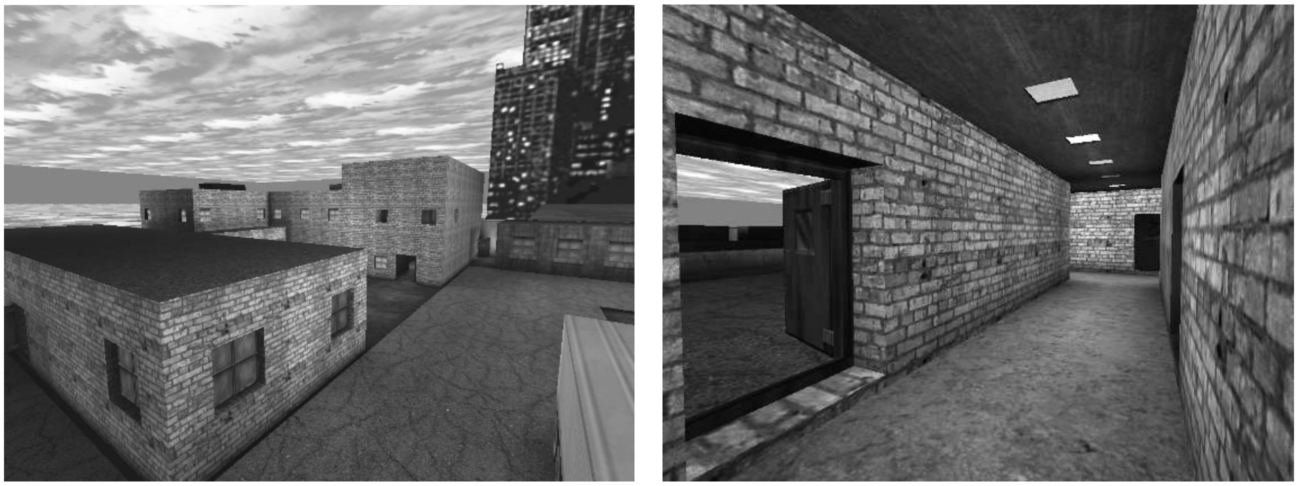


Figure 10. Screenshots of Magnum Docks Deluxe

the mission of the squad. The mission of the squad consists of three tasks: first, obtain item 1; then obtain item 2; and finally, obtain item 3.

Obtaining Item 1. Item 1 is placed on the second floor of a building, as shown in Figure 11a. A bot of the squad is assigned the task to obtain this item. The bot sends a signal to the rest bots once it gets the item. The other bots then move forward to the second item immediately. Figure 11 illustrates the scenario in sequence.

In the scenario illustrated above, 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 asking for another teammate to set an ambush with it. These choices need to be taken into consideration if the enemy is staying at the spotted position.

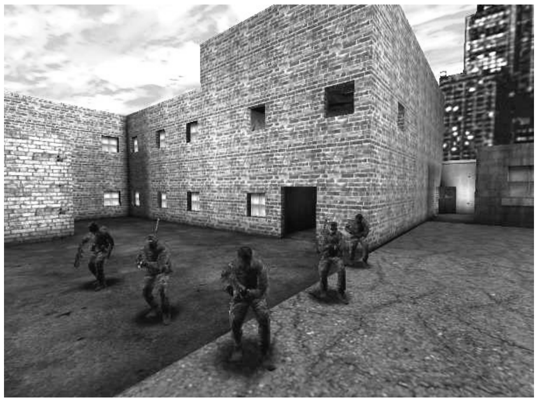
Obtaining Item 2. The rest of the squad (i.e., the leader and two followers) starts to move to item 2 after they receive a signal from bot 1. Figure 12 gives an illustration of this scenario in detail.

