BUILDING AND USING A MODEL OF INSURGENT BEHAVIOR TO AVOID IEDS IN AN ONLINE VIDEO GAME

by

PATRICK J. ROGERS-OSTEMA

B.S., Kansas State University, 2006

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Approved by:

Major Professor
David A. Gustafson

Abstract

IEDs are a prevailing threat to today's armed forces and civilians. With some IEDs being well concealed and planted sometimes days or weeks prior to detonation, it is extremely difficult to detect their presence. Remotely triggered IEDs do offer an indirect method of detection as an insurgent must monitor the IED's kill zone and detonate the device once the intended target is in range. Within the safe confines of a video game we can model the behavior of an insurgent using remotely triggered IEDs. Specifically, we can build a model of the sequence of actions an insurgent goes through immediately prior to detonating an IED. Using this insurgent model, we can recognize the behavior an insurgent would exhibit before detonating an IED. Once the danger level reaches a certain threshold, we can then react by changing our original course to a new one that does not cross the area we believe an IED to be in. We can show proof of concept of this by having human players take on the role of an insurgent in an online video game in which they try to destroy an autonomous agent. Successful tactics used by the autonomous agent should then be good tactics in the real world as well.

Table of Contents

List of Figures vi
List of Tables vii
Acknowledgements viii
Dedication ix
Chapter 1 - IEDs; An Increasing Threat to Today's Military 1
Chapter 2 – Hypothesis 1
2.1 Why this is novel 3
2.1.1 Related Work 3
2.1.2 How this is different 6
Chapter 3 - Basic Concepts 7
3.1 A Transition Matrix 7
Chapter 4 - The Game Concept 9
4.1 The map 9
4.2 Vision 11
4.3 Environmental Modifiers 11
Chapter 5 - The Game Client – The Human Insurgent Perspective 12
5.1 Movement and Vision 12
5.2 Planting IEDs 12
5.3 Detonating IEDs 14
5.4 Miscellaneous Client Controls & Notes 16
Chatper 6 - The Game Server – Host to the Client and Home of the Autonomous Agent 16
6.1 Agent Capabilities 16
6.2 Foundation for an Autonomous Agent 17
6.2.1 Fuzzy Logic 17
6.2.2 Randomization 18
6.2.3 Heat Maps 19
6.2.4 Probability Theory 19
6.3 The Fuzzy Logic System – A Custom Implementation 19
6.3.1 Variables 20
6.3.2 Fuzzy Rules 21

6.3.2.1 Operators 21
6.3.2.2 A Complex Example 25
6.4 Fuzzy Path Finder 28
6.4.1 Fuzzy Path Generation 28
6.4.1.1 Waypoint Generation 28
6.4.1.2 Fuzzy Rules 29
6.4.1.3 Waypoint Generation Continued 30
6.5 Exposed IED Marking 31
6.6 Suspected IED Marking 33
Chapter 7 - The Insurgent model 34
7.1 Deriving Useful Insurgent States from Action Sequence Data 34
7.2 Important States 36
7.3 Important SubStates 36
7.4 The Transition Matrix 37
7.5 Calculating Probabilities 38
7.6 Using the Insurgent model 38
7.7 Action Sequence Data 39
7.8 Post Mortem Propagation 41
Chapter 8 - Game Engine Components 41
8.1 Off the shelf components 41
8.2 Custom built components 42
Chapter 9 - Results 42
9.1 Iteration Zero 42

9.2 Iteration One 44
9.2.1 Iteration One Version A 44
9.2.2 Iteration One Version B 46
9.3 Iteration Two 49
9.3.1 Iteration Two Version A 49
9.3.2 Iteration Two Version B 50
Chapter 10 - Conclusion 54
10.1 A Brief Caveat 54
10.2 A Step in the Right Direction 54
10.3 40 Games, 5 wins, and the Future 55
References 56

List of Figures

Figure 1 Heat Map of Player Deaths ... 5
Figure 2 Aerial map view ... 10
Figure 3 Camouflaged agent ... 11
Figure 4 Insurgent Planting IED ... 13
Figure 5 IED Detonation ... 15
Figure 6 Fuzzy Logic Memberships Values ... 18
Figure 7.1 Before IED Detection ... 33,45
Figure 7.2 After IED Detection ... 33,45
Figure 8 Screenshot of the Match Player ... 35
Figure 9 A small piece of the transition matrix ... 38
Figure 10 Insurgent Action Sequence Data ... 40
Figure 11 Agent Action Sequence Data ... 40
Figure 12 An example iteration zero path ... 43
Figure 13 Iteration one version A path finding problem ... 45
Figure 14 Iteration one version B path finding solution ... 46
Figure 15.1 Initial Path ... 47
Figure 15.2 Avoiding IED 1 ... 47
Figure 15.3 Avoiding IED 2 and walking right into hidden IED 3 ... 48
Figure 16 Tactic Failure ... 48
Figure 17 Paranoid Agent ... 50
Figure 18 Specific insurgent behavior indicative of threat ... 52
Figure 19.1 Initial Plan ... 53
Figure 19.2 Insurgent Threat Detected ... 53
Figure 19.3 Agent Progresses, but still avoids suspicious areas ... 53

List of Tables

Table 1 Raw Transition Matrix Table Representing 76 seconds (or steps) 8

Table 2 Transition Matrix Table Representing 76 seconds 8

Table 3 Environmental Modifiers 12

Acknowledgements

I would like to thank my major professor David A. Gustafson for his continuous support and insight through this project and thesis. I would also like to thank my committee members Scott A. Deloach and Gurdip Singh for their support. Finally, I would like to thank my fiancée Sarah A. Shultz for her unrelenting kindness and support.

Dedication

I dedicate this work to my parents, Jay E. Ostema II and Rebecca L. Rogers. Thank you.

Chapter 1 - IEDs; An Increasing Threat to Today's Military

Insurgency tactics, including the use of Improvised Explosive Devices (IEDs) are a prominent concern for today's military. After first appearing in the 1940's, when they were used to derail German trains, IEDs have continued to be the cause of countless casualties [1]. In 2009, the U.S. Military recorded 8,159 IED incidents involving detonation, disarming, or detection in Afghanistan alone [2]. Advances in wireless jamming technologies have been somewhat effective in preventing IED detonations; however, some IEDs are not affected by this technique and remain a threat.

Replacing human soldiers with autonomous agents would be one way to greatly reduce the amount of human casualties from IED attacks. Certain non-combat tasks, such as transporting something from point A to point B could conceivably be done with fewer or possibly no human soldiers given a suitable autonomous agent replacement. Even if the task was too mission-critical to leave entirely up to an autonomous agent, an agent could be sent ahead of the rest of the convoy or squad in order to test the waters and find the safest route.

With the exception of IEDs attached to vehicles and other mobile objects, IEDs are usually placed and concealed (planted) at a set location before an intended target arrives. A fairly straightforward way to avoid an IED is to avoid the area in which it is planted. However, determining where an IED is located, especially when they are typically camouflaged or hidden, can be a very difficult task. Sometimes though, there are clues in the environment, especially a suspicious individual (a probable insurgent), that could indicate the potential threat of an IED in the area. A model of insurgent behavior might prove useful in detecting these areas.

Chapter 2 – Hypothesis

We can use a video game to build a model of insurgent behavior that can be used to avoid IEDs. In the video game, humans play the role of an insurgent trying to destroy an allied agent. The game client will track every relevant action of the insurgent. Each action is relayed to the game server where it is then stored sequentially in a game log. These sequentially stored actions

or action sequences will then be used to build an insurgent model. The construction of the insurgent model begins by deriving a state from each action in the game log. Then, each action sequence in the game log corresponds to a transition from one state to the next in the insurgent model. The insurgent model now describes what the insurgent has done in the game using states that describe relevant insurgent behavior.

We will use a transition matrix to store the probabilistic finite state machine which models insurgent behavior. This includes the behavior leading up to the successful detonation of an IED. This can be accomplished by observing the actions or states that an insurgent goes through before detonating an IED. In order to capture every relevant aspect of the insurgent's behavior, the game client sequentially relays insurgent actions to the game server. The game server is not capable of collecting all of these actions itself. The action sequences gathered can then be used to construct the insurgent model. This is accomplished by relating each action to a state and each action sequence to a transition between states. Each transition between states is then stored in the transition matrix in the appropriate field. The insurgent model can now be analyzed to determine which states and state changes are significant. The autonomous agent can utilize this insurgent model to identify areas that could contain IEDs and determine a means to avoid them. The effective behavior of the autonomous agent should be effective behavior in the real world as well.

An online video game offers a good setting to generate and collect action sequence data for the insurgent model. A downloadable game client that allows a player to control an insurgent whose goal is to destroy an autonomous agent trying to get from point A to point B can capture this important action sequence data. This action sequence data can then be used to construct our insurgent model. This approach offers several advantages. The most obvious advantage is that there is no danger in learning insurgent tactics in the virtual world. Another advantage is that the action sequence data collected is generated by real people from around the world giving the action sequence data for various, possibly innovative tactics. Also, because the action sequence data is generated from an entertaining video game, people will want to generate more and more action sequence data at no expense to the project.

2.1 Why this is novel

This idea is novel in that it combines the two unrelated concepts of human behavior analysis and collecting data from a video game together in order to achieve a goal. Human behavior analysis focuses on analyzing videos of human activities to determine what activity the humans are performing [15]. Collecting data from video games is a technique used by game developers to improve overall game quality by analyzing data output from the game system which is relevant to whichever aspect of the game the developers are trying to improve [4].

2.1.1 Related Work

Using models of human behavior is nothing new. In fact, there is a field dedicated to this practice; human behavior analysis [15] [17]. This field focuses on analyzing videos of one or more people and then deriving some useful information about what the people in the video are doing [15]. An example of this would be monitoring a video feed from a camera that watches over a subway station. Human behavior analysis could be used on this video feed to detect certain behavior, such as a stealing a purse, and then automatically notify the proper authorities.

One approach that has been used in this area is the Hidden Markov Model (HMM) [15]. HMMs describe unseen or unknown states in a discrete state space based on related observable information. The progress of time is modeled by a sequence of probabilistic jumps from one state to another [6]. Although this model has proven effective in human activity analysis from video feeds, it may not be ideal for our purpose. One reason is that HMMs assume that the feature sequence being modeled is the result of a single person or action [15]. Because this project involves the interaction of an insurgent and an agent, this criterion would not be met. There are alternative versions of HMMs that are capable of modeling the result of multiple entities, but they can become rather complex [16]. Because we want something that can be referenced in real time, this is not desirable. Furthermore, HMMs are designed to be used in cases where the state of an entity is not directly observable. In our case though, we can directly observe the game actions selected by the insurgent, the images seen by the insurgent, and record this information in a game log. [15]

In video event analysis, a more general form of human behavior analysis, finite state machines (FSMs) or finite state automata have proven to be very useful [16]. FSMs are a well studied and commonly used computer science concept [16]. Their strengths include modeling sequential behavior with observable states and transitions and the ability to be learned from training data [16]. Being well understood, their time complexities can be analyzed in a straightforward process [16]. A distinction to note between an FSM and an HMM is that in an FSM, states are observable as opposed to hidden. However, uncertainty can also be modeled in an FSM through the use of probabilities in transitions [16].

Likewise, collecting data relevant to the entertainment value of a video game has also been done before [4]. This has been used to improve overall game quality and usually the data collected has been much more general in nature [4]. Multiplayer and massively multiplayer online games in particular, collect data to ensure that the game is balanced or fair. This can be done in several ways. Collecting and analyzing data about the game economy, for example, allows the developer to monitor and change, if necessary, the distribution of wealth [4]. This is important because it ensures the game's currency and items bought with currency maintain an acceptable value. This in turn helps keep the game's difficulty level where it should be by ensuring players must exert the desired effort to obtain some item of value. Collecting game data is also a great way to detect cheating [4]. Because detecting cheating in real time can be infeasible due to performance requirements of the game system, a good way to detect cheating is doing an analysis on game data with a completely different system. The game data is output by the game system, but because the system analyzing the game data is a separate entity from the game system, it has no hindrance on performance. If a problem is detected, the appropriate actions can then be taken against the player.

Bungie made extensive use of collecting game data in order to ensure fairness for its multiplayer levels in the popular video game Halo 3 [3]. It did this by using a heat map, seen in Figure 1, to record player deaths in each level [3]. This captured data was used in several cases as reason to alter the level due to imbalances of deaths in certain regions indicating an advantage for one side [3].

