2005

Adaptive behavior for fighting game characters

Leo Lee
San Jose State University

ADAPTIVE BEHAVIOR FOR FIGHTING GAME CHARACTERS

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Leo Lee

May 2005

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

[illegible]

Dr. Christopher Pollett

[illegible]

Dr. Rudy Rucker

Jeffrey D. Smith

Dr. Jeffrey Smith

APPROVED FOR THE UNIVERSITY

[illegible]

ABSTRACT

ADAPTIVE BEHAVIOR FOR FIGHTING GAME CHARACTERS

by Leo Lee

This project attempts to show the practicality and effectiveness of machine learning in video game Artificial Intelligence (AI) by developing an adaptive AI system for a fighting game called *Alpha Fighter*. The use of sequential prediction, n-grams, hidden Markov models, and dynamic Bayesian networks are discussed in the context of the game and considered for their suitability for the AI system. The design and implementation of the AI system as well as more notable aspects of the game itself are also discussed. Results on a survey of the effectiveness of the adaptive and non-adaptive versions of the AI are given, showing that the adaptive AI was perceived to be more intelligent and challenging. The adaptive AI was also perceived to provide slightly better game-play.

ACKNOWLEDGEMENTS

I would like to thank my mom, dad, and sister, Si Si, for their support in countless ways and on countless occasions. I would also like to thank my advisor, Dr. Chris Pollett, for his suggestions and guidance, Dr. Rudy Rucker for his help and support – even after his retirement, and Dr. Jeff Smith for his time and effort.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

There has been a tremendous advance in computer graphics over the three decade history of the video game industry. Games have gone from using simple 2D geometric shapes, to surrounding the gamer in immersive 3D environments. A multitude of new computer graphics techniques have been used successfully in commercial games. For example in 1993, one of the earliest and most successful first person shooters, *Doom*, used binary space partitioning (BSP), a method for fast visible surface detection in a static environment. Recent games, such as *Doom 3* and *Half-Life 2*, have been making extensive use of techniques like normal mapping, which provides characters and objects with high levels of detail without the need for a high number of polygons in the meshes. This is achieved by explicitly providing the surface normal vectors throughout a mesh in the normal map. Since normal vectors are used to calculate shading for the surfaces of a mesh, the level of detail is proportional to the number of normal vectors specified.

Artificial Intelligence (AI) in video games has not evolved nearly as much within these three decades. Although many AI techniques have been used in games, the majority of games have relied on only the simplest techniques, such as finite-state machines and decision trees [Tozour 2002]. A list of commonly used game AI techniques is presented in [Rabin 2004]. Although these techniques can be effective, they are mostly static in nature; that is, all behaviors must be predefined by the developer. On the other hand, another list is presented in [Rabin 2004] of promising game AI techniques that have seen little use in practice. These techniques include neural networks, genetic algorithms, and reinforcement learning, the majority of which are various forms of machine learning. Machine learning is the next stage in the evolution of game AI.

In a recent article about the 2005 Game Developers Conference [Sanchez-Crespo 2005], Sanchez-Crespo stated a lack of machine learning in games, citing that "characters in a fighting game will seldom adapt to our fighting style." The goal of this project was to remedy this problem by creating an AI system for a one-on-one fighting game, *Alpha Fighter,* which would allow the non-player character (NPC) to learn and adapt to the dynamically changing fighting style of a player. The main purpose was to realize the practicality of machine learning for game AI, as well as its potential to enhance game-play. It is believed that an adaptive AI would enhance game-play, because the player would be less likely to become bored if the behavior of the NPC is dynamically adjusting to the player, as opposed to just being static.

The rest of this paper is structured as follows: Section 2 presents some of the AI research done during the early part of this project. Section 3 lays out the requirements of this project. Then, the high level design and implementation details are explained in Sections 4 and 5 respectively. In Section 6, some results on the effectiveness of the AI system are given. Finally, Section 7 provides concluding remarks and possible future work on ways to extend and improve the AI system.

2. BACKGROUND

This section discusses the more notable AI research conducted during the early stages of this project. Four major types of statistical AI models were studied. These four types of statistical models will only be discussed as they relate to the goal of building an adaptive fighting game AI system. Detailed requirements for such a system are discussed in Section 3.1. However, it should be mentioned briefly before presenting the models, that the model must be easily trainable and make predictions efficiently and robustly. The reader may want to examine [Russell and Norvig 2003] for more general information about these models.

2.1 Sequential Prediction

Sequential prediction is a technique that basically performs prediction by using a string matching algorithm [Mommersteeg 2002]. A sequence of events in a game can be viewed as a string, where each unique character represents a particular type of event, and event[i] occurs before event[j] for all $i < j$. In other words, events are queued chronologically. In sequential prediction, the goal is to find the end index of the longest pattern match, p, such that p matches a proper suffix, s, of the entire string, and p and s are distinct substrings. Let p_{last} be the end index of p. Based on the history of events so far, the most probable next event must be event[p_{last} + 1] (Figure 1).

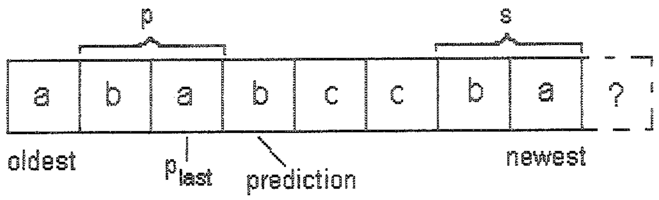
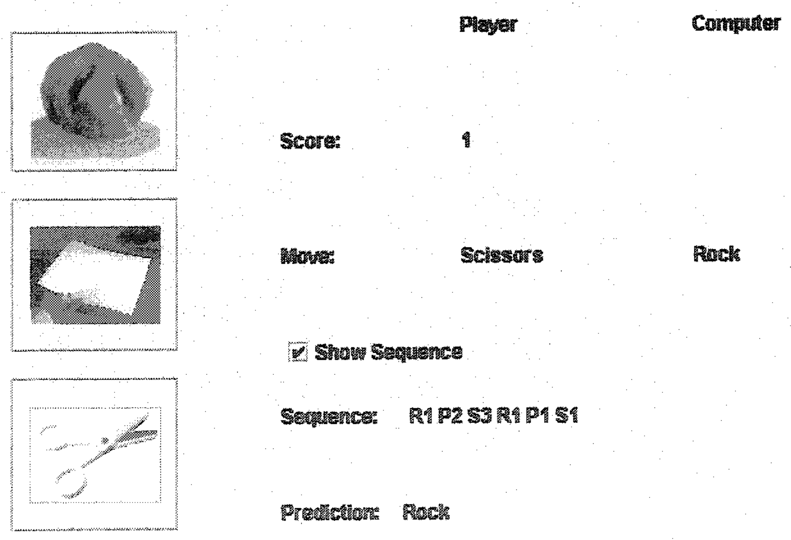


Figure 1. Basic concept for sequential prediction.