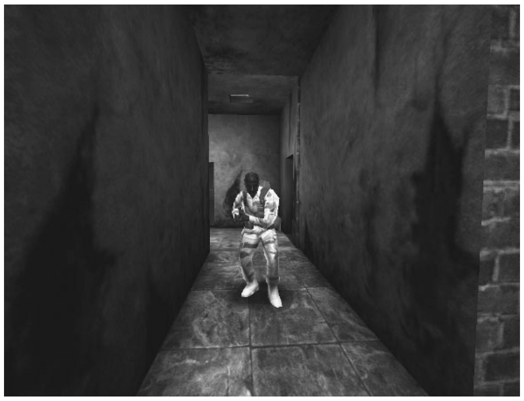
In comparison to the previous scenario, this time the squad changes the path in response to the spotted enemy. The mechanism behind this is that the cost of the original path is increased and there is another path available that seems to have lower cost (risk). Therefore, the squad gives up the shortest path and takes the safest path instead. Furthermore, as the squad moves forward and spots an enemy bot at the L-shape turn, it manages to shoot down the enemy due to the fact that it is the only path it can take.

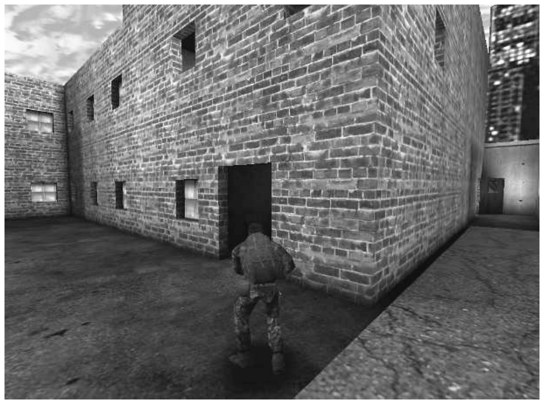
Obtaining Item 3. Figure 13 illustrates the scenario of obtaining item 3 in sequence. This scenario focuses on coordinated bot behaviors in securing a T-shape turn.

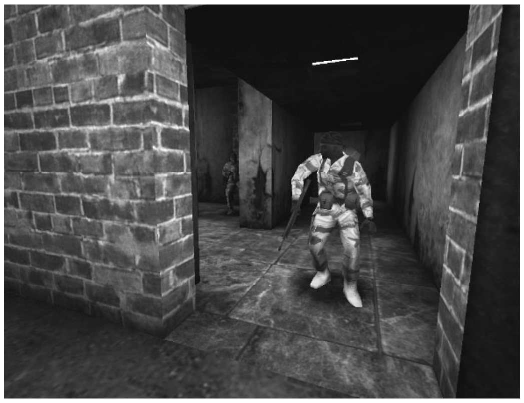
To summarize, our experiments have verified that the implementation of the framework is efficient and effective in creating realistic bot behaviors. Although our simulation can be conducted on distributed computers, we have been intentionally running eight bots (a leader, three followers, and four bot opponents), a human commander, and a recording camera at the same time on a single Pentium 4 PC to test the efficiency of our implementation. The display and simulation results show that the movements of the bots are smooth as judged by the students in our center. Different actions are smoothly switched to balance the long-term goals and short-term reactions. The modified steering behaviors successfully coordinate the movement of the bots in the same squad. In addition, our area-based grid representation and the hierarchical A* path-finding algorithm have been proved to be very efficient and effective in finding the best tactical path. It should be mentioned that we have conducted many experiments with different scenarios and human players. The human players acting as opponents and the squad leader also recognized very realistic behaviors generated by the bots that are hard to be differentiated from those generated by human players in various combat scenarios. For example, even with the same experimental setup, the bots are able to take different actions and move accordingly in response to the different actions of the human players.

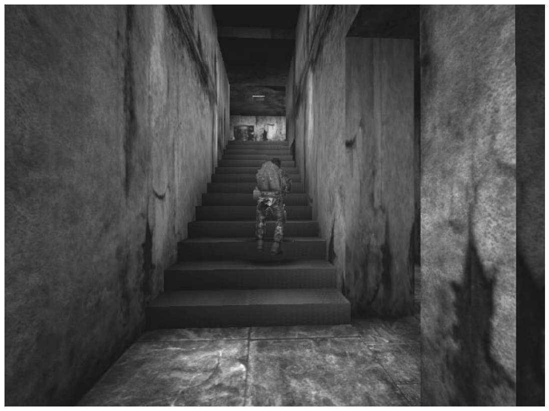
It should also be noted that the behaviors of the bots in our current implementation are still somewhat hard-coded in the sense that we need to explicitly tell the bots about the locations for executing some tactics (e.g., the locations for securing). Thus, if these locations are occupied by other bots or some static objects, the bots are unable to execute the proper actions. We hope to solve this problem by some terrain reasoning algorithms so that the bots can identify the suitable tactical key points based on the current scenario.