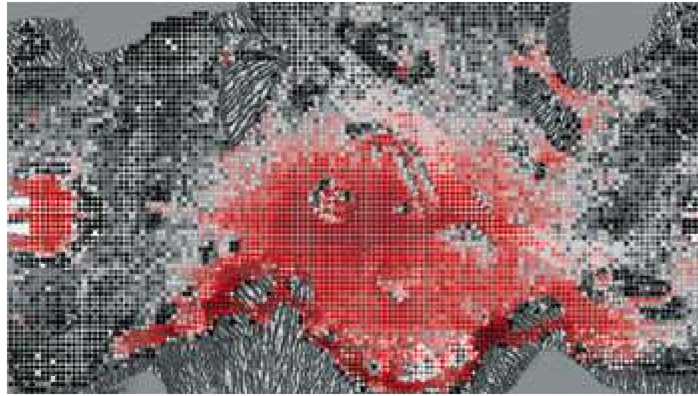


Figure 1 Heat Map of Player Deaths

Blizzard, the makers of the immensely popular game, World of Warcraft or WoW, unintentionally became contributors to what some described as a 'disease model' when a virtual disease in WoW spread much more frantically than originally intended or anticipated [5]. Scientists found player's reactions to the virtual disease fascinating and felt that even though it was 'just a game' that it was a fairly accurate example of how the people playing might react to a similar situation in real life[5].

I want to quickly note that this example is different than what we are doing for several reasons. First, the authors of the article use the term 'disease model' in a very loose way. The scientists did not generate any real scientific model in this situation. They merely made human observations about what players tended to do such as noting that some players fled the cities. [5] Second, the game data that Blizzard collects is intended to improve the game only and would not capture all of the aspects that an epidemiologist would need to build an accurate model. Specifically, the game data blizzard captures from the server would be used for normal bookkeeping or balancing purposes, and would not directly include specific player actions except for player deaths or kills.

2.1.2 How this is different

Our hypothesis is unique because it combines these two concepts in a way that we have not seen in the literature reviewed. It does this in that it captures much more specific data from a game which is only used to improve a universal model rather than the game itself, and that it uses action sequence data from a video game as the source for human behavior analysis.

The action sequence data collected for the insurgent model is different than the previous examples for several reasons. One difference is that it is being intentionally generated to build a scientific model which has relevance even outside of the game. The game data typical game designers collect has no applicability outside of their games. It is only used to improve game quality which, in theory, improves revenue. Once the improvement is complete, the game data is useless. Our action sequence data and insurgent model on the other hand, has use and worth even after the game is complete. Another difference is that game data, specifically game data generation, is typically added as an afterthought once the game is already finished. In this game, the entire purpose of the game is to generate action sequence data that can be used to build a model.

With commercial video games, game data has only been used to improve the video game itself. For our purpose, the only function of the video game is to generate and refine our action sequence data. Essentially, the focus has been reversed. Furthermore, game data typically collected by game developers is very general in nature and is generated strictly from what the game server normally sees. The biggest difference between normal game data collection and what we're doing is the action sequence data we generate comes from not only information available on the game server, but also information that only the game client can directly determine as well. One example of this is determining if it's possible that the insurgent is looking at the agent. Only the game client can determine this because only the game client renders the agent from the insurgent's point of view. The game client then relays this information back to the server, where it is recorded.

With human behavior analysis, videos are used as the source of information analyzed to derive a meaningful model [15]. In our case, we'll be using action sequence data from our video

game to derive a meaningful model from. This is clearly a very different source of information and in a way makes analyzing human behavior easier simply because we can pick and chose what information is important enough to record, and what information is superfluous.

As the information above supports, we are indeed doing something novel by using two unrelated concepts together in a way that we have not seen in the literature we reviewed.

Chapter 3 - Basic Concepts

A FSM would be a good foundation for the insurgent model. The ability to model the observable sequential state changes constituting the insurgent's behavior in a straightforward fashion is extremely important. This will allow for an easy understanding of which states and state changes lead to a successful IED detonation. The ability to learn the model directly from action sequence data, our training data, is also very important. It will allow for the insurgent model to be derived with minimal maintenance effort. The ability to use probabilities as transitions will also make modeling uncertainty a straightforward process. Being well understood and easy to work with will make the overall development of the model relatively intuitive as well.

3.1 A Transition Matrix

A transition matrix is a specific representation of FSMs. It models state transitions as probabilities in a table form with the left column representing the "from" state and the top row representing the "to" state. A transition matrix is typically used to model the transitions of a Markov chain [6]. A Markov chain can be simply described as a system with a discrete set of states that changes at discrete intervals [6]. The next state it enters depends solely on the probability distribution for its current state and not on any previous steps [6]. As the system changes randomly, it may be impossible to predict the exact state of the system in the future; however, certain states are more likely than others [6].

An example of a raw transition matrix below models someone that sits for 24 seconds, stands up for one second, and then sits back down for 51 seconds for a total of 76 seconds. Note

that the total number of transitions is always one less than the total number of steps (seconds in this case).

To / From	Sit	Stand
Sit	73	1
Stand	1	0

Table 1 Raw Transition Matrix Table Representing 76 seconds (or steps)

The values in a raw transition matrix may be hard to use in their current form as they are not normalized. Converting to probabilities is a straightforward process. It begins by totaling the number of "from" transitions for a given state or row. This total is then used to divide each value in that row returning a probability. A normalized form of the above raw transition matrix is seen below

To / From	Sit	Stand
Sit	.986	.014
Stand	1.0	0.0

Table 2 Transition Matrix Table Representing 76 seconds

Now that we have converted the values in the matrix, we can use these values to see from any given state, what is likely to happen next. In this instance, we see that 98.6% of the time the person will remain sitting if they are already sitting. 1.4% of the time they will stand up if they are sitting. 100% of the time if the person is standing they will sit back down.

A Transition Matrix is a good choice for the insurgent model for several reasons. One of the most important reasons is that it allows us to quickly and easily model action sequence data about the insurgent. Another reason is that it inherently models randomness which is important to accurately modeling reality and the insurgent. Not all insurgents are the same and we never know exactly what they are going to do next. However, because of the probability distribution of

the insurgent's given state, we can get a good idea of what is coming next. Lastly, looking up an index in a transition matrix can be done in constant time and speed is very important as we need to react to insurgent behavior as quickly as possible.

Chapter 4 - The Game Concept

This game needs to capture the tactics that an insurgent would use on an allied agent traversing an insurgent occupied area.

The allied agent will be an autonomous agent that's capable of negotiating varying terrain at a respectable speed with an end goal of reaching the other side of the map. The agent's only way of directly defending itself is a mounted gun with a fairly limited effective range. While the mounted gun is ineffective at disarming or detonating nearby IEDs, it is capable of terminating the insurgent if the insurgent gets fairly close to the agent. The agent's only real defense against IEDs then, is to avoid them.

The insurgent, controlled by a human player, is also capable of moving through varying terrain at the same relative speed as the agent. Rather than reaching a given destination though, the insurgent's task is to destroy the agent or severely delay it from reaching its destination. The insurgent is initially equipped with three small remotely triggered IEDs. The insurgent may place these anywhere on the map and has the option of instantly deploying them with no cover, or completely concealing them from view. Completely concealing an IED takes nine seconds.

4.1 The map

The map in the game should be large enough that traversing it is a non trivial task. It should also contain different path options with varying environments such that planting an IED in any one location does not guarantee that the agent will come anywhere near it.

The map is large enough that it can hold various types of objects and features giving both the insurgent and agent many options in paths to take each with advantages and disadvantages. It is in a mostly desert setting with two substantial bodies of water which cut off direct paths from some parts of the map to others. The map's surface is covered by three major features.

Bushes take up a substantial portion of the map. They provide cover, but at the cost of reduced speed. Hills also make up a lot of the map. They also reduce movement speed, but provide an extended visual distance. Open space makes up the remainder of the map. It provides average movement speed and visual distance. At the center of the map is a small city. The city is the starting place for the insurgent. Its walls and buildings provide quick hiding spots. Outside the city are several clusters of houses which can also provide line of sight cover. The last main feature of the map is roads. Roads provide the fastest movement speed and average visual distance. The agent's starting location is on a road at the "bottom" of the map. An aerial screenshot of the map can be seen below in Figure 2. The roads are white, water is solid green, and bushes look like airbrushed green. The information windows in the top right and bottom left corners are for debugging purposes only.

Figure 2 Aerial map view

4.2 Vision

Vision is a very important aspect of the game. To make the game realistic the vision of the insurgent and agent alike is affected by the environment. There is also a maximum visual distance past which the agent and insurgent cannot view each other. Given optimum conditions (both insurgent and agent on a hill) this distance is 250 units or about one sixth of the width of the map. The distance at which the agent and insurgent can see each other is calculated as the average of their adjusted visual distances. This adjusted visual distance is effected by environmental modifiers and will be elaborated upon shortly. Line of sight ultimately determines if the agent and insurgent can see each other, even if they are within visual range. If they have line of sight, no obstacles between them, then the agent can see the insurgent and plausibly the insurgent can see the agent. By plausibly we mean that the agent's model will be rendered by the game client assuming the insurgent is looking in the right direction. The agent's model is camouflaged for the environment though and can be difficult to detect even if the insurgent is looking right at it. This is illustrated in Figure 3 with the agent in some bushes.

Figure 3 Camouflaged agent

For use in the insurgent model, and the construction thereof, we will record if the agent is rendered by checking if its bounding box is inside the game client frustum. This will be useful as it indicates if it was possible for the insurgent to see the agent even though we will never know for sure if the insurgent did.

4.3 Environmental Modifiers

As already mentioned, different environment types affect the agent and insurgent in various ways. Each different location has an effect on the adjusted visual distance and movement speed of the respective entity. For example, In the foliage the adjusted vision

distance is halved making the average maximum visual distance for the agent and insurgent(how far apart they can see each other) smaller. However, the movement speed of whichever one is in the foliage is also reduced to 45% of normal. These effects are listed below in table 3.

	Vision Mod	Movement Mod
Foliage	50%	45%
Water	60%	10%
Houses	80%	75%
City	80%	75%
Road	80%	100%
Open	80%	75%
Hills	100%	40%

Table 3 Environmental Modifiers

Chapter 5 - The Game Client – The Human Insurgent Perspective

5.1 Movement and Vision

We discussed the basic abilities of the insurgent earlier, but here we will describe what the insurgent can do and how long it takes to do it. Movement is fairly straightforward and can be accomplished by using the arrow keys or the WASD keys. The agent and insurgent have the same base movement rate. If the agent gets past the insurgent then the insurgent better find a road or hope the agent decides to climb a hill if the insurgent plans to catch back up. Looking around can be accomplished by moving the mouse and the insurgent will look in the direction pointed to by the mouse. It's also possible to move one way and look another.

5.2 Planting IEDs

Successfully planting and detonating IEDs is the insurgent's main focus. The insurgent starts with three small IEDs. I use the term small because the IED's kill radius is not terribly big compared to some real world counterparts. The insurgent may switch between these three by

pressing one, two, or three respectively on the keyboard (not the number pad). The currently selected IED will be displayed in the bottom left information window on the game client. Right clicking the mouse will begin planting the currently selected IED. During this time the insurgent may rotate his vision, but may not move or else planting will be interrupted. Planting may be resumed by standing over the partially planted IED represented as a barrel, seen below in Figure 4, and right clicking again with the appropriate IED selected. Once a cumulative of nine seconds has passed an IED will be completely planted and concealed from the agent (and probably the insurgent as well).

Figure 4 Insurgent Planting IED

A 100% or completely planted IED is undetectable by the agent. However, anything below 100% planted becomes more likely to be detected as the agent gets closer to it. The agent begins to be able to detect partially planted IEDs once they are closer than 80% of his adjusted maximum vision. Once this happens, the complement of the IED's planting percentage is used to determine if the IED is visible or not. This happens by multiplying the complement of the IED's planting percentage by .8, representing 80%, and by the agent's adjusted maximum visual distance. If this value is greater than the distance to the IED, nothing happens. Otherwise, the IED is detected and any corresponding defensive maneuvers can be made. The condition for this is in equation form below.

```
Distance_To_IED <= Adjusted_Vision*.8*(1.0-Bomb_Percent)
```

(5.1)

Suppose that the IED is 50% planted and the adjusted maximum visual distance of the agent is 100 distance units.

```
Distance_To_IED <= 100*.8*(1.0-.5)

Distance_To_IED <= 100*.4

Distance_To_IED <= 40

(5.2)
```

We start by taking the complement of the IED's planting percentage which is conventiently also .5. This value is then multiplied by 80% of the agent's adjusted maximum visual distance. In this case, this is .5 multiplied by .8 multiplied by 100 resulting in 40.