There are various algorithms to perform the string matching. An efficient, O(n) running time algorithm is presented in [Mommersteeg 2002], the details of which will not be discussed here. This algorithm was implemented in a simple Rock Paper Scissors (RPS) game (Figure 2).



	Player	Computer
Score:	1	
Move:	Scissors	Rock

☑ Show Sequence

Sequence: R1 P2 S3 R1 P1 S1

Prediction: Rock

Figure 2. RPS game applet using sequential prediction.

Furthermore, a small survey was conducted to see the effectiveness of sequential prediction (Table 1). Note that the number of rounds where the player and computer tied was not recorded.

Table 1. Effectiveness survey for RPS game with sequential prediction.

Player	Player's Score	Computer's Score	# Rounds	% Computer Won
A	7	14	21	66.67
B	15	10	25	40
C	9	8	17	47.06
D	14	14	29	48.28
E	11	9	20	45
Total	56	55	112	49.1

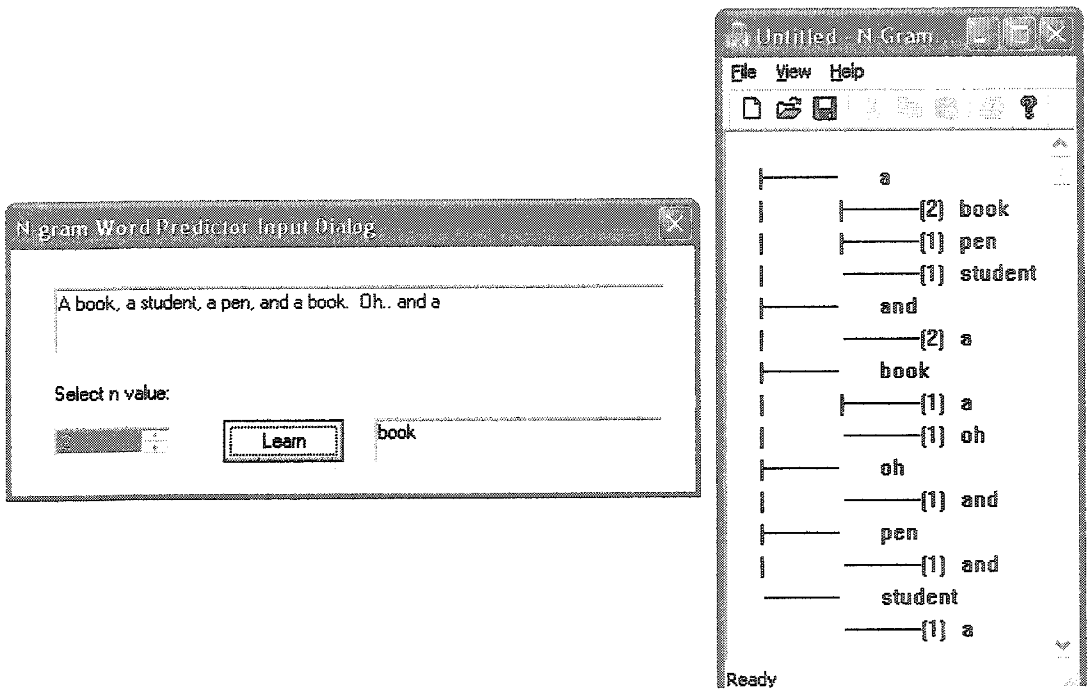
The results show that with sequential prediction, the computer won 49.1% of the time on average. If the AI simply picked random moves, the expected probability of the computer winning would be 33.3%. This is because for any given independent round, there is one winning move out of three possible moves. The 15.8% difference indicated sequential prediction to be at least somewhat effective in making its predictions.

2.2 N-Grams

N-grams provide an extremely simple and scalable statistical model. It is based on the Markov assumption, that any particular state is only dependent on a finite number of previous states. Such a process is called a Markov chain or Markov process [Russell and Norvig 2003]. The exact number of dependent previous states is determined by the order of the Markov chain. For example, in a first order Markov chain, a state depends only on the state immediately preceding it. The n-gram model assumes a Markov process, where given the order of the Markov process, i, the order of the n-gram, $n = i + 1$. The terms unigram, bigram, and trigram refer to n-grams with n equal to 1, 2, and 3 respectively.

One way to implement an n-gram would be to keep count of how many times each particular sequence of events has occurred. This is easily done by keeping a log of size $n - 1$ of previous events, $e_{i-n+1}, \ldots, e_{i-1}$, where e_i is the next event. A count is kept for every possible sequence, that is, all permutations of size n. Various data structures such as a matrix or a tree could be used for this. Now, given a sequence of size n, $e_{i-n+1}, \ldots, e_{i-1}, e_i$, the corresponding count can be retrieved.

To train the n-gram when a new event occurs, one increments the count corresponding to the sequence of events, $e_{i-n+1}, \ldots, e_{i-1}, e_i$, which are the events currently in the log concatenated with the new event. Likewise, the statistical probability that an event e_i occurs is also relatively simple to calculate. First, find the corresponding count, c, to the sequence formed by concatenating the logged events with e_i, thus forming $e_{i-n+1}, \ldots, e_{i-1}, e_i$. Note that this is basically the same sequence for training the n-gram, except in this case e_i is not a new event that has just occurred, but rather the event one wishes to find the probability of. It then follows that $P(E_i = e_i \mid E_{i-1} = e_{i-1}, \ldots , E_{i-n+1} = e_{i-n+1}) = c / (\text{sum of all counts})$. A word prediction program was created to experiment with n-grams (Figure 3).