a. The squad and the commander are spawned in front of a building.


b. An enemy bot is patrolling along the corridor inside the building.


c. Bot 1 is assigned the task to obtain the item in the building. The shortest path is found through the entrance in front of the bot.


d. Bot 1 stops at an observing node to check for threats before moving on. The patrolling enemy bot is spotted at this time.


e. Bot 1 moves on following the stairs after the enemy bot leaves.

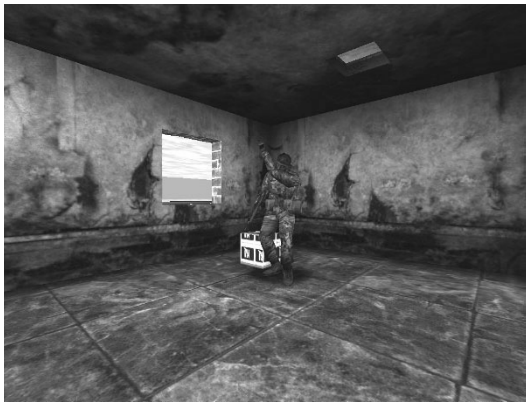
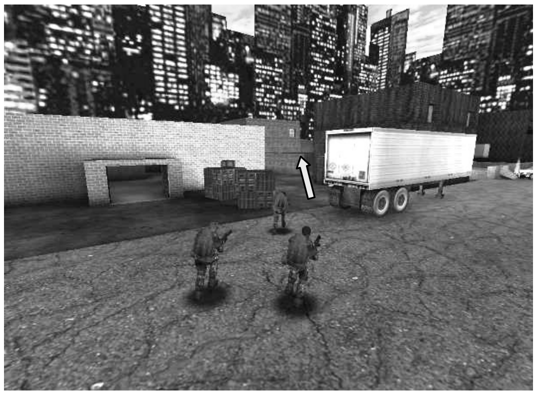
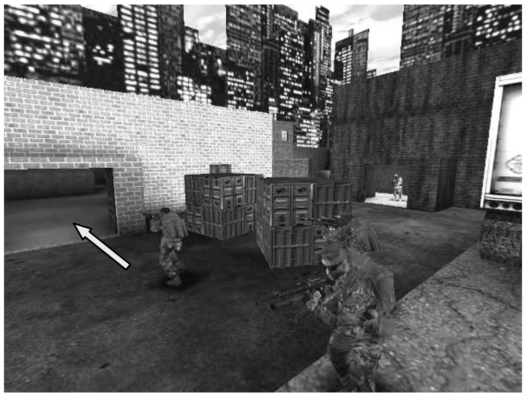

f. Bot 1 successfully gets the item and sends a signal to the rest bots of the squad.

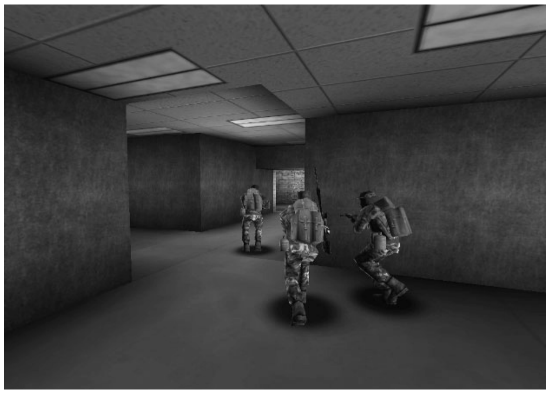
Figure 11. Obtaining item 1



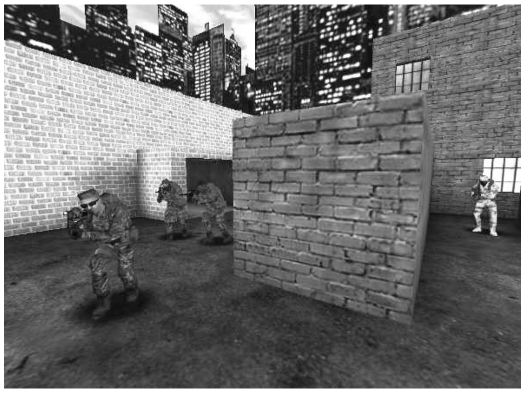
a. The squad finds the shortest path to Item 2 as pointed by the arrow. The bots move forward in a leader-following pattern.