We can see from this that once the agent is within 40 distance units then he is capable of detecting this IED and will plot a course around it as soon as possible. With the kill radius of the IEDs being 17 distance units, about the size of a medium house, it's clear that completely planting IEDs is crucial for them not to be detected. There is one exception to this however. That is, the agent needs line of sight to the IED to be able to detect it. So, a well placed, but not completely planted IED behind a wall, house, or hill could lead to a kill for an insurgent. This is because the agent would not be able to see the IED in time to avoid it.

5.3 Detonating IEDs

Detonating an IED within the kill radius, 17 distance units, of the agent is the ultimate goal for the insurgent. Delaying the agent from reaching its destonation in under seven minutes still counts as a win for the insurgent, but this victory is a not an importantnt one. This is because a victory by time running out is not much of a climax for the player in the first place, but more importantly it does not provide useful action sequence data. After all, not reaching the destination in time is at least half determined by how the agent responds to the insurgent's threat. So even if the insurgent is victorious simply because it slowed the agent down, because of the randomization of the agent's choices(which we'll discuss more later) this is bound to happen eventually anyway. Thankfully this form of victory rarely occurs anyway. Taking out the agent with an IED, however, is not only fun for the player, but also provides useful action sequence data. If the agent is in range, this can be accomplished by clicking the left mouse button with the

proper IED selected. Caution should be exercised though. If the agent is not within the kill radius, the IED will detonate regardless and with no effect on the agent. A screenshot of this is below in Figure 5 where the kill radius is represented by a red hemisphere. The bottom left information window shows that it was "IED 1" that was detonated and that "IED 1" is the insurgent's selected IED. Normally, the top right window is information about the client's environment. However, in this screenshot (actually taken from the server's perspective) it is debugging information and would not appear in a production version.

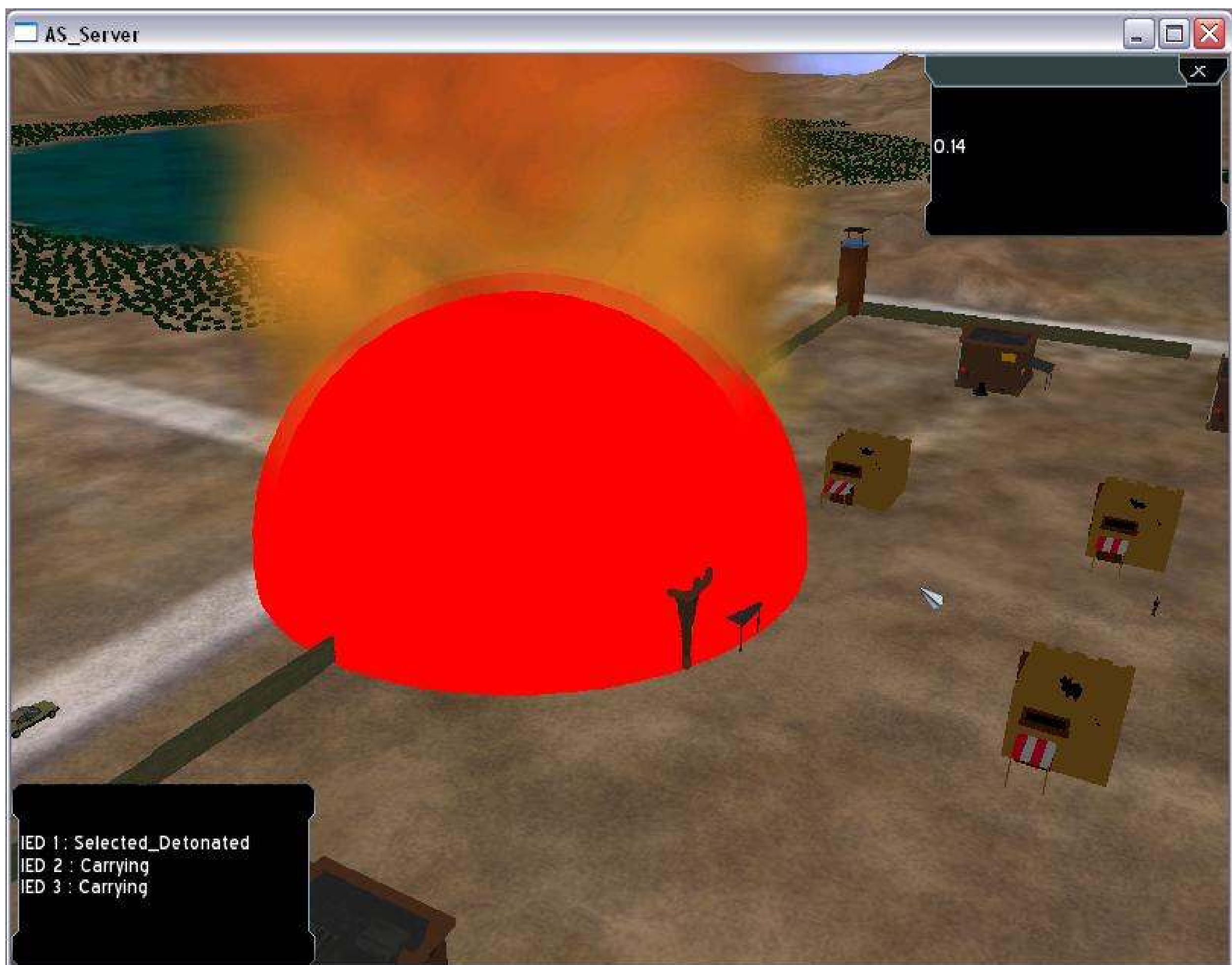


Figure 5 IED Detonation

5.4 Miscellaneous Client Controls & Notes

The game client has several other controls which do not relate to controlling the insurgent in any way. After the client executable has started up and the game window is properly rendering the world, the player must do several things before the game will start. They must first connect to the server. To connect to the server the player must press the nine key (not on the numberpad) . After the client has connected, the upper right dialog window will read "The Board is Set." At this point, the player must press the zero key (also not on the numberpad) in order to start the game. It is also important to note that because of the way that the game was designed, there are sometimes small discrepancies between what the surrounding environment looks like and how the game actually interpets the insurgent's surrounding environment. Because of this, it is helpful for the player to check the top right information window as it will display how the game is currently interpretting the insurgent's environment and therefore modifying the insurgent's vision and movement. After the player has finished a game, or if for some other reason they want to quit, they may do so by pressing the escape key.

Chatper 6 - The Game Server – Host to the Client and Home of the Autonomous Agent

6.1 Agent Capabilities

The autonomous agent's mounted gun is its only direct form of protection against the insurgent. It is capable of shooting the insurgent if the insurgent comes within 30% of the agent's adjusted maximum visual distance and there is no obstruction of line of sight between the agent and insurgent. In order to make this somewhat realistic, in the sense that the insurgent doesn't just automatically die if it gets too close to the agent, this capability is set on a timer that goes off every three seconds. This gives the insurgent on average at least a brief moment to recover in case he is caught off guard by an extremely close agent.

While the agent's gun is the most direct form of protection, its most effective form of protection is a choosing a quick, yet safe, path to its destination. If the insurgent is sloppy or caught off guard, the agent might be lucky enough to happen upon a detectable, partially planted IED before it gets within the kill radius of it. The details for this were covered previously in the

insurgent capabilities – planting IEDs section. Most of the time, the agent won't be lucky enough to happen upon a caught off guard insurgent. Assuming a well prepared insurgent is the adversary, it's likely that the IEDs will be completely hidden and our agent will need to figure out where.

6.2 Foundation for an Autonomous Agent

We want the agent to follow a set of rules that will keep it out of harm's way, but at the same time we want the agent to be unpredictable so that the insurgent can't learn its tactics. These requirements are somewhat contradictory in a sense, but using the concepts below, we've found a happy medium.

6.2.1 Fuzzy Logic

Fuzzy logic is a concept that allows computers to make decisions with human-like ambiguity where the result is approximate instead of precise. In contrast to binary logic, where a variable is either zero or one, fuzzy logic variables have membership values for a given set which range between zero and one. A membership value of zero indicates the variable has no membership in the set whereas a membership value of one indicates the variable has complete membership. Any value in between zero and one represents partial membership within a set with a higher values representing greater membership. [7]

In the following example in Figure 6, we can see that the fuzzy variable `Slow` ranges from zero to 55. All speeds up to 40 have 100% membership in `Slow` while any speeds between 40 and 55 have only partial membership.

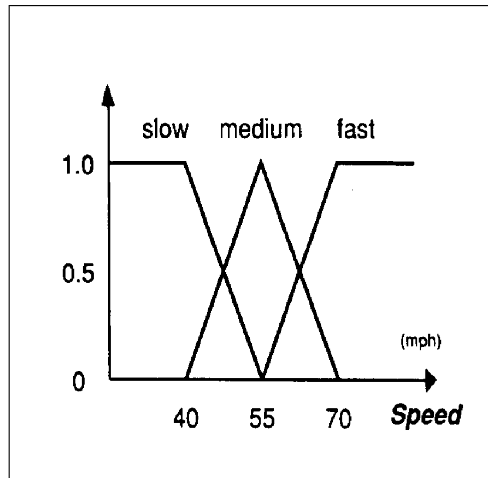


Figure 6 Fuzzy Logic Memberships Values [14]

Fuzzy logic will be a very useful tool because it allows us to easily work with the insurgent model and also generate unpredictable results. The autonomous agent will be making heavy use of the insurgent model. It is crucial, for performance reasons, that the agent is based on a system which can quickly and easily relate to the uncertainty of the insurgent model's probabilities. With fuzzy logic being focused around approximation instead of precision, this integration will be fairly straight forward. We also want our autonomous agent to be somewhat unpredictable. If our agent is predictable then insurgents will quickly discover its patterns and use them to their advantage. Fuzzy logic, along with some randomization, will allow for easy development of rules that will produce the results that we want.

6.2.2 Randomization

Randomization, at least the general concept itself, may seem a bit simple, but it is vital to making the agent unpredictable. The use of random number generators will allow the agent to greatly vary the tactics it uses with every decision it makes. By seeding path generation functions with random numbers, we can enhance the unpredictability of the agent.

6.2.3 Heat Maps

A heat map was introduced earlier in Figure 1 with the Bungie example involving the number of deaths indicated by the intensity of red in a given position. Heat maps lend themselves to use in the agent because they allow quick and easy storage of position specific data. A heat map is basically a two dimensional array of values that can be converted into a picture, if desired, with the color in the picture being based on the values in the array. This is an effective way to store information about the game map. Although the game is 3D, we can store useful information about the environment in 2D because the third dimension, “up”, only has one value for each x, y. Basically, the third dimension of the game is really only used for graphical purposes and determining line of sight. As far as non-graphical map data goes, the third dimension doesn’t really matter.

6.2.4 Probability Theory

Probability theory concerns itself with the mathematical analysis of non-deterministic events. A probability indicates that if a certain event takes place for a long enough period of time, that certain statistical patterns will emerge. At the beginning of the paper, we discussed using a transition matrix to model insurgent behavior. Specifically, what we will use are the probabilities in the matrix given the insurgent’s state to determine if the agent is in danger. Using statistics gathered from action sequence data we can determine a certain belief level of what is going to occur next. This makes probability theory a powerful tool for our agent to use. [10]

6.3 The Fuzzy Logic System – A Custom Implementation

The necessity of a real time system greatly influenced the design of the Fuzzy Logic System. After all, if the agent takes too long to make decisions there's a very good chance that he'll never complete his mission. With some fuzzy logic defuzzification techniques being rather involved and time consuming, it was decided to go with a more simplistic approach.

6.3.1 Variables

There are four types of custom variables that the Fuzzy Logic System handles.

FuzzyVariables are composed of three different values. These values are called High, Medium, and Low. Each value is a float that is representative of the FuzzyVariable's membership within a respective set with zero representing no membership and one representing full membership. FuzzyVariables in this system are by convention named contextually to convey an intuitive meaning. For example, if we have the FuzzyVariable `Distance_To_Goal` and its High value is set to one, then this indicates that the distance to goal is high.

FloatyVariables are basically the Fuzzy Logic System's version of a float. They can be used in the system when exactness is important. They can also double as a flag or Boolean value when set to one or zero. As an example we could have the FloatyVariables `Is_Insurgent_In_Range` which as discussed earlier, is something that is simply either true or false. If we set `Is_Insurgent_In_Range` to one this would indicate that the insurgent is in range.