A book, a student, a pen, and a book. Oh.. and a

Select n value: 2

book

```
├──────   a
│         ├──────[2]  book
│         ├──────[1]  pen
│          ──────[1]  student
├──────   and
│          ──────[2]  a
├──────   book
│         ├──────[1]  a
│          ──────[1]  oh
├──────   oh
│          ──────[1]  and
├──────   pen
│          ──────[1]  and
 ──────   student
           ──────[1]  a
```

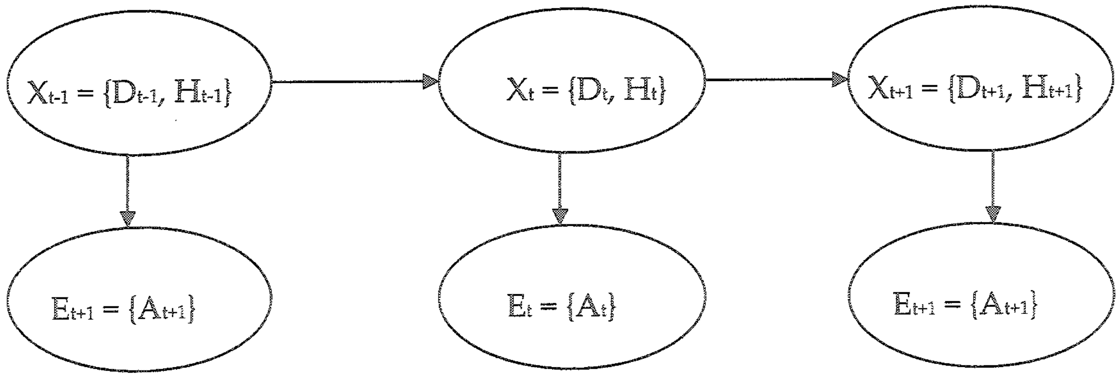
Ready

Figure 3. N-gram Word Predictor input dialog. (Left) Input dialog with training text. (Right) Display of trained bigram model with counts in parenthesis.

2.3 Hidden Markov Models

A temporal model is one in which the variables are updated over time. A hidden Markov model (HMM) is a temporal probabilistic model with a set of hidden state variables, X, a set of observable evidence variables, E, a transition model, and a sensor model. In an HMM, the state variables are all combined to form an aggregate mega-state [Russell and Norvig 2003]. Thus, technically an HMM only has a single state variable. E is dependent on X according to the sensor model, which simply defines the conditional probabilities of the form $P(E_t \mid X_t)$, where t is the current time-step. Likewise, the transition model defines how the state of one time-step is dependent on the state from previous time-steps. These are conditional probabilities of the form $P(X_t \mid X_{t-1}, \ldots, X_{t-i})$. The transition model implies that an HMM is a Markov chain of order i. Note that i was used to avoid confusion with the n for n-grams, since $n = i + 1$.

As an example, the relevant states in a fighting game might include the distance between fighters, D, and health of the NPC, H. The evidence variables might simply be the action of the player, A. Therefore, $X = \{D, H\}$ and $E = \{A\}$. This HMM could then be represented as in Figure 4, where the arrows indicate dependency.



$$\begin{tikzcd}
X_{t-1} = \{D_{t-1}, H_{t-1}\} \arrow[r] \arrow[d] & X_t = \{D_t, H_t\} \arrow[r] \arrow[d] & X_{t+1} = \{D_{t+1}, H_{t+1}\} \arrow[d] \\
E_{t+1} = \{A_{t+1}\} & E_t = \{A_t\} & E_{t+1} = \{A_{t+1}\}
\end{tikzcd}$$

Figure 4. Example hidden Markov model.

It is worth noting where the "hidden" part of the name in hidden Markov model comes from. The set X are states that are typically not observable. A feature of the HMM is that there are inference techniques that work with hidden states as part of the model. For example, one can find the most likely path through the HMM given an observation of the evidence variables, E. In the *Alpha Fighter* AI system, HMMs are only used to find the most likely state of the evidence variables, E, given the state of the input variables, X. In particular, E is the prediction of the AI system, for example, on what action the player will do next, and X contains relevant states of the world, and perhaps the history of previous actions the player has done. Using an HMM in this fashion, the values of X are not truly hidden, as they are simply states of the world that the AI system can directly observe. However, it is still technically a hidden Markov model.

2.4 Dynamic Bayesian Networks

A dynamic Bayesian network (DBN) is a generalization of an HMM [Russell and Norvig 2003]. Whereas an HMM has a single aggregate mega-state, a DBN maintains each individual state as separate. The purpose is so the existence of a dependency relationship between every pair of states can be independently specified. This is not possible for an HMM, which requires a transition probability between every possible mega-state value; in other words, every combination of every state. This is making an implicit assumption that every state is dependent on every other state. For the DBN, reducing the number of dependencies in the model would reduce the size of the transition and sensor models. The drawback for using a DBN is the added complexity in determining where all the dependencies lie, in other words, which states are connected. Taking the HMM example from Section 2.3, the corresponding DBN model might look like Figure 5, assuming that the health of the NPC, H, does not depend on the distance between the fighters, D.

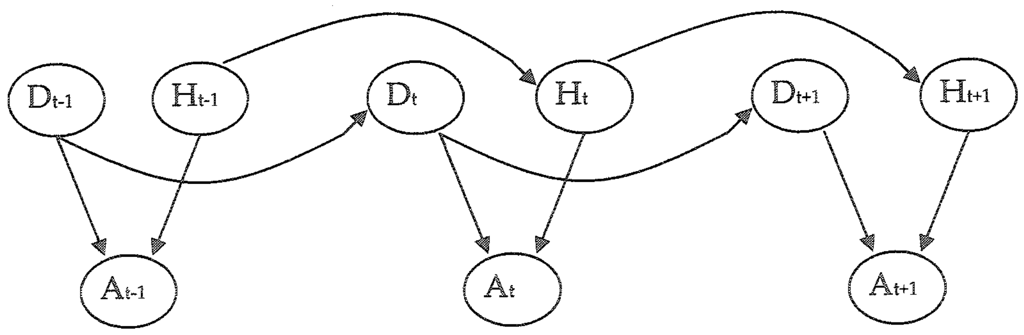


Figure 5. Example dynamic Bayesian network.

3. REQUIREMENTS

A complete computer game requires many components for it to be polished and presentable (Figure 6). Major modules include the game logic, graphics, physics, AI, controls, and sound. In addition, visual and audio resources are required. As AI is the main topic of this report, the requirements for the game AI will be discussed in more detail, while the other components will only briefly be mentioned.

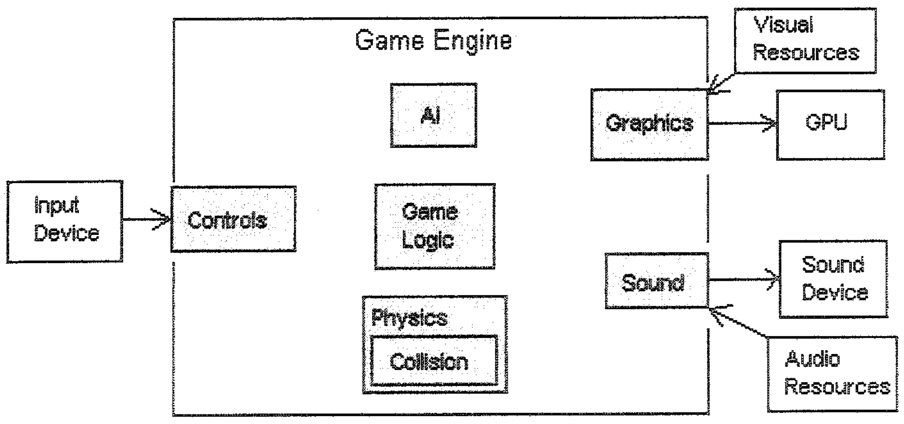


Figure 6. Major components of a game.