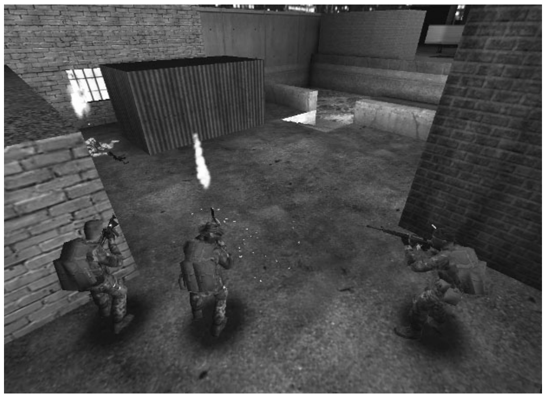
b. An enemy bot is spotted at the entrance of the building where the squad needs to pass by. The squad immediately changes to another path as pointed by the arrow.



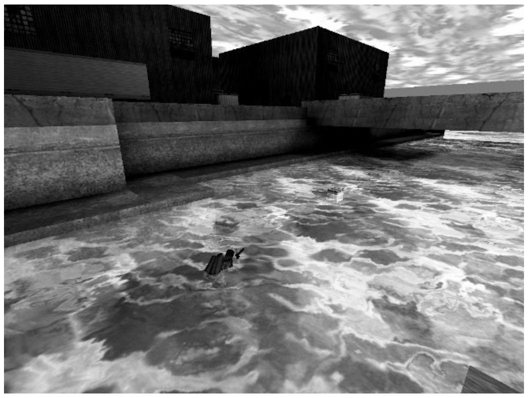
c. The squad takes the new path to Item 2 in a leader-following pattern.



d. The squad encounters an L-Shape turn on the way. The leader executes a 'Secure L-Shape Turn' sequence to check for potential threats.

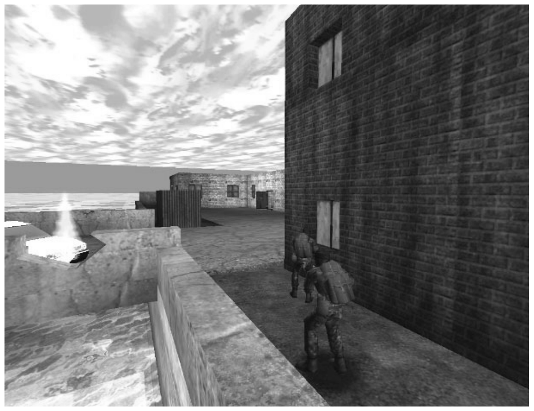


e. An enemy bot is spotted during the securing action. The bots in the squad move to appropriate positions and shoot down the enemy.

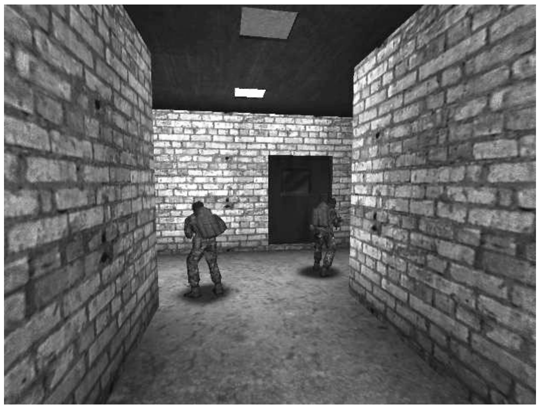


f. Bot 2 continues moving to Item 2. It sends a signal to the rest of the squad after it reaches the item.