FuzzyHeatMaps are basically FuzzyVariables in heat map form. Every x, y location on the map has a High, Medium, and Low Bitmap associated with it. Each Bitmap stores a value between zero and 255 (black to white). At runtime, these values are rescaled between zero and one. In short, they allow fuzzy information about a given location to be easily accessed at runtime and easily stored in a user friendly form.

FuzzyExterns are the final type of variable that the Fuzzy Logic System uses. In reality, they are one of the three types of variables that we just covered, except that they are stored and managed outside of the core Fuzzy Logic System. This is useful if we need to use a variable that needs to be referenced or managed by other components of the agent or game server. For instance, in the previous example about the FloatyVariable `Is_Insurgent_In_Range`, `Is_Insurgent_In_Range` would need to be a FuzzyExtern. This is because the core Fuzzy

Logic System itself has no way of knowing if the insurgent is in range or not. This information must be updated by the game server.

6.3.2 Fuzzy Rules

A fuzzy rule is composed of an antecedent and a consequent. The purpose of the antecedent is to determine if the consequent fires. The antecedent is basically the same as the "if" part of an "if-then" construct. The purpose of the consequent is to react to the antecedent based on the degree of truth (modifier) of the antecedent. This is basically the fuzzy logic analog to the "then" part of an "if-then" construct. Both the antecedent and the consequent use operators to evaluate and if necessary modify variables. Although the antecedent and consequent have some operators that share the same name, operators in the antecedent strictly make evaluations on variables whereas the operators in the consequent strictly make modifications to variables.

6.3.2.1 Operators

The membership operators "is" and "isnot" work with FuzzyVariables and FuzzyHeatMaps. When used in an antecedent the "is" operator does a direct look up in whichever field is requested. When used in the consequent the "is" operator does a direct replacement to whichever field is requested using the modifier from the antecedent. Suppose we have the rule If Distance_To_Goal is High Then Speed is High. We begin processing this rule by performing a look up on Distance_To_Goal's High value. If this value is non-zero then the consequent Then Speed is High will fire. When this happens, the non-zero value, called the modifier, is passed to the consequent. The modifier is then assigned to Speed's High value. We can see an example of this below in 6.1.

```
Initial values:
FuzzyVariable DTG = FuzzyVariable(0.0,0.4,0.8) // FuzzyVariable(Low, Medium, High)
FuzzyVariable Speed = FuzzyVariable(0.2,0.1,0.1)

Rule:
If DTG is High Then Speed is High

Evaluation:
If DTG is High Then Speed is High
If 0.8 > 0 Then Speed is High                    // is  operator returns a non-zero value
If True (0.8) Then Speed is High                 // the antecedent is true
If True Then (0.8) Speed is High                 // modifier is passed to the consequent
If True Then Speed = FuzzyVariable(0.2,0.1,0.8)  // is operator replaces Speed's High -
                                                 // value with the modifier

(6.1)
```

The membership operator "isnot" is simply the complement of "is" and behaves exactly the same way except it performs a one minus to the requested value. We see an example of this below in 6.2.

```
Initial values:
FuzzyVariable DTG = FuzzyVariable(0.0,0.4,0.8)
FuzzyVariable Speed = FuzzyVariable(0.2,0.1,0.1)

Rule:
If DTG isnot High Then Speed is High

Evaluation:
If DTG isnot High Then Speed is High
If (1.0 - 0.8) > 0 Then Speed is High            // isnot operator takes complement and -
                                                 // returns a non-zero value
If True (0.2) Then Speed is High                 // the antecedent is true
If True Then (0.2) Speed is High                 // modifier is passed to the consequent
If True Then Speed = FuzzyVariable(0.2,0.1,0.2)  // is operator replaces Speed's High -
                                                 // value with the modifier

(6.2)
```

The operators less-than, equal-to, and greater-than work with FloatyVariables in the antecedent. These work in the traditional binary logic fashion. If Is_Insurgent_In_Range = 1 would compare the value in Is_Insurgent_In_Range to 1 and if they were equal it would pass on 1 as the modifier and fire the consequent.

The operators “And” and “Or” allow the construction of complex antecedents. The operator “And” allows the construction of complex consequents as well. These operators can be nested although it is important to note that precedence is given to the leftmost operation as when the operations are formed the leftmost operation is parsed first and then combined with a recursive call on the rest of the string. In the antecedent, both “And” and “Or” function in the standard mathematical way where “And” is “Min” and “Or” is “Max”. In the consequent, the operator “And” allows for more than one consequent to be fired for a given antecedent. In this case, the modifier from the antecedent is the same for each component consequent. Suppose we had the rule If Distance_To_Goal is High And Time is Low Then Speed is High and Be_Cautious is Low. The “And” operator in the antecedent corresponds to the “Min” operation and will result in the lesser of the two values being returned. So, if Time is Low returns 0.9 and Distance_To_Goal is High returns 0.8 then 0.8 will be the final result for the antecedent. 0.8 is non-zero and so the consequent will fire with 0.8 being the modifier. Next, Speed’s High value and Be_Cautious’ Low value will be replaced with 0.8. An example of this is seen in 6.3.

```
Initial values:
FuzzyVariable DTG = FuzzyVariable(0.0,0.4,0.8)
FuzzyVariable Time = FuzzyVariable(0.9,0.3,0.0)
FuzzyVariable Speed = FuzzyVariable(0.0,0.3,0.0)
FuzzyVariable Be_Cautius = FuzzyVariable(0.2,0.5,0.3)

Rule:
If Distance_To_Goal is High And Time is Low Then Speed is High and Be_Cautius is Low

Evaluation:
If Distance_To_Goal is High And Time is Low Then Speed is High and Be_Cautious is Low
If Min(Distance_To_Goal is High, Time is Low) Then Speed is High and Be_Cautious is Low

If Min(0.8,0.9) Then Speed is High and Be_Cautious is Low        // And takes the Min
If 0.8 > 0.0 Then Speed is High and Be_Cautious is Low           // antecedent is non zero
If True (0.8) Then Speed is High and Be_Cautious is Low          // antecedent is true
If True Then (0.8) Speed is High and Be_Cautious is Low          //modifier passed to consequent
If True Then Speed = FuzzyVariable(0.0,0.3, (0.8)) and           // Speeds High value is replaced
             Be_Cautious = FuzzyVariable((0.8),0.5,0.3)          //Be_Cautious’ Low value replaced
```

(6.3)

The operators “isless”, and “ismore” allow FuzzyVariable and FuzzyHeatMap values to be adjusted, rather than replaced, in the consequent. The “isless” and “ismore” operators basically function the same as each other except that one subtracts where the other adds. The “ismore” operator increases the value in the desired membership field by up to a maximum amount stated in the consequent immediately following the desired field to adjust. This increase is of course affected by the modifier. All modifications cap at zero or one. Suppose we had the example rule If DTG is High Then Speed ismore High .6. We would first determine if the antecedent was true in the usual manner. If the antecedent is true, and then the non-zero modifier is passed to the consequent. The modifier is then multiplied by the maximum adjustment amount specified in the consequent to get the amount that we will adjust the desired membership field by. Assuming the modifier was 0.5, in this case we would have 0.5 x 0.6 resulting in 0.3. Finally, we would add 0.3 to whatever value was already in the membership field High for Speed. Completing the process, we would do a boundary check on the membership field High. If the new value was less than zero it would be reset to zero. If the new value were greater than one it would be reset to one. An example of this can be seen below in 6.4

```
Initial values:
FuzzyVariable DTG = FuzzyVariable(0.0,0.4,0.5) // FuzzyVariable(Low, Medium, High)
FuzzyVariable Speed = FuzzyVariable(0.2,0.1,0.8)

Rule:
If DTG is High Then Speed ismore High .6

Evaluation:
If DTG is High Then Speed ismore High .6
If 0.5 > 0 Then Speed ismore High .6            // is  operator returns a non-zero value
If True (0.5) Then Speed ismore High .6         // the antecedent is true
If True Then (0.5) Speed ismore High .6         // modifier is passed to the consequent

If True Then Speed = FuzzyVariable(0.2,0.1,0.8 + (0.6x0.5))

                                                // Speed’s High value is adjusted based
                                                // on the modifier and max adj amount

If True Then Speed = FuzzyVariable(0.2,0.1,1.1 > 1.0)   // new value is out of bounds
If True Then Speed = FuzzyVariable(0.2,0.1,1.0)         // value is corrected
```

(6.4)

As mentioned before, “isless” works in the exact same way except that it subtracts from a value instead of adding to it.

The operators “dec”, and “inc” work with FloatyVariables in the consequent. The operators“dec” and “inc” are short for decrement and increment respectively. The “inc” operator increases the FloatyVariable’s value by up to a maximum amount specified immediately after the “inc” operator. The modifier is multiplied by the maximum amount to get the actual amount adjusted. The “dec” operator functions in the same way except that it decreases the value instead of increasing it. Suppose we had the rule If Time is Low Then Actual_Speed inc 5.0. We would evaluate the antecedent in the normal way. Assuming a true antecedent with a modifier of 0.5 we continue by multiplying 0.5 by the specified maximum adjustment amount which in this case is 5.0. This would result in an actual adjustment of 2.5. We finish by adding 2.5 to the value of Actual_Speed. An example of this is below in 6.5.

```
Initial values:
FuzzyVariable Time = FuzzyVariable(0.5,0.4,0.5)
FloatyVariable Actual_Speed =10.0

Rule:
If Time is Low Then Actual_Speed inc 5.0

Evaluation:
If Time is Low Then Actual_Speed inc 5.0
If 0.5 > 0 Then Actual_Speed inc 5.0            // is  operator returns a non-zero value
If True (0.5) Then Actual_Speed inc 5.0         // antecedent is true
If True Then (0.5) Actual_Speed inc 5.0         // modifier passed to consequent
If True Then Actual_Speed = 10.0 + 5.0x(0.5)    // modifier multiplied by max adj val
                                                // and added to the FloatyVariable
If True Then Actual_Speed = 10.0 + 2.5
If True Then Actual_Speed = 12.5                // new value for Actual_Speed

(6.5)
```

6.3.2.2 A Complex Example

The following example illustrates many components of the Fuzzy Logic System working in unison. Suppose we have the rule If Is_Insurgent_In_Range = 1.0 Or Threat is High Then Movement_Speed dec 5 And Caution ismore High 0.5. We begin by evaluating the complex antecedent If Is_Insurgent_In_Range = 1.0 Or Threat is High. The “Or” operator corresponds to “Max” and so our final antecedent value will be the maximum of our two component antecedents. Upon inspection, we find that Is_Insurgent_In_Range is equal to 0. Because we are doing an equality check of it with the value 1, we find that If Is_Insurgent_In_Range = 1.0 is false

and so returns the value 0. Threat is High, on the other hand, evaluates to 0.5. So our complete antecedent becomes Max(0.0,0.5) which evaluates to 0.5. This being non-zero, makes our antecedent true and our modifier 0.5. This modifier is then passed to both component consequents in our complex consequent Then Movement_Speed dec 5 And Caution ismore High 0.5. We begin by handling Then Movement_Speed dec 5 by multiplying the maximum adjustment value of 5 by 0.5 resulting in 2.5. We then subtract 2.5 from the value already stored in Movement_Speed which is 10.0 resulting in Movement_Speed being 7.5. We then move on to handle Caution ismore High 0.5. We begin by multiplying the maximum adjustment value of 0.5 by the modifier 0.5 resulting in 0.25. We then increase the value in Caution's High membership field by .25. The value in Caution's High membership field turns out to be 0.1 and so we add .25 to it resulting in .35. An example of this is seen in 6.6.

Initial values:
FloatyVariable Is_Insurgent_In_Range = 0.0
FloatyVariable Movement_Speed = 10.0
FuzzyVariable Caution = FuzzyVariable(0.0,0.0,0.1)
FuzzyVariable Threat = FuzzyVariable(0.0,0.6,0.5)

Rule:
If Is_Insurgent_In_Range = 1.0 Or Threat is High Then Movement_Speed dec 5 And
    Caution ismore High 0.5

Evaluation:
If Is_Insurgent_In_Range = 1.0 Or Threat is High Then Movement_Speed dec 5 And
    Caution ismore High 0.5

If Max(Is_Insurgent_In_Range = 1.0, Threat is High) Then Movement_Speed dec 5 And
    Caution ismore High 0.5

If Max(0.0, 0.5) Then Movement_Speed dec 5 And Caution ismore High 0.5

If 0.5 > 0 Then Movement_Speed dec 5 And Caution ismore High 0.5

If True (0.5) Then Movement_Speed dec 5 And Caution ismore High 0.5

If True Then (0.5) Movement_Speed dec 5 And (0.5) Caution ismore High 0.5

If True Then Movement_Speed =10.0 – (5 x 0.5) And (0.5) Caution ismore High 0.5

If True Then Movement_Speed =10.0 – (2.5) And (0.5) Caution ismore High 0.5

If True Then Movement_Speed =7.5 And (0.5) Caution ismore High 0.5

If True Then Movement_Speed =7.5 And Caution = FuzzyVariable(0.0,0.0,0.1 + (0.5 x 0.5))

If True Then Movement_Speed =7.5 And Caution = FuzzyVariable(0.0,0.0,0.1 + 0.25)

If True Then Movement_Speed =7.5 And Caution = FuzzyVariable(0.0,0.0,0.35)

(6.6)

6.4 Fuzzy Path Finder

The Fuzzy Path Finder is the brain behind the agent. It makes all of the decisions about where the agent will go and how it will get there. It uses the Fuzzy Logic System to examine potential waypoints.