3.1 Requirements for Adaptive Game AI

The AI for a fighting game is inherently complex. Compared to action games where there are usually many enemies, in a fighting game, all the focus of the player is on the single NPC fighter. Therefore, the player will undoubtedly notice if the NPC behaves in an unintelligent way, such as if it were to repeat the same mistake many times. In addition, the NPC has two major concerns, each of which can be broken down into a multitude of smaller concerns. The two major concerns for the NPC are how to behave offensively, and how to behave defensively. For offense, the NPC must not only know which attacks to perform, but also how to position itself for these attacks. For defense, the NPC must know which defensive techniques, such as blocking and dodging, are effective in any given situation. It must also have ways to counter the player's offense. On top of everything, timing must be taken into consideration. For example, blocking too early could tip the player off as to the actions of the NPC, while blocking too late could be ineffective against a quick attack.

The level of complexity necessary for a good fighting game AI system must be taken into careful consideration in its design. The following subsections will detail other specific requirements of the AI system.

3.1.1 Flexibility

At this point, it is worth reiterating the goal of this project, which is to build an AI system for *Alpha Fighter* that will adapt to the player's changing fighting style. A typical fighting game has a very fast pace, compared to say, a role-playing game or an adventure game. Each round will usually last no longer than a few minutes. Furthermore, the player typical changes fighting techniques as the round progresses. For example, when the player's health is low, the player may become more defensive. On the other hand, the player may become more aggressive when the NPC's health is low. Due to these factors, the AI must be flexible enough to adapt quickly. If the NPC is slow to adapt, then it will never be able to keep up with the player's changing fighting style. A fast paced game requires a flexible AI.

3.1.2 Robustness and Scalability

In addition to flexibility, an AI system must also be robust. A good player will always try to find new strategies that work well. If there are strategies where the NPC cannot adapt, then the player will exploit that vulnerability, and consider it a glitch. Over time, this exploit will cause the game to become boring to the player. Therefore, an AI system must be robust so that it is able to adapt to any possible strategy the player may develop.

As a consequence of this, an AI system should also be scalable. Suppose a particular exploit was found during the development of the game where the player can always defeat the NPC by performing a particular sequence of actions. In order to correct this, a new strategy to counter this exploit must be integrated into the AI system. This is assuming this strategy could not already be learned by the AI system as is, since otherwise the exploit would not exist. Therefore, the strategy must be created manually and integrated into the system. This process of importing a new strategy should be made simple, and preferably require no code modification. Thus the AI system needs to be scalable.

3.1.3 Efficiency

Time efficiency is always a big concern in games. A video game must run in real-time. White it depends on the particular player, typically a frame rate below roughly 60 frames per second (fps) will be noticeable to the player. This is often referred to as lag. Sustained low frame rates below roughly 30 fps will cause enough frustration to the player for the game to become unplayable. Each frame, the game must update the state of the world. This includes performing physics calculations, collision detection, game logic, processing controls, as well as processing the AI. This is only half the work, as the frame must then also be rendered. It is worth noting however that some of the rendering work can be offloaded to the Graphics Processing Unit (GPU). Despite that, performing these computations at 60 fps is a daunting task, and requires much attention to efficiency. Therefore, training and using the AI system must be a time efficient process.

It is worth mentioning that video games running on consoles also require attention to space efficiency due to the limited amounts of memory on typical video game consoles. However, space efficiency can be mostly ignored for *Alpha Fighter* since its only target platform is the PC.

3.2 Game Engine

In addition to requirements for the AI system, the game engine's various other components have requirements of their own. Needless to say, the game engine must also be efficient, the reasons for which were explained in Section 3.1.3. Sections 3.2.1 through 3.2.3 detail particular requirements for the various modules of the game engine.

3.2.1 Physics and Collisions

Alpha Fighter does not require a sophisticated physics engine. In fact, it only requires very limited physics, just enough as to provide for the simple movements of the characters, such as walking and jumping. In general, a fighting game could involve much more complicated physics depending on its features. For example, if rag doll animations were desired, there would be need for inverse kinematics. Rag doll animations are physics based, as opposed to being predefined, and moves a character according to the effects of external forces and the constraints on the moveable joints of the character. In the context of a fighting game, a rag doll animation may be used by a fighter on the receiving end of an attack. However, for the purposes of *Alpha Fighter*, the features are limited and a complex physics engine is beyond the scope of the project.

An important part of a game's physics module is collision detection and reaction. For *Alpha Fighter*, collision detection needs to be extensible and flexible, but does not necessarily need to be extremely accurate. Since the character models for the fighters can become quite complicated in terms of their geometry, the collision mechanism should be flexible enough to handle various shapes and sizes of fighters. Furthermore, if the game were ever extended to require collisions between other types of objects, or even if an oddly shaped fighter were introduced, the collision system should be extensible enough to allow easy addition of new collision object types. The collision detection mechanism needs to be accurate enough to detect roughly where a collision occurred, but does not need to pinpoint the exact point of contact. Again, this would be necessary for a system using rag doll animations, perhaps as part of the collision reaction mechanism. For *Alpha Fighter*, it is sufficient to just be able to detect that a collision occurred and the rough vicinity of the collision point. For example, it is sufficient to know that the player's right hand hit the head of the NPC, without needing the exact coordinates.

3.2.2 User Interface

In addition to the above, *Alpha Fighter* must also have responsive and intuitive controls. It should have adequate sounds and music to help immerse the player, as well as provide feedback, such as when a fighter gets hit. Furthermore, it must provide an easy to read heads-up display (HUD) for both the fighters' health levels. This is a visual feedback to indicate the progress of the player.

3.2.3 Camera System

The camera system for *Alpha Fighter* must be such that it is automatic, and provides the player with a good view of the action at all times. This means both of the fighters must always be on the screen, and preferably oriented in such a way as to provide an intuitive feel for the player's controls. From the player's point of view, it should be clear how the movement controls will translate to the fighter's movements. It should also be clearly observable to the player what actions both fighters are doing at any particular time. Lastly, the camera should try to bring the player as close to the action as possible, without disrupting the other requirements. In other words, the camera should zoom in on the action to the point where the player can still clearly make out what is happening. This way, the player will have a more detailed view of the action and be more immersed in the action than if the camera was further away.