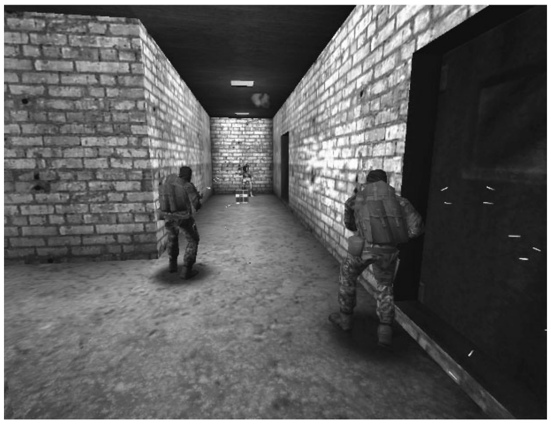
Figure 12. Obtaining item 2



a. The leader and Bot 3 keep on moving after Item 2 is obtained.



b. Item 3 is placed in a building. They move in and secure at a T-Shape turn at the end of a corridor.



c. The two bots move coordinately to shoot down the enemy at the T-Shape turn and successfully obtain the item. The task is achieved.

Figure 13. Obtaining item 3

6. Conclusions and Future Work

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

The simulation results show that the AI framework could produce very realistic bot behaviors in various scenarios. The bots can not only promptly respond to immediate situational changes but also consistently pursue their goal to the end, giving the human players a very challenging and replayable experience. Different layers of the AI framework work consistently to serve their respective purposes. Moreover, each layer of the framework can be modified and extended without affecting other layers. For example, we can introduce more steering behaviors or add more FSMs into the Bot Behaviors Module to increase the variability of the behaviors.

The proposed framework also provides a basis to make future improvements. For example, a more sophisticated higher-level reasoning and planning layer could be designed to select and schedule different behaviors. Especially, spatial reasoning algorithms need to be developed to enhance situational awareness of the bots. For example, with the spatial reasoning capability, the bots could identify suitable “secure” locations and negotiate their tactical movements in a room-clearing operation. Different learning algorithms could also to be integrated into the respective decision-making layers for a bot to cope with unexpected situations.

7. References

[1] Commercial Platform Training Aids [Web site]. Los Angeles: Institute for Creative Technologies, University of Southern California [cited August 25, 2006]. Available from: http://www.ict.usc.edu/content/view/49/103/

[2] The Official U.S. Army Game: America's Army [homepage]. Monterey, CA: The MOVES Institute, Naval Postgraduate School; c2006 [cited August 25, 2006]. Available from: http://www.americasarmy.com/

[3] Infiltration: this is as real as it gets [homepage]. Sentry Studios; c1999-2004 [cited October 18, 2005]. Available from: http://infiltration.sentrystudios.net/

[4] Unreal Tournament [homepage]. Raleigh, NC: Epic Games; c2004 [cited August 25, 2006]. Available from: http://www.unrealtournament.com/

[5] Biddle, E. S., L. Stretton, and J. Burns. 2003. PC games: A testbed for human behavior representation research and evaluation. In *Proceedings of 2003 Conference on Behavior Representation in Modeling and Simulation*, May 12-15, Scottsdale, AZ.

[6] id Software: Quake II [Web site]. Mesquite: id Software; c2002 [cited August 25, 2006]. Available from: http://www.idsoftware.com/games/quake/quake2/

[7] Laird, J. E., A. Newell, and P. S. Rosenbloom. 1987. Soar: An architecture for general intelligence. *Artificial Intelligence* 33 (1): 1-64.

[8] 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*, May 16-18, Orlando, FL.

[9] 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*, May, Montreal, Canada.

[10] Champandard, A. J. 2003. Pathematics: Routing for autonomous agents. In *Proceedings of Game Developers' Conference*, March 6-9, San Jose, CA.

[11] Waveren, J. P. V. 2001. The Quake III Arena Bot. PhD diss., Delft University of Technology, The Netherlands.

[12] Khoo, A., G. Dunham, N. Trienens, and S. Sood. 2002. Efficient, realistic NPC control systems using behavior-based techniques. In *2002 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, March 25-27. Menlo Park, CA: AAAI Press.