6.4.1 Fuzzy Path Generation

Path generation begins when the Fuzzy Path Finder's generatePath function is called. This function takes two important parameters which are the starting x and y location of the agent. With this information we begin building a path towards the other side of the map.

6.4.1.1 Waypoint Generation

Each potential next waypoint starts as a randomized offset from the current waypoint. To optimize speed, if a certain cardinal direction from the current waypoint is next to a wall then any offset in that direction is prevented. The offset can be as far as 130 distance units, or about a twelfth of the map, from the current waypoint. This maximum offset can shrink though, given that the path to the new waypoint involves a collision with a wall, IED, or suspected IED zone. Before going any further a generated waypoint is tested to make sure it is not in a collision zone, such as inside a house. A simplified version of this is seen in 6.7. If it passes this test, then the waypoint is scored using a set of Fuzzy Logic System rules.

```
getPotentialWayPoints(x0, y0)
        WayPoint PotentialWP[20]
        Counter = 0
        While ( counter < 20) {
                WP = WayPoint(x0 + randX, y0 + randY) // RandX&Y can be negative
                If ( !getCollision(WP) ) Then
                        PotentialWP[counter] = WP
                        Counter++
        }
        Return PotentialWP
```

(6.7)

6.4.1.2 Fuzzy Rules

The rules used to score the waypoint attempt to get the agent to its destination as quickly as possible without allowing it to pass through any dangerous locations. The first rule (6.8a) is probably the most important in that it tells the agent to get to the other side of the map. It does this by gradually increasing the map score as the location gets closer to the goal. The rules two(6.8b) and five attempt(6.8e) to make the agent somewhat gracefully reach the other side of the map. Rule five does this by lowering the score of locations around the city that would slow down the agent's progression around it. Rule two does this by lowering the score of locations inside the city that would slow down the agent from getting out of it. Rule four(6.8d) helps the agent chose environments that are easy to travel, although, it only hints at this because sometimes it's best to take the road less traveled. Rule six(6.8f) strongly discourages the agent from getting in the water (as it shouldn't anyway) by decreasing the map score. Rule seven(6.8g) encourages the agent not to just hang around the same place by decreasing the value of areas that it has already been. This rule greatly aids in the agent not getting stuck behind a wall for example. Sometimes this rule does lead to some strange looking path choices however. Rules three(6.8c) and eight(6.8h) make sure that the agent stays out of harm's way by strongly devaluing locations that either contain visible IEDs or seem suspicious based on insurgent behavior. These rules can be seen in 6.8 in their true form.

If DistanceToGoal is Low Then MapScore inc 1 (6.8a)

If City_Proximity is High And City_Exit is High Then MapScore dec .075 (6.8b)

If IED_Proximity is High Then MapScore dec .2 (6.8c)

If Man is Low Then MapScore dec .015 (6.8d)

If Smooth is High Then MapScore dec .025 (6.8e)

If Water is High Then MapScore dec .08 (6.8f)

If AgentHistory is High Then MapScore dec .05 (6.8g)

If InsurgentThreat is High Then MapScore dec .2 (6.8h)

(6.8)

6.4.1.3 Waypoint Generation Continued

The waypoint's score is really the score of the path from the previous waypoint to that waypoint. This is calculated by adding up the score of each discrete location in a straight line on the way to the waypoint. Once the waypoint location is reached, this value is divided by the number of discrete spaces (steps) it had to move to get there. After the waypoint is scored, the waypoint and its score are placed in an array that holds twenty waypoints. This entire process is then repeated twenty times or until an attempt limit is reached. If the attempt limit is reached and there are at least five potential waypoints then the process proceeds. Otherwise, it continues looking for more waypoints.

Once this process is done, all five to twenty waypoints are sorted by map score. Next, one of the top five waypoints is randomly selected and returned. A simplified example of this is shown in 6.9.

```
generateNiceWayPoint(x0,y0)
    Counter = 0
    WayPoint TempWP[20]
    TempWP = getPotentialWayPoints(x0,y0)
    While (Counter < 20) {
            scoreWP(x0,y0,&TempWP) //Passed by reference. Score field is updated
            Counter++
    }
    Return  GetGoodWP(TempWP) // This randomly returns one of the top five
```

(6.9)

At this point the waypoint must pass some final scrutiny before it is successfully added as the next waypoint. The waypoint is now tested using a modified Bresenham line drawing algorithm that makes sure the path to the waypoint does not pass through any undesirable locations such as visible IEDs zones, collisions zones, or suspected IED zones. If the waypoint passes the test then it is selected as the next waypoint and the entire process moves on to finding the next waypoint. If it fails then the waypoint is thrown out and a new one is generated.

This entire process, which can be seen in 6.10, is repeated until a waypoint at the other side of the map is reached.

```
generatePath(StartX, StartY)
        Counter = 0
        WayPoint Path[300]
        WayPoint TempWP
        Path[Counter] = WayPoint(startx,starty)
        While (Path[Counter].Y  > Destination_Y){
                TempWP = getNiceWayPoint(Path[Counter].X, Path[Counter].Y)
                If (checkHazards(x0,y0,TempWP)) Then
                        Counter++
                        Path[Counter] = TempWP
                        increaseRandXY() // increases RandX and RandY up to a cap
                Else
                        decreaseRandXY() // decreases RandX and RandY down to a cap
        }
        Return Path
```

(6.10)

6.5 Exposed IED Marking

Earlier we saw the algorithm for detecting an exposed IED. When this occurs, the FuzzyHeatMap "IED_Proximity" has its High Bitmap updated. It is updated using a simple algorithm that draws a circle about double the size of the kill radius around the spotted IED. The value for High field decreases radially as the distance from the IED increases. More specifically, at the just beyond the edge of the circle the value is zero and the value right at the IED location is one. This is modeled in 6.11.

```
markIED(x0, y0)
        //DR = Danger Radius
        //TV = Temp Value (a float)
        For( I = -DR ; I <DR+1 ; I++)
                For( J = -DR ; J <DR+1 ; J++)
                        TV = distanceBetween(x0,y0,x0+I,y0+J)
                        If TV < DR Then
                                TV = 1 – (TV/DR)
                                IEDHeatMap.setValue(x0+I,y0+J,TV)
```

(6.11)

Once this height map is updated, a check is then performed to see if our current path plan leads us into this danger zone. If it does, a new path is generated. We can see an example of this in the output maps in Figures 7.1 and 7.2 below. We see that in 7.1 the IED danger radius came too close to the agent's path and so a new path shown in 7.2 is generated. Otherwise our course stays the same.

Figure 7.1 Before IED Detection

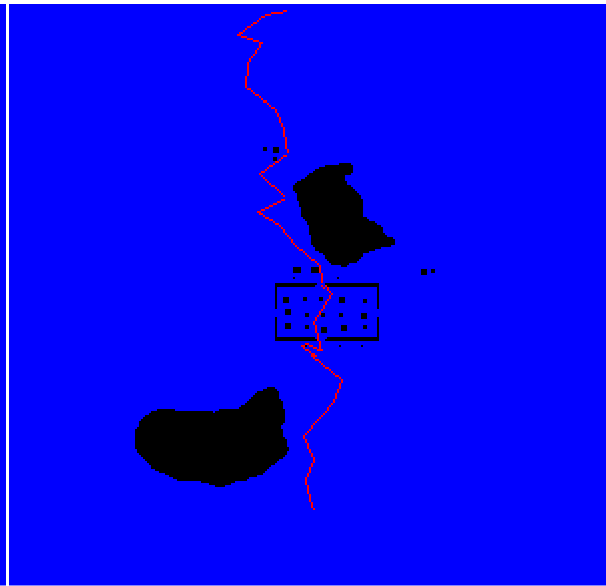


Figure 7.2 After IED Detection

6.6 Suspected IED Marking

Suspected IED marking works in a similar fashion as exposed IED marking. The main difference is that instead of getting information of the IEDs whereabouts from the IED itself, we get the information from the insurgent and the insurgent model. Aside from that, the only difference is that a different FuzzyHeatMap named "InsurgentThreat" is marked and the danger radius is much larger. Specifically, the danger radius for an exposed IED is merely 40 distance units as compared to the danger radius of a suspected IED which is 175 distance units. The reason for the dramatic difference is that with an exposed IED we know exactly where it is. With a suspected IED, we don't really know where it is other than the fact that it is extremely likely that it is within visible distance of us and the insurgent. 175 distance units is about the average maximum visible distance for the agent and insurgent to see each other and so it seems to be a good choice.

Chapter 7 - The Insurgent model

The insurgent model allows the agent to discover possible hidden IED locations solely based on insurgent state information.

7.1 Deriving Useful Insurgent States from Action Sequence Data

In order to determine what insurgent states were relevant for the insurgent model, I decided to make a "Match Player" so that I could re-watch matches using game log data. Originally, it simply displayed, in 2D, the insurgent's location and the agent's location overlaid on the game map. Over time, I added more features and a status window which output lots of information about the insurgent and his surrounding environment until I was satisfied that I was capturing enough useful information. This in turn corresponded to capturing more action sequence data in the game logs as well. The final version can be seen below in a Figure 8.

Figure 8 Screenshot of the Match Player

Everything displayed on the screen in the match player is derived from the game logs and can also be calculated in real time during a match. For example, to tell if the insurgent is approaching or fleeing the agent we start with the insurgent's 2D movement vector. The line orthogonal to this vector is computed and shifted such that it intersects the insurgent's location. We then move this orthogonal line to the location of the agent. If this movement is in the same direction as the insurgent's movement vector then we know that the insurgent is approaching the agent. Otherwise, we know that the insurgent is fleeing. The exact code of how this works is a little bit different, but it can be calculated very quickly (constant time).

7.2 Important States

To keep the insurgent model, a transition matrix, relatively small and tractable we need to select a limited number of states that convey important, but not superfluous, information about what the insurgent is doing. After watching numerous games with the Match Player I decided that the following `states` capture the important state information about the insurgent.

OutOfRange – This state means that the agent and insurgent are out of visible range of each other and neither one can directly observe the other's actions.

Plant – This state indicates that the insurgent is planting an IED.

Detonation – This state indicates that the insurgent is detonating an IED.

Monitor – This state indicates that the insurgent is stationary and looking over an area. The Insurgent must be in range to enter this state.

Approach – This state indicates that the insurgent is moving toward the agent. The Insurgent must be in range to enter this state

Flee – This state indicates that the insurgent is moving away from the agent. The Insurgent must be in range to enter this state.

7.3 Important SubStates

Each state has a certain number of `substates` that further describe what the insurgent is doing in a given state.

Base – Each state has this as a substate. It is representative of the insurgent not being in any special circumstances for a given state. This is the only substate that OutOfRange has. This substate in Detonation indicates that the agent was within the kill radius of the IED at the time of detonation. In every other state this substate indicates that the agent is within 80% of the average adjusted maximum visual distance and that the agent is visible in the insurgent's viewing frustum.

Close – Detonation is the only state with this as a substate. This substate indicates a near miss in which the detonation of an IED was not within the kill radius of the agent; however, it was within 40 distance units of the agent.

Insig – Detonation is the only state with this as a substate. This substate indicates that an IED was detonated, but it was beyond 40 distance units from the agent and therefore insignificant.

The remaining substates, and all combinations of them, are substates of Plant, Monitor, Flee, and Approach.

Averted – This substate indicates that the insurgent is not looking in the direction of the agent. Specifically, the agent is not within the viewing frustum of the insurgent.

Distant – This substate indicates that the insurgent is over 80% of the average adjusted maximum visual distance. This is important because IED planting and IEDs cannot be detected at this distance.

Obstructed – This substate indicates that there is no line of sight between the agent and insurgent. Specifically, there is some piece of geometry such as a wall or hill that is blocking the line of sight.