3.3 Visual and Audio Resources

Since *Alpha Fighter* is entirely focused on two fighters during any single round, the characters should be visually appealing. This includes the character model, textures, and the animations. In particular, the animations should look smooth and natural. Furthermore, the sounds used must clearly identify with the player as giving the feedback desired. For example, a sound used to indicate a successful hit should clearly carry that meaning for the player.

4. DESIGN

Figure 6 showed how a game can be divided into various modules. The discussion in this section will focus on the design of the AI module. The design of the collision mechanism will also briefly be discussed. As for the rest of the game engine, the reader is referred to the UML class diagram in Appendix A, and the *Alpha Fighter* source code on the enclosed CD.

4.1 Design Considerations for the AI

An adaptive AI system can be roughly divided into three parts, the model used to store the knowledge, the learning mechanism, and the decision maker (Figure 7).

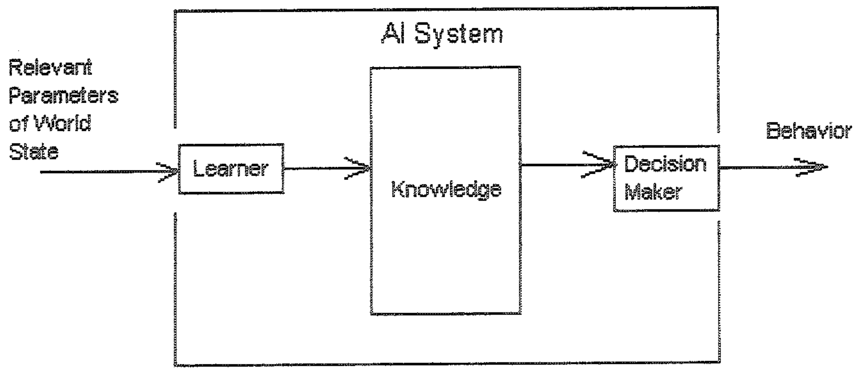


Figure 7. Three major components of an adaptive AI System.

The first part is how the knowledge itself is represented, so as to provide for the requirements discussed in Section 3.1. The second part deals with how the underlying knowledge is modified over time. The third part uses the stored knowledge to make informed decisions on what actions to perform; the behavior of the NPC. The following three subsections describe the considerations that went into choosing the appropriate techniques for each of these three components.

4.1.1 A Case for Direct Adaptation and Reinforcement Learning

Manslow explained the difference between direct and indirect adaptation in machine learning systems [Manslow 2002]. In learning with indirect adaptation, the AI system basically acts as a function to map various states of the world to various predefined behaviors. In this case, not only are all of the NPC's behaviors predefined, but the decision as to which behavior to use in any given situation is also predefined. Indirect adaptation requires much more work up front, since the designer must incorporate all the behaviors and decisions into the system. A benefit of indirect adaptation is that it is easy to control, since all behaviors are well-defined and known.

Direct adaptation on the other hand, is when particular parameters of the world state are fed into the AI system, and the system will change its behavior so as to reach favorable values for these parameters. For example, relevant parameters for *Alpha Fighter* might include the player's health and the NPC's health. The AI system can then try to adjust its behavior in such a way that the NPC's health is maximized, while the player's health is minimized. If used correctly, direct adaptation can provide a wider range of behavior than indirect adaptation. Furthermore, these behaviors do not need to be predefined. They are discovered by the AI when exploring and modifying its behavior. The disadvantage of direct adaptation is that it is harder to control, as the behaviors are not manually defined and not necessarily restricted to a known set of possibilities. Despite this, *Alpha Fighter* uses direct adaptation and overcomes some of the potential problems associated with it, which are mentioned below.

Two ways for an AI system to learn are by optimization and reinforcement [Manslow 2002]. If the effects of the NPC's behavior on the parameters are well known, then any optimization technique can be used to find the optimal behavior. Since this is not the case for *Alpha Fighter*, reinforcement learning is used, which does not require knowledge of how various behaviors affect the parameters in advance. The basic idea behind reinforcement learning is to reward or punish the behavior according to its consequence, in other words, whether the parameters were changed favorably or unfavorably. Then in the future, that behavior will be more likely to occur if it was rewarded, and less likely to occur if it was punished. Reinforcement learning is learning as it goes, whereas optimization tries to learn internally, before it commits to any actions.

One concern of reinforcement learning is that it can be slow to adapt [Manslow 2002]. This would go against the requirement that our AI system must be flexible. However, the speed to which the AI will adapt using reinforcement learning is dependent on how much and how often rewards and punishments are given. By simply increasing the amount of reinforcement, the AI can be made to learn arbitrarily fast. The tradeoff is that the faster the AI can adapt to something new, the faster it will unlearn something old. This parameter of how fast the AI behavior should change is one that must be tweaked in practice to best fit the game.

Another concern with reinforcement learning is that a good set of parameters must be chosen. By including parameters that are not relevant for the NPC to adapt, the AI will suffer from the problem of overfitting. Overfitting is when the AI correlates a behavior with a set of parameters that are too specific. This makes the AI unable to generalize about what the relevant parameters of this situation are which makes the behavior good. For example, an irrelevant parameter in the case of *Alpha Fighter* might be the particular stage the fighters are fighting in. Clearly, if the only difference between various stages is the look, then the stage should have no affect on the NPC's behavior. However, if the AI were to take this parameter into account, it may form the belief that a particular behavior is only good for a particular stage. From the perspective of the AI, the stage is simply another parameter; it cannot discern its importance. When the fight occurs at a different stage, the AI will not be able to generalize the knowledge learned during the old stage to the new stage. This will severely limit the usefulness of the AI's knowledge, since it is too specific. On the other hand, if too few parameters are used such that some relevant parameters are excluded, the AI will generalize too much. This will cause the AI to believe a particular behavior is good for a large, general group of situations, while in reality the behavior may only be good for a subset of that group. Therefore, reinforcement learning requires a very careful selection of the set of relevant parameters.

4.1.2 Choosing the AI Model

Taking the requirements laid out in Section 3.1 into consideration and the various AI models described in Section 2, it was decided to use a hybrid of n-grams and hidden Markov models for the *Alpha Fighter* AI system. Sections 4.1.2.1 through 4.1.2.5 explain the various factors that influenced the decision.