[13] Muñoz-Avila, H., and T. Fisher. 2004. Strategic planning for Unreal Tournament bots. In *Proceedings of AAAI-04 Workshop on Challenges in Game AI*, July 25-26. San Jose, CA: AAAI Press.

[14] Wray, R. E., J. E. Laird, A. Nuxoll, D. Stokes, and A. Kerfoot. 2004. Synthetic adversaries for urban combat training. In *Proceedings of the 16th Conference on Innovative Applications of Artificial Intelligence*, July 27-29, San Jose, CA, pp. 923-30.

[15] UnrealSpeed [Web site]. UnrealSpeed Team [cited August 25, 2006]. Available from: http://unrealspeed.planetunreal.gamespy.com/

[16] UnrealScript Language Reference [Web site]. Tim Sweeney, Epic MegaGames, Inc. [updated December 21, 1998; cited August 25, 2006]. Available from: http://unreal.epicgames.com/Unrealscript.htm

[17] Magerko, B., J. E. Laird, M. Assanie, A. Kerfoot, and D. Stokes. 2004. AI characters and directors for interactive computer games. In *Proceedings of the 16th Conference on Innovative Applications of Artificial Intelligence*, July 27-29, San Jose, CA, pp. 877-83.

[18] Kaminka, G. A., M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. 2002. GameBots: A flexible test bed for multiagent team research. *Communications of the ACM* 45 (1): 43-5.

[19] Unreal Tournament Java Bot [Web page]. A. Marshall, R. Rozich, J. M. Sims, and J. Vaglia [cited August 25, 2006]. Available from: http://sourceforge.net/projects/utbot

[20] Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 2 (1): 14-23.

[21] Brooks, R. A. 1990. Elephants don't play chess. In *Designing autonomous agents: Theory and practice from biology to engineering and back*, edited by P. Maes, 3-16. Cambridge, MA: MIT Press.

[22] Jones, R. M., J. E. Laird, P. E. Nielsen, K. Coulter, F. Koss, and P. Kenny. 1998. TacAir-Soar: Generating autonomous behavior for a distributed military training environment. In *Proceedings of the 15th National Conference on Artificial Intelligence*, July 26-30, Madison, WI.

[23] Ferguson, I. A. 1992. TouringMachines: An architecture for dynamic, rational, mobile agents. PhD diss., University of Cambridge, Clare Hall, UK.

[24] Müller, J. P. 1996. *The design of intelligent agents: A layered approach*. Volume 1177 of Lecture Notes in Artificial Intelligence. Berlin: Springer-Verlag.

[25] Blumberg, B. 1997. Old tricks, new dogs: Ethology and interactive creatures. PhD dissertation, Media Lab, Massachusetts Institute of Technology.

[26] Stone, P. 2000. *Layered learning in multiagent systems: A winning approach to robotic soccer*. Cambridge, MA: MIT Press.

[27] Jönsson, F. M. 1997. An optimal pathfinder for vehicles in real-world digital terrain maps [cited August 25, 2006]. Available from: http://hem.passagen.se/fmj/pathfinder/index.html

[28] Reynolds, C. W. 1999. Steering behaviors for autonomous characters. In *Proceedings of Game Developers Conference 1999*, March 15-19, San Jose, CA, pp. 763-82.

[29] Darksniper: Combat mapping (infiltration maps and resources) [Web site]. D. Brown [cited August 25, 2006]. Available from: http://members.tripod.com/darksniper_1/id42.htm

Zhuoqian Shen *is currently a research associate at the Learning Sciences Lab, National Institute of Education, Nanyang Technological University (Singapore). He received his double BEng degrees in power engineering and computer engineering from Shanghai Jiao Tong University (P.R. China), as well as an MEng in computer engineering from Nanyang Technological University. His current research interests include computer games, game AI, behavioral animation, and ICT in classrooms.*

Suiping Zhou *is currently an assistant professor in the School of Computer Engineering at Nanyang Technological University (Singapore). Previously, he was a postdoctoral fellow at Weizmann Institute of Science (Israel). He received his BEng, MEng, and PhD in electrical engineering from Beijing University of Aeronautics and Astronautics (P.R. China). His current research interests include distributed interactive applications, parallel/distributed systems, and human behavior representation for virtual training systems.*