7.4 The Transition Matrix

The final transition matrix, or insurgent model, has a total of 33 state and substate combinations. This gives us a total of 33^2 or 1089 transitions in the insurgent model. This may seem like a lot, but it can be stored in an easy to read text file that is only 30 KB in size which can partially be seen below. With the leftmost column being the "From" state / substates and the top row being the "To" state / substates, it is easy to visually access model information. The class that contains the insurgent model has functions that can easily save and load the raw transition matrix in this format as well.

TransitionMatrix - Notepad

517

To / From	Detonation_Base	Detonation_Close	Detonation_Insig
Detonation_Base	0	0	0
Detonation_Close	0	0	0
Detonation_Insig	0	0	0
Monitor_Base	4	0	0
Monitor_Obstructed	0	0	0
Monitor_Averted	0	0	0
Monitor_Distant	0	0	0

Figure 9 A small piece of the transition matrix

Each value in the matrix stores the total times that the insurgent went from a certain state / substate to another. The number at the top left of the text file is the total steps that have been stored in the raw transition matrix.

7.5 Calculating Probabilities

The values in the raw transition matrix, or insurgent model, are of little use in their current form as they are not normalized or in probability form. At run time this is fixed using the process described in section 3.1. These new values are stored in our probability transition matrix.

7.6 Using the Insurgent model

We want to use our Insurgent model, or transition matrix, to see when the insurgent is doing something that statistically leads to a successful (for the insurgent) detonation. One way of doing this is to check the probability that the insurgents current state / substate leads to the state / substate Detonation Sub Base. This can be completed in constant time as we are simply looking up a value in four dimensional array as seen in 7.1.

Danger = TransitionMatrix[Current_Ins_State][Current_Ins_SubState] [Detonation][SubBase]

(7.1)

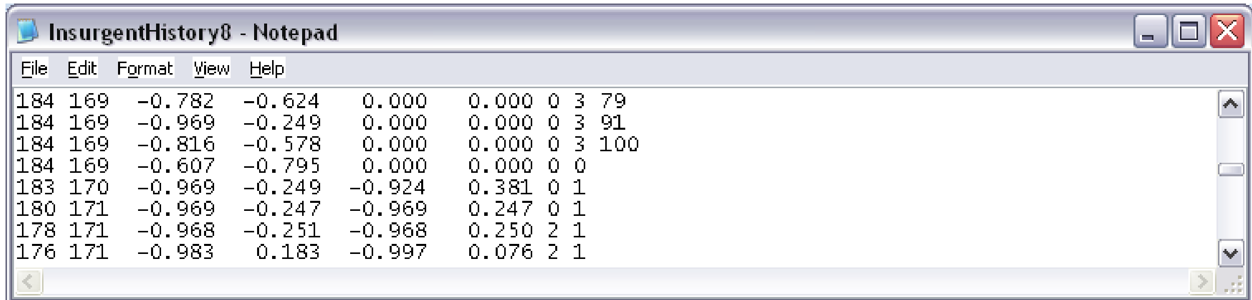
If the value returned is non-zero then we know statistically what percent of the time that the insurgent goes from his current state / substate to making a successful detonation. How to respond to this is up to the agent or person, but this is the likeliness of being within the kill radius of a detonated IED in the next second. Later, in the results section, we will cover how the agent handles this.

7.7 Action Sequence Data

Before we can use the Insurgent model, we need to first fill it with action sequence data. This action sequence data is stored in game logs in text form. There are two files generated for each match. The first file is the insurgent's game log. It's important to note that in order to make game logs manageable that distance units are divided by five in order to allow them to be consistent with heat maps, heightmaps, and other game data was downscaled for simplicity.

The insurgent's game log begins with the total number of discrete steps, specifically seconds, that the match was. After this a new line is started and a specific format takes over. The line starts with the insurgent's x and y position in integer form. Followed is the insurgent's 2D vision vector stored to three decimal place float precision with the x component y component. This is followed by the insurgent's 2D movement vector information stored in the same way. Followed is an integer that can be zero to three. This value represents the insurgent's vision information. Zero means that the agent is out of range. One means the agent is within viewing range, but is being obstructed by some geometry. Two means the insurgent is looking at the agent. Three indicates that the agent is in range, but the insurgent is not looking at him. The next number is also an integer, but it ranges from zero to seven. Zero represents the insurgent standing still. One means the insurgent is moving. The values two through four represent planting the IEDs one through three respectively. The values five through seven represent detonating the IEDs one through three respectively. If the insurgent is not planting an IED then the line ends here. If the insurgent is planting an IED, an integer representing the percent planted is the last value on the line. This pattern is repeated until the duration of the game is logged.

Following the last line of this pattern is a single integer ranging from zero to four. Zero represents victory for the agent. One represents victory for the insurgent by IED detonation. Two represents victory for the insurgent by time running out. Three indicates that the player exited the game before it was over typically indicating they felt they could not win. Four represents that the connection between the server and client was lost. An example section of game log for the insurgent can be seen in Figure 10.

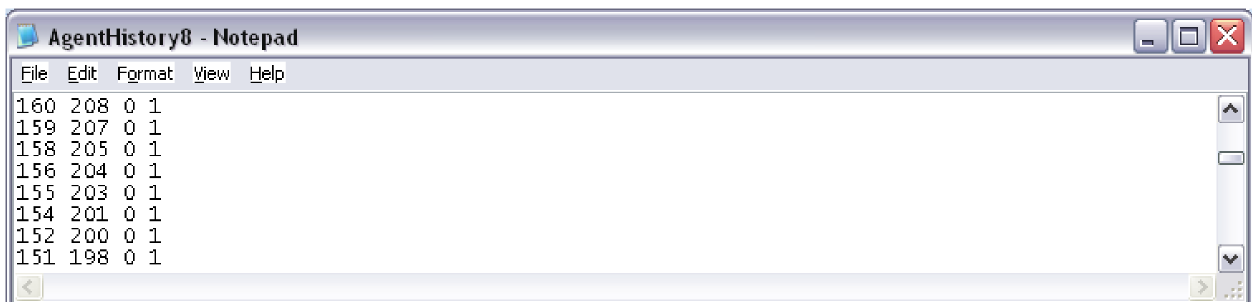

InsurgentHistory8 - Notepad

File Edit Format View Help

```
184 169  -0.782  -0.624   0.000   0.000 0 3 79
184 169  -0.969  -0.249   0.000   0.000 0 3 91
184 169  -0.816  -0.578   0.000   0.000 0 3 100
184 169  -0.607  -0.795   0.000   0.000 0 0
183 170  -0.969  -0.249  -0.924   0.381 0 1
180 171  -0.969  -0.247  -0.969   0.247 0 1
178 171  -0.968  -0.251  -0.968   0.250 2 1
176 171  -0.983   0.183  -0.997   0.076 2 1
```

Figure 10 Insurgent Action Sequence Data

The agent's game log is much simpler. It is a repetition of the following information for the duration of the game. The agent's x and y location are stored as integers. Following this is a binary integer representing if there was line of sight between the agent and insurgent. Zero indicates there was while one indicates there was an obstruction. The final value on the line is also a binary integer which indicates whether the insurgent was "Distant" or not; zero meaning no and one meaning yes. Distant simply means that the agent was over 80% of average adjusted maximum visual distance away from the insurgent. An example can be seen in Figure 11.


AgentHistory8 - Notepad

File Edit Format View Help

```
160 208 0 1
159 207 0 1
158 205 0 1
156 204 0 1
155 203 0 1
154 201 0 1
152 200 0 1
151 198 0 1
```

Figure 11 Agent Action Sequence Data

7.8 Post Mortem Propagation

With action sequence data and the insurgent model laid out, we just need to load our action sequence data into our insurgent model. We really want to see what leads up to successful insurgent detonation and even though we want to gather as much information as possible, only the game logs that contain action sequence data of an IED-detonated-insurgent-win are really useful to us. One could use action sequence data from agent wins to learn about what cues an insurgent gives that indicate you're not near an IED, but that's a future project.

Given the action sequence data stored in the game logs it's easy to either directly read or quickly derive the insurgent's state / substate at any given time. As such, we just wait until we have a decent set of action sequence data from matches where the insurgent wins via IED detonation, and then we create or update our insurgent model with this new data. Each second or step simply adds one to the correct "To/From" location in the raw transition matrix and then the raw transition matrix is resaved. At this point, the raw transition matrix can be converted into a normal transition matrix and then used by the autonomous agent.

Chapter 8 - Game Engine Components

8.1 Off the shelf components

We want to make a game that is as realistic as possible and a great step in that direction is using 3D graphics. Three dimensional graphics are also the standard for video games today and we want to ensure that our game is appealing to as many people as possible. OGRE, which stands for Object-Oriented Graphics Rendering Engine, is an open source 3D graphics engine [8]. It is a relatively easy to use, well documented engine with a large community of programmers and developers. It is made available under the GNU Lesser General Public License (LGPL) which also makes it ideal for this project as it is free to use. OGRE is a great choice as it allows for a quick implementation of 3D graphics with relatively little knowledge of 3D graphics theory or complex libraries.[8]

One of the aspects that will make the action sequence data collected from this game so useful is that the game can be played from around the world. To accomplish this we need to

make the game playable over the internet. RakNet is an open source game networking engine [9]. Similar to OGRE, it makes a normally complex and time consuming component of a game quick and easy to implement. It also is free to use, for our purpose at least, and has a substantial community of users that offer free assistance. Using RakNet, We can implement a client server model for our game with minimal effort. [9]

With OGRE handling graphics and RakNet handling networking all that's left to do is handle keyboard/mouse input and action sequence data. OGRE actually has built in keyboard and mouse event handling so this is already taken care of for us. The action sequence data can be saved using a simple C++ file stream as it does not need to be in any proprietary format.

8.2 Custom built components

With the exception of OGRE and RakNet all other components were custom built using Visual Studio C++ 2008 Express [13]. This includes the client, server, autonomous agent (and all of its sub components), and match player. I should note that I did use EasyBMP to make manipulating Bitmaps easier [11]. All 3D models either came stock with OGRE or were taken from Google SketchUp's 3D Warehouse. The height map used in the game world was created using Nem's Mega 3D Terrain Generator [12]. The overall world design and terrain detail were created by me using a combination of MS Paint and C++ code [13].

Chapter 9 - Results

In order to better illustrate the effectiveness of the Insurgent model and the evolution of the agent, the results chapter is broken into three distinct sections. Each section contains results from different agent implementations ranging from simple to complex. The differences in implementation will be elaborated upon in each section.

9.1 Iteration Zero

Iteration zero was the first version of the game and agent. The purpose of this version was to establish a base effectiveness or survivability for a naïve agent only protected by its mounted weapon and primitive path finding abilities. This could be representative of someone

that had an idea of where they were going and the ability to protect them self, but had no understanding of insurgency tactics and would not be able to identify an IED. The agent made no use of the insurgent model as the insurgent model did not exist yet. The agent also had an extremely simplistic set of rules processed by the Fuzzy Path Finder. Specifically, the only rule that it used was the rule which increased map score based on closeness to the goal. The agent in this model was also totally blind to IEDs, even exposed ones, and to insurgent behavior. As mentioned earlier, the agent's gun is functional in this version. An example path it would take can be seen in Figure 12.

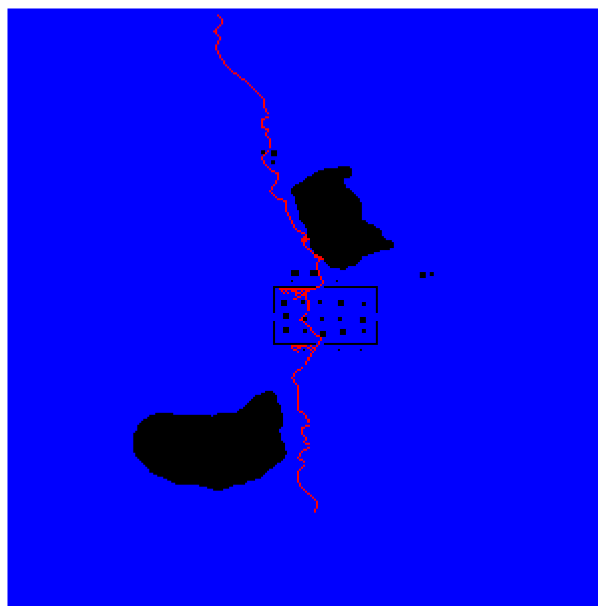


Figure 12 An example iteration zero path

As the picture shows, the agent exhibited poor path choices frequently becoming stuck in or around the city and also had a tendency to get stuck on the edges of the lakes. Although its mounted gun would attack the insurgent if the insurgent got too close, the agent's naïve perspective of the insurgent and IEDs made it an easy target to take down anyway. In fact, the insurgent won over 90% of the time in iteration zero. The agent proved extremely easy to destroy as it basically ignored everything except when the insurgent got too close. To destroy the agent many players played the role of the insurgent by simply first finding the agent and then

placing all three of the IEDs in a line in front of the agent. Almost always the agent would travel within the kill radius of one of the IEDs. In the rare case when the agent won, it was almost always because the agent took a strange path and/or the player controlling the insurgent was new to playing the game and was initially confused about the agent's starting location. In short, almost all of the matches in iteration zero were under one minute and resulted in the agent's demise.