4.1.2.1 Sequential Prediction

Despite the effectiveness of sequential prediction, it was not suitable for the *Alpha Fighter* AI system. To understand the reason, the difference between deterministic and probabilistic methods for making predictions, as explained in [Laramée 2002], must be distinguished. In deterministic prediction, the most probable event is always the choice. On the other hand, in probabilistic prediction, an event is chosen at random, but weighed by the probability of that event. In this case, although the most probable event has the most likelihood of being chosen, every other possible event still has a chance, albeit a smaller chance.

The problem with sequential prediction is that it assumes the use of deterministic prediction in order to gain in efficiency. The iterative nature of the algorithm allows for the most probable next event to be quickly determined. As long as only the most probable event is desired, sequential prediction works effectively. However, *Alpha Fighter* requires probabilistic prediction in order for the NPC to explore other possibly favorable behaviors, as well as for the NPC to behave more naturally. To elaborate, if deterministic prediction were used, the NPC would lock itself to repetitively performing the same action, as long as that action is positively reinforced, because it will continue to be the most probable.

Although this may lead the NPC to victory, there are potentially better options that the NPC is not exploring. Furthermore, the repetitive nature of the NPC's actions would seem unintelligent to the player. By providing some randomness to the NPC's behavior, the player will regard the NPC as less predictable, which typically leads to a perception of higher intelligence. Due to this line of reasoning, sequential prediction, which does not lend itself to probabilistic prediction, was ruled out for the *Alpha Fighter* AI.

4.1.2.2 N-grams

Although n-grams do lend themselves to both deterministic and probabilistic predictions, they lack the requirements needed for robustness. In particular, when the value of n is high, the n-gram becomes too specific and overfitting can occur. On the other hand, with low n values, the n-gram becomes too general. This is analogous to the problem discussed earlier concerning how the set of parameters chosen can affect the effectiveness of reinforcement learning. For example, assume that a trigram model was being used. As long as the trained model contains the sequence of the previous two events, the trigram will be able to give probabilities for each event that can occur next. However, if the training never involved this particular sequence of two events, then the trigram will have no knowledge as to the probabilities for the next event. Therefore, a high value of n, in this case a trigram, can lead to a complete failure in the predictive capability of the AI.

This problem of overfitting can be somewhat avoided by using smaller values for n. However, in using a smaller value, the accuracy of the higher order n-gram is lost. In the extreme case, when a unigram is used, the model simply makes its prediction based on the frequency with each event has occurred, without regard to the recent history. The more times an event occurs, the more probable it is in a unigram, with no regards to the context of the occurrence.

The lack of robustness in the n-gram model makes it seemingly unsuitable for use in *Alpha Fighter*'s AI system. However, as shall be seen in the next section, this problem can be overcome by extending the n-gram model.

4.1.2.3 Hidden Markov Models

Hidden Markov models can be used to overcome the lack of robustness in n-grams. The idea is that while a single n-gram model will always suffer from either being too specific or too general. A set of multiple n-grams can be used to overcome this problem. For example, suppose again that a trigram was being used. If and when the trigram fails to make a prediction because of the overfitting problem, a bigram can be used instead to make the prediction. While it is less likely that the bigram will also fail to make a prediction, in the case that it too fails, a unigram can be used. In the case of a unigram, the only time when it would fail, is when the model has not been trained at all. In other words, the model has absolutely no knowledge. In this extremely rare case, which would only happen at the beginning, the AI can simply pick a random event to be the prediction.

This idea of using multiple n-grams can be accomplished by incorporating the n-grams into a hidden Markov model [Charniak 1996].

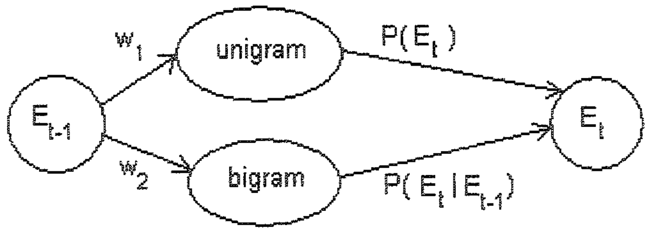


Figure 8. Combining multiple levels of n-grams in an HMM.

Figure 8 gives an example of how a unigram and bigram can be incorporated into an HMM. In such an HMM, each n-gram is given a weight, which represents the value of its prediction. The higher the n value, the higher the weight should be, since typically higher order n-grams are more accurate. However, each n-gram in the HMM should have a weight w_n, such that $0 < w_n < 1$, and n is the n-value of the n-gram. The summation of all the weights should equal 1: $\sum w_n = 1$. The probabilities of any particular event occurring according to each n-gram should also be such that $0 \leq P_n(E = e) \leq 1$, where P_n is the probability using the n-gram of order n. To calculate the absolute probability of a particular event occurring, simply take the summation of the products of w_n and P_n over all values of n: $P(E = e) = \sum w_n P_n(E = e)$. The higher order n-grams will naturally have more influence on the probability, but the predictive model is robust in that even when the higher order n-grams fail to make a prediction, thus giving a probability of 0, lower order n-grams can still exert some influence on the final probability.

This model of incorporating multiple n-grams into an HMM satisfies all of the requirements discussed earlier for the AI. In particular, it is robust, flexible, and efficient. Therefore, it was the model chosen for use in the *Alpha Fighter* AI system.

4.1.2.4 Dynamic Bayesian Networks

Recall from Section 2.4 that an HMM is a specific type of a DBN, and that a DBN allows for more efficiency at the cost of more work in determining dependency relationships. Although efficiency is a concern, the number of parameters used in the *Alpha Fighter* statistical models is small enough that the efficiency of using an HMM is not a concern. Using a DBN in this situation was not necessary and would only have complicated matters.

4.1.2.5 Minor Considerations

Two other AI techniques were also considered briefly, but quickly dismissed. However, they are prominent techniques and certainly worth discussing. Artificial neural networks (ANN) and genetic algorithms (GA) are two fairly new AI techniques, at least in their use in games. They were quickly found to be not suitable for *Alpha Fighter*, where the rate of adaptation needed to be much faster than either ANNs or GAs could provide. Neural Networks are useful when dependency relationships are unknown beforehand [Kaukoranta *et al.* 2004]. The ANN simply goes through a training period to adjust its weights so as to fit the structure of the good inputs. This training period typically takes a long time, and would not meet the flexible, quick adapting requirement needed for *Alpha Fighter*. The decision is further reaffirmed as Manslow discusses the tradeoffs between using reinforcement learning with lookup tables and neural networks in [Manslow 2004]. Manslow notes that while neural networks can better generalize, they are slow to adapt. On the other hand, using lookup tables, which is basically what our HMM is, allows for a quick learning rate. Without going into the details, GAs also tend to evolve very slowly. It is more suitable for offline learning, not the fast paced, in-game learning required for *Alpha Fighter*.