9.2 Iteration One

9.2.1 Iteration One Version A

The purpose of iteration one is to model someone that:

1. Can protect them self directly from an insurgent with a weapon
2. Knows where they're going
3. Can identify visible IEDs
4. But does not have an understanding of insurgent behavior.

This version of the agent still does not make use of the insurgent model. The agent in this version, however, does have substantial upgrades from iteration zero. Most importantly it can identify exposed IEDs and then plot a course around them. This agent also has an extended rule set which greatly increases its ability to maneuver around lakes and in and around the city as well. It also rates locations with faster mobility slightly higher as well.

At first this version showed a remarkable improvement over iteration zero. Agent deaths decreased from 90% to below 25%. An example of the agent avoiding an exposed IED has already been shown earlier, but it can be seen again below in Figures 7.1 and 7.2.

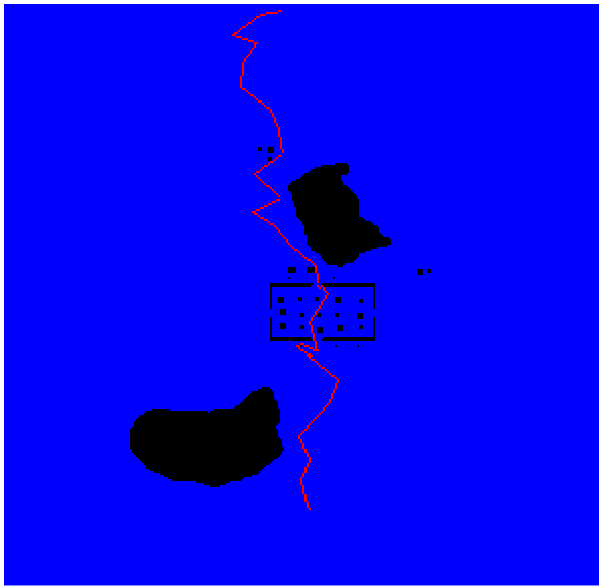


Figure 7.1 Before IED Detection Figure 7.2 After IED Detection

However, after a while, I discovered it had a significant flaw. The flaw, which can be seen in the path output in Figure 13, is that the agent was extremely susceptible to being boxed in by visible IEDs and then running out of time.

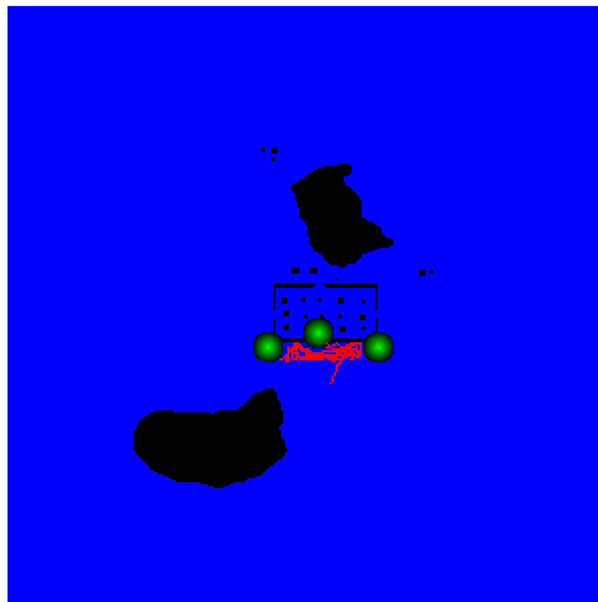


Figure 13 Iteration one version A path finding problem

I did not feel this accurately modeled what a somewhat capable agent or person would do. Although iteration zero also had a similar problem by getting stuck in the city, it was so susceptible to being killed in so many other ways, it really didn't matter.

9.2.2 Iteration One Version B

Iteration one version B is identical to iteration one version A except that the agent is capable of circumventing being boxed in. I accomplished this by adding an additional FuzzyHeatMap and accompanying rule that devalued the immediate area around a waypoint after it was used. This functions much the same way as how the area around an exposed IED is devalued, except to a lesser degree. This has the effect of making it so that the agent will not stay in the same place for an extended period of time and will eventually leave an area even if it has to back track. This can be seen in Figure 14 below.

Figure 14 Iteration one version B path finding solution

This mechanism appeared to have an added effect of making the agent slightly more "jumpy" or unpredictable as well. This new version, which was used to generate action sequence data for the insurgent model, had agent deaths at no greater than 20% for the average player. However, this version, simply because it pays no heed to what the insurgent does, still

has its weaknesses. One fairly effective tactic which can be seen below is to funnel the agent into a hidden IED using exposed IEDs. This strategy can be seen in Figures 15.1-15.3 below where the agent is funneled into an entrance of the city where a concealed IED is located.

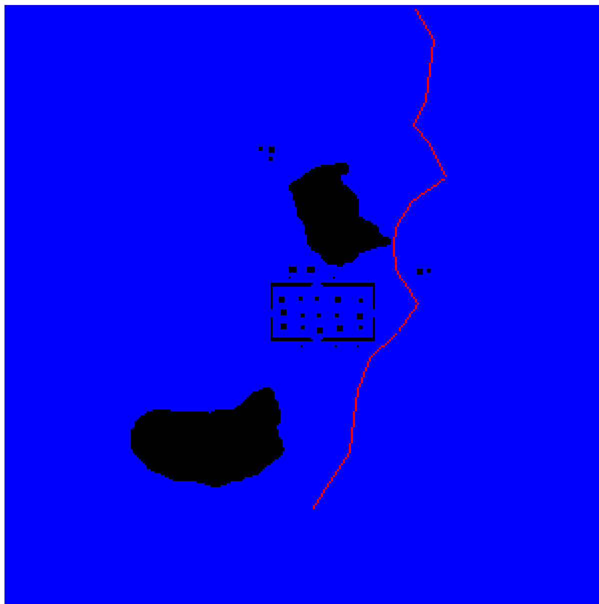


Figure 15.1 Initial Path

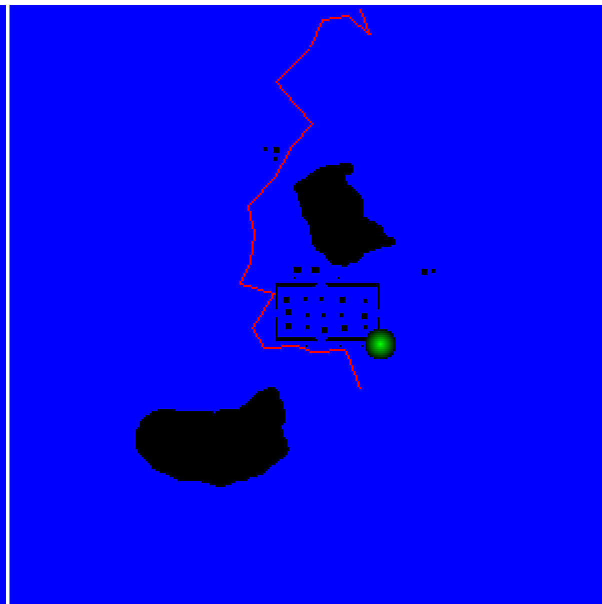


Figure 15.2 Avoiding IED 1

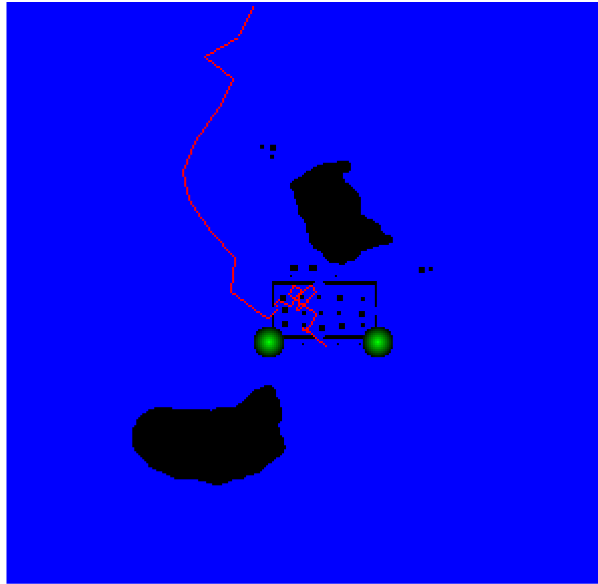


Figure 15.3 Avoiding IED 2 and walking right into hidden IED 3

However, it's also possible for this tactic to fail miserably if the agent decides to circumvent the funnel altogether which is illustrated in Figure 16. With all three IEDs already planted, there's nothing the insurgent can do to stop the agent in this situation.

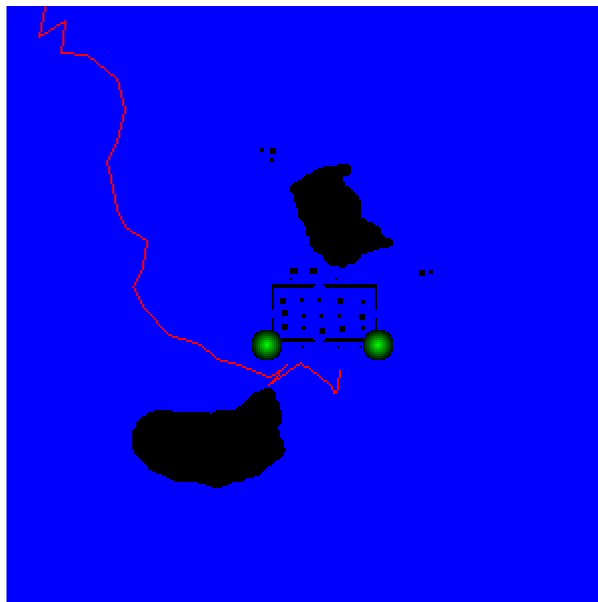


Figure 16 Tactic Failure

9.3 Iteration Two

9.3.1 Iteration Two Version A

The purpose of iteration two is to model someone that:

1. Can protect them self directly from an insurgent with a weapon
2. Knows where they're going
3. Can identify visible IEDs
4. Can use an insurgent model to identify dangerous situations

In many ways iteration two version A is identical to iteration one version B. The only real difference is that the agent finally uses an insurgent model generated from iteration one version B's action sequence data. It uses this, as explained earlier, by looking up the insurgent's current state / substate and checking the probability that a successful detonation will occur. In iteration two version A, if this value is non-zero then it is added to a cumulative threat value. Once a second this is done. If the cumulative threat value reaches a certain threshold then the agent reacts. It marks the area directly in front of it as dangerous, checks to see if this area is in its path, and generates a new path around it if it is. This version was very difficult to kill with a less than 15% death rate for the agent. The problem is that it hardly ever won either due to the agent being extremely paranoid. The problem is apparent in Figure 17.

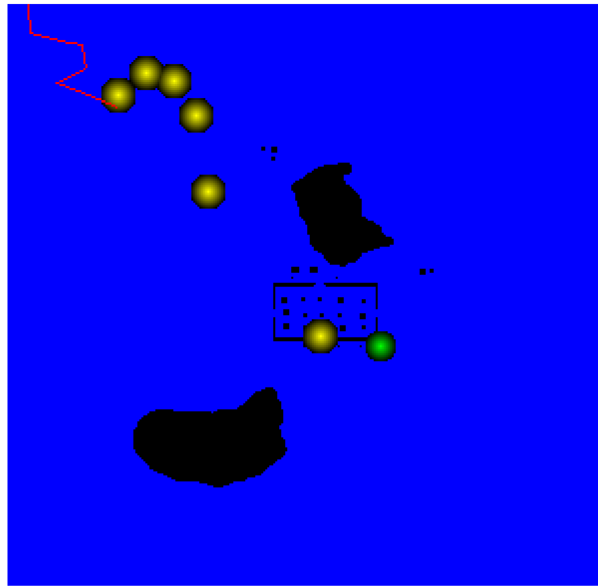


Figure 17 Paranoid Agent

The agent was a bit overly protective of itself and although it rarely walked into an IED it also rarely finished a match. The insurgent could simply sit and stare at the agent, and assuming he positioned himself in the right place, keep the agent from ever crossing a section of the map. Even though the agent appears to be close to winning in the above output path, it ran out of time before it did. This is simply because I sat and stared at it once it got into the hills, and it proceeded to backtrack itself to a loss.