4.1.3 Hierarchical Decision Making

The decision making part of the AI can be separated into three levels: strategic, tactical, and operational [Kaukoranta *et al.* 2004]. As noted in Section 3.1, the AI of *Alpha Fighter* is complex by nature. Breaking the decision making process into three layers of abstraction would make the AI easier to understand and implement. Each layer is only concerned with the goals at its scope, without having to worry about the goals of the other layers. Furthermore, each layer can be adjusted separately. For example, one layer could be made adaptive, while another layer non-adaptive. For these reasons, hierarchical decision making was used for the AI system of *Alpha Fighter*, both to deal with its complexity, and to make it extensible.

4.2 The *Alpha Fighter* AI System

Having discussed the various considerations taken into account, and stated the components and techniques chosen for the *Alpha Fighter* AI system, the following subsections present the design of the chosen AI model and techniques used specific to *Alpha Fighter*.

4.2.1 The Four Statistical Models

Section 4.1.2 considered various statistical models that could be used for *Alpha Fighter* and it was concluded that an HMM with embedded n-grams would be used. There are four areas in *Alpha Fighter* where such a model is deployed. First, as pointed out in [Laramée 2002], "… [in] fighting games, players develop signature techniques and move sequences." Therefore, one HMM is used to predict the player's attack patterns, and is called the "player attack model." The other three HMMs are used to keep statistics on the effectiveness of the NPC's behavior. The first and second of these HMMs are at the tactical level, and keeps track of which tactics are effective in which situations. There are two types of tactics, regular and counter, and each has its own dedicated HMM. The difference between the two types is explained in Section 4.2.3.1. These two HMMs are called the "regular tactics model" and "counter tactics model" respectively. The final HMM is at the operational level, and keeps track of which attacks are effective in which situations. This model is called the "NPC attack model."

4.2.2 Training the Four Models

There are two ways to use the acquired knowledge in an AI model, prediction and production [Laramée 2002]. The player attack model is an example of where prediction is used. The AI is concerned with finding out which attack the player likely to perform next in a particular situation. The other three models are used for production, since they define the behavior of the NPC. Due to the difference between the player attack model and the other three models, there are two methods of training, one for each group.

For the player attack model, each time the player finishes an attack, the type of attack is placed in the player attack log. This log is of size n, n being the order of the highest n-gram used. When a new item is added to this log, the AI will update its player attack model by incrementing the count of that particular attack for each of the n-grams being used. For example, suppose the high n-gram being used was a trigram and the player attack log contains the attacks: punch, kick, punch, where the first element is the most recent and just added. The AI would train the player attack model by updating the following counts, and indirectly, the probabilities: P(punch), P(punch | kick), and P(punch | kick, punch), for the underlying unigram, bigram, and trigram models respectively.

The other three models are trained using reinforcement learning. In reinforcement learning, it is very important to provide reinforcement at the right time [Manslow 2004]. The regular and counter tactics models should be reinforced whenever a fighter is damaged. If the player is damaged, the behavior of the NPC is positively reinforced. If the NPC is damaged, then the behavior of the NPC is negatively reinforced. The amount of reinforcement is proportional to the amount of damage taken. A log of recent previous tactics executed by the NPC is kept. During reinforcement, the current tactic is reinforced by the highest amount, whether it is positive or negative. Tactics occurring before that are also reinforced, but each subsequent tactic is reinforced with a lesser and lesser amount, until either the amount is insignificant so as not to cause any reinforcement, or the tactics log is exhausted.

The NPC attack log uses the same idea, with the exception that it is reinforced during two additional events. The NPC attack log is not only reinforced when damage occurs, but also when an attack is unsuccessful. This makes the AI more likely to explore, and less likely to repeat the same attack for a long time. If the AI starts repeating its attacks and they are unsuccessful, it will eventually be negatively reinforced to the point where other attacks are more likely to be picked. However, the amount of reinforcement for a successful attack is greater than the amount of reinforcement for an unsuccessful attack. The reasoning behind this is that a successful attack causes damage, which is a clear indication of progress, in the case where the player is damaged, or setback, in the case where the NPC is damaged. On the other hand, although an unsuccessful attack by the NPC should be negatively reinforced, it is not as strong an indicator of setback as when damage is sustained.

4.2.3 The *Alpha Fighter* AI Hierarchy

4.2.3.1 The Strategic Level

At the strategic level, the NPC in *Alpha Fighter* has two major goals defined, hurting the opponent, and protecting itself. This level is relatively simple, since it only needs to decide which goal to work towards. This is the only layer of the AI in *Alpha Fighter* that is entirely non-adaptive. The fact that reinforcement learning can be used in a particular situation does not mean that it should be used [Manslow 2004]. The role of the strategic layer in *Alpha Fighter* is so simple that a predefined, reactionary behavior is more appropriate than trying to make it needlessly adaptive.

As mentioned briefly in Section 4.2.1, there are two types of tactics at the NPC's disposal, regular and counter. Normally, the NPC simply chooses a regular tactic, perform it to completion, and then chooses a new one. This corresponds with the goal of hurting the opponent. However, anytime the player initiates an attack, the AI system is sent a warning of a possible threat. It is important to note that the warning only tells the AI that an attack is being executed, but contains no details on what the attack may be. On receiving this warning, the AI will analyze the potential threat based on its prediction of what the threat may be, and determine whether the player's attack is truly a threat. For example, if the player is not within striking range, the AI can deem the threat as harmless. If the threat is found to be harmless, the AI simply ignores the warning and continues with its current regular tactic. On the other hand, if the threat is found to be serious, the AI will immediately replace its current tactic with a counter tactic (Figure 9), in other words, the AI will change its goal to protecting itself.

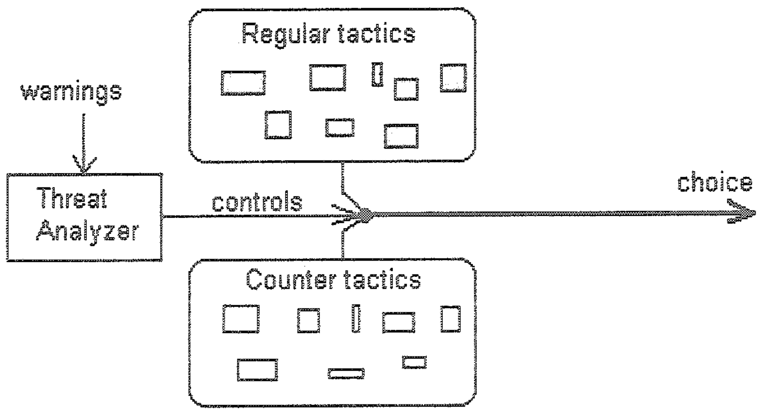


Figure 9. The strategic level.