9.3.2 Iteration Two Version B

Iteration two version B is the same as iteration two version A, except in the way that it reacts to threat information from the Insurgent model. The goal in version B was to keep the lowered death rate of version A, but also to improve the likeliness that the agent would win the game. There are five major things I changed in order to achieve this goal.

The first thing I changed was I made the insurgent threat danger radius area much larger than it was before. Before the radius of the danger circle was about a third of the average adjusted maximum visual distance at 80 distance units. I changed it to 175 distance units, over

doubling its original radius. I made this change because it is likely that somewhere in the area that the insurgent is looking over, there is a hidden IED. Rather than repeatedly marking off little pieces of the map at a time, I figured it would be faster to just make off a larger area.

The second thing I did was make it so that after a dangerous insurgent area is marked, another one cannot be marked again until the agent has moved significantly away from the first area. While this may make the agent a little more vulnerable, it's only vulnerable in a small section of the map, not to mention this keeps the agent from building a giant line of distrust across the map.

The third thing I did came from my extensive watching of matches using the Match Player. I noticed that almost always a successful IED attack only happened when the insurgent was closer to the goal side of the map than the agent. So I made it so that if the agent is closer to the goal than the insurgent, then the cumulative threat value could not increase. While this may make the agent vulnerable, it's only in very specific situations that seemed to rarely occur.

The fourth thing I did was make it so that the cumulative threat value would decrease over time if the agent went out of range of the insurgent. This is a slow process so if this only happened for a few seconds it would have little effect, but if the agent had not seen the insurgent in a long time it would basically reset the agent's cumulative threat value.

The fifth and final thing that I did was make it so that if the agent witnessed the insurgent move further away from the goal than him and then move back closer to the goal than him, that until the insurgent either got distant, out of range, or was otherwise visibly obstructed, that the cumulative threat value would not increase. This is just a reaction to the idea that if the insurgent gets behind the agent, until he has a chance to go somewhere that he can plant an IED in front of the agent, without the agent noticing, he probably is of little threat. This is illustrated below.

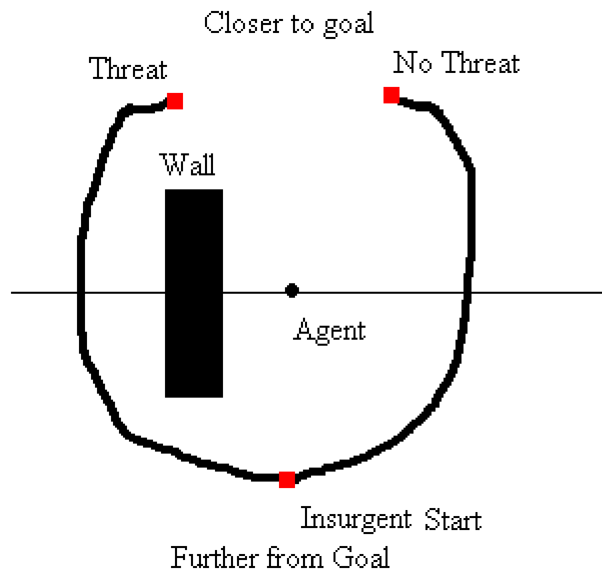


Figure 18 Specific insurgent behavior indicative of threat

With these modifications, I experienced death rates similar to version A, at about 15%, but the agent actually typically reached the goal before time ran out as well. An example of this can be seen in Figures 19.1 – 19.3 below in the agent's progression.

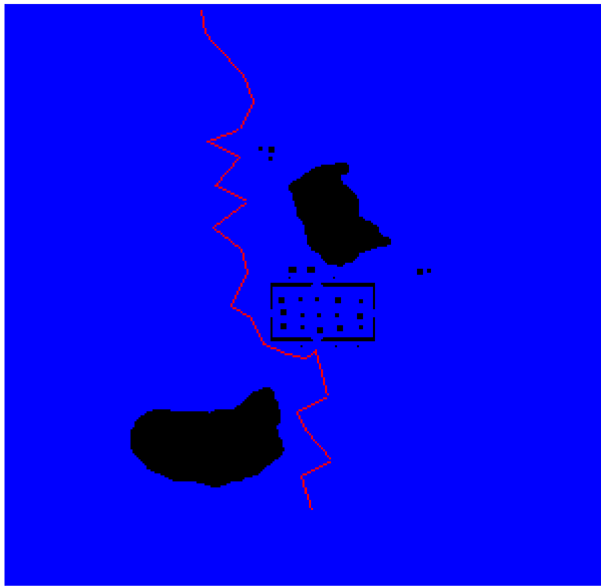


Figure 19.1 Initial Plan

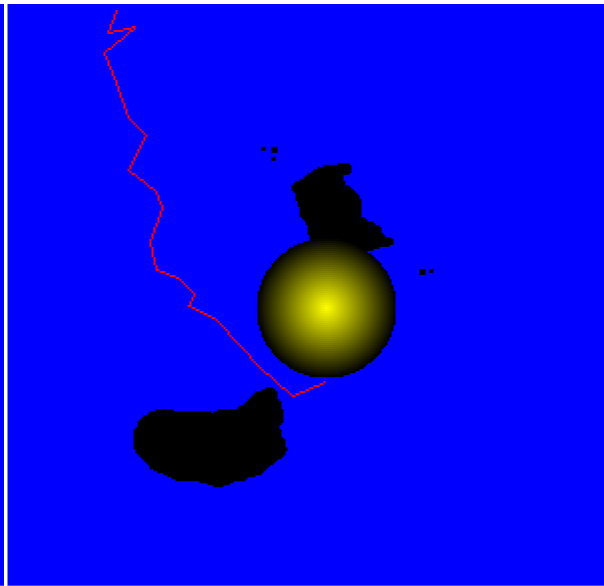


Figure 19.2 Insurgent Threat Detected

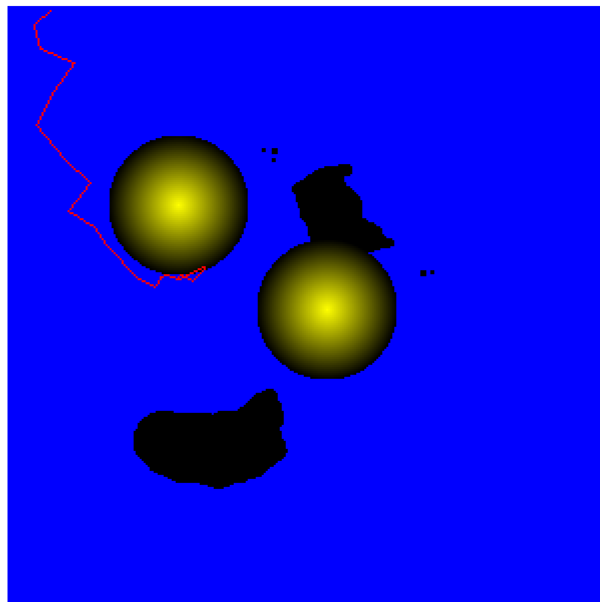


Figure 19.3 Agent Progresses, but still avoids suspicious areas

Chapter 10 - Conclusion

Before any final conclusions are drawn I would like to first clarify why all of the death rates are cleanly divisible by five. This is because I've had many different people play the game as the insurgent and generate action sequence data for me. Some were about as good as me and some weren't. Due to this, for each game log set where someone played at least 10 games and occasionally 20, I chose the game log set that had the highest death rate for the agent. So even though a total of possibly sixty games were played for each version, the data set selected to illustrate its effectiveness is composed of either 10 or 20 games. This being the case, any number of victories for the insurgent divided by 10 or 20 is going to result in a percentage cleanly divisible by five.

10.1 A Brief Caveat

With that understood I would also like to indicate that given different insurgent players, myself included, that these death rates could be different. These are just the rates that I experienced while hosting the game. I make no claims of being a skilled insurgent myself or being friends with skilled insurgents.

10.2 A Step in the Right Direction

The Insurgent model and autonomous agent both seem to be reasonably functional and effective to their purpose. The insurgent model, while limited to the virtual world, captures insurgent cues and relates them to what the insurgent is likely to do next. It definitely seems to have an improvement on survivability and strategy when used by the agent. In time, a real world analog could be developed and used to save lives. The agent also demonstrates great potential. While a real world analog would need to be quite different and more advanced, the idea of replacing non combat soldiers on major supply routes could definitely save lives as well.

10.3 40 Games, 5 wins, and the Future

Iteration one version B supplied the game logs containing the action sequence data that was fed into the Insurgent model used in iteration two. This data came from a total of 40 games in which there were only a total of five wins. These five wins were fed into the raw transition matrix making a total number of 517 steps. In those eight and a half minutes there were two state / substate sets that linked to successful detonations. The first one was Monitor SubBase with a 7% chance of leading to a successful detonation. The second one was the Flee SubBase with a 4% chance of leading to a successful detonation. With more time and games played, it would be interesting future work to see how this matrix would look with hundreds of game logs worth of action sequence data to reference. Other interesting future work would be looking multiple steps back to see if there were a certain series of state / substates that would give a more accurate reading of whether or not an IED was being watched over in proximity to the agent.

References

1. Stockfish, David; Yariv Eldar, Daniella HarPaz Mechnikov (1970). Dokszyc-Parafianow Memorial Book — Belarus (Sefer Dokshitz-Parafianov). Tel Aviv: Association of Former Residents of Dokszyce-Parafianow in Israel. p. 274. http://www.jewishgen.org/yizkor/dokshitsy/dok274.html.

2. Whitlock, Craig (2010, March 18). Soaring IED attacks in Afghanistan stymie U.S. counteroffensive. *The Washington Post.* Retrieved from http://www.washingtonpost.com

3. Thompson, Clive (2007, August 21). Halo 3: How Microsoft Labs Invented a New Science of Play. *Wired*. 15(09). Retrieved from http://www.wired.com/gaming/virtualworlds/magazine/15-09/ff_halo?currentPage=3

4. Kennerly, David (2003, August 15). Better Game Design Through Data Mining. Gamasutra. Retrieved on April 21, 2010 from http://www.gamasutra.com/

5. (2007, August 21). Virtual game is a ‘disease model’. BBC News. Retrieved on April 21, 2010 from http://news.bbc.co.uk/

6. Markov chain. In *Wikipedia*. Retrieved April 21, 2010, from http://en.wikipedia.org/wiki/Markov_chain

7. El-Sharkawi, Mohamed. *Fuzzy System and Control* [PDF slides]. Retrieved from http://cialab.ee.washington.edu/index_files/tutorial/fuzzy.pdf

8. Torus Knot Software. (2010, April 25). Object-Oriented Graphics Rendering Engine (Version 1.6.1) [Software]. Available from http://www.ogre3d.org/

9. Jenkins Software LLC (2010). RakNet (Version 3.713) [Software]. Available from http://www.jenkinssoftware.com/raknet/index.html

10. Probability theory. In *Wikipedia*. Retrieved April 21, 2010, from http://en.wikipedia.org/wiki/Probability_theory

11. The EasyBMP Project. (2006). EasyBMP [Software]. Available from http://easybmp.sourceforge.net/

12. Nem’s Tools (May 2003). Nem’s Mega 3D Terrain Generator [Software] Available from http://nemesis.thewavelength.net/

13. Microsoft (2008). Visual C++ 2008 Express Edition [Sofware]. Available from http://www.microsoft.com/express/Windows/

14. Fuzzy logic applications (n.d.). Retrieved April 21, 2010, from http://www.logicaldesigns.com/LDFUZ1.htm

15. Turaga, Pavan, Chellappa, Rama, Subrahmanian, V.S., and Udrea, Octavian (2008). Machine Recognition of Human Activities. IEEE Transactions on Circuits and Systems for Video Technology. 18(11), 1473-1488

16. Lavee, Gal, Rivlin, Ehud, and Rudzsky, Michael (2009). Understanding Video Events: A Survey of Methods for Automatic Interpretation of Semantic Occurrences in Video. IEEE Transactions on Systems, Man, and Cybernetics. 39(5), 489-504

17. Huang, Kaiqi, Wang, Shiquan, Tan, and Maybank, S.J. (2009). Human Behavior Analysis Based on a New Motion Descriptor. IEEE Transactions on Circuits and Systems for Video Technology. 19(12), 1830-1840