Initially, it may seem unfair that the AI receives a warning of an incoming attack, instead of purely relying on its prediction models to determine when an attack is coming. However, upon closer examination, this is not unlike what a human player does. This behavior is actually trying to mimic the reactionary behavior of a human player at first sight of an incoming attack. A human player will typically be able to see an attack coming, but not necessarily know which attack it is. The player must then use good judgment to guess what the attack may be, and possibly perform a counteraction. The NPC is simply doing the same thing, but since it has no vision, it is simply fed the warning.

4.2.3.2 The Tactical Level - Tactics

The tactical level is where the AI chooses a specific tactic from the set determined at the strategic level, that is, either a regular tactic or counter tactic. As stated earlier, both types of tactics have their own corresponding statistical model, which keeps track of the effectiveness of each tactic in various states of the world. For both models, the distance between the fighters is taken into account. This parameter is relevant since a particular tactic may work at close range, but fail when the fighters are far apart. For the regular tactics model, the category of action the player is currently performing is also taken into consideration. An action category is one of the following: attacking, blocking, jumping, moving, hurting, and jump attacking. With the exception of the jumping category, these do not tell the AI exactly what the player is doing, but do give it a general idea. The action category of the player is relevant for deciding which regular tactic to perform, since a tactic may work for say, while the player is jumping, but not work in other situations where the player is standing. The counter tactics model does not use the action category, but actually goes one level more specific and uses the AI's prediction of the player's current action. Recall that a counter tactic is used only when a player attacks. Certainly, the effectiveness of a particular counter tactic depends on what the player's attack is. For example, a jump attack counter tactic would likely work well when the player does a low attack, but probably fail when the player does a high attack or jump attack.

Through the use of the regular and counter tactics models, the AI at the tactical level will adapt to the player as it learns which tactics work well in each particular situation. Figure 10 summarizes the ways a regular tactic and counter tactic are chosen.

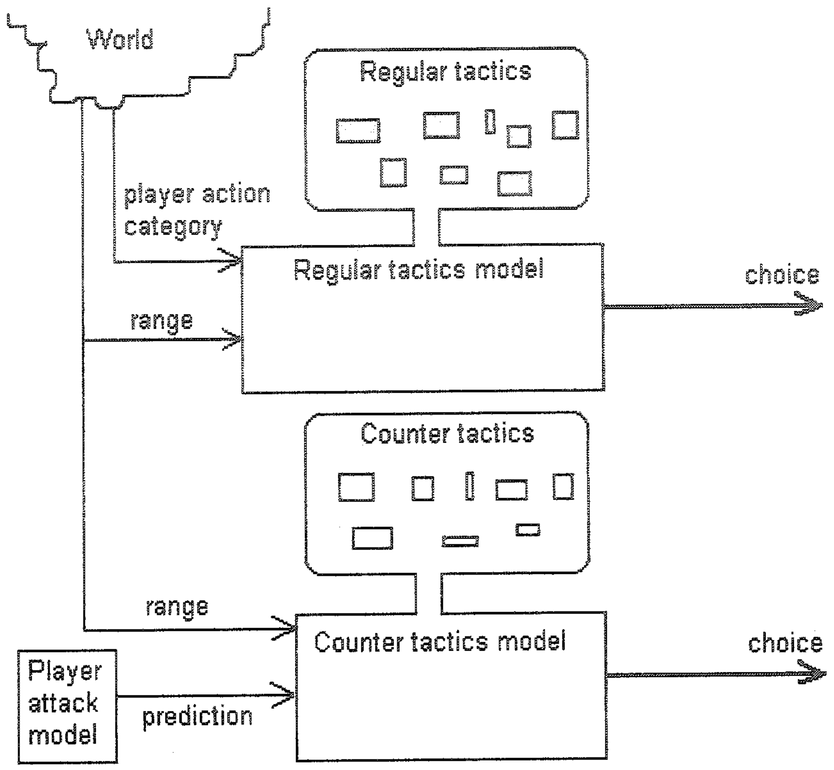


Figure 10. Choosing a tactic.

The tactics are defined in a script file to facilitate ease of modification and creation. This also allows the different fighters to easily associate themselves with a different set of tactics, by simply referring to the appropriate tactics file. The format of the tactics file is presented in Appendix E.

4.2.3.3 Operational Level - Steps

Each tactic is made up of a sequence of one or more steps. Table 2 summarizes all the various steps available.

Table 2. List of tactical steps.

Step	Description
MoveToRange	Move to a specified distance to the opponent.
MoveWithinRange	Move within a specified distance to the opponent.
Attack	Perform an attack.
Block	Perform a block.
Jump	Jump in the specified direction.
JumpAttack	Jump in the specified direction and attack after given time.
Wait	Wait for specified time.
SideStep	Move sideways for specified time.

The NPC simply executes the sequence of steps in its current tactic, starting from the beginning, and moving to the next step once the current step is done. Some of these steps such as jump and wait are predefined behaviors. Others have a small bit of logic built in, such as the move to range step, which makes the NPC walk forward or backward depending on its relative position to the player. The most interesting steps are attack, block, and jump attack. The block step simply tells the NPC to block, but does not specify to block high or low. It is only at execution time that the operational level of the AI uses the player attack model to make a prediction as to which attack the player is doing, if any. If a low attack is predicted, then a low block is performed, otherwise a high block is performed. This means the block step is an adaptive component of the AI at the operational level. The attack and jump attack steps are likewise adaptive. For each of these two steps, while the step indicates an attack should be performed, the type of attack is not specified. At execution time, the NPC attack model is consulted to find a probabilistic prediction of which attack is likely to succeed, and that attack is performed.

4.3 An Extensible Collision System

The various character models are fairly complicated in shape, and cannot be bounded by a primitive geometric volume such a sphere or a box. Using such bounding volumes would lead to grossly inaccurate collision detection as the volumes cannot fit tightly around the character. Although one approach would be to perform collision detection per polygon, this level of accuracy is simply not necessary and the time efficiency tradeoff makes it not worthwhile. The method employed by *Alpha Fighter* is to use multiple primitive geometric volumes, such that they conform fairly tightly around the character models. Since different character models have different shapes, these primitives are made easy to specify by taking a data driven approach. Each individual character has an associated fighter data file (see Appendix D), where its collision objects can be specified. Currently, the collision system supports spheres and capsules (Figure 11), but is set up in such a way to be easily extendible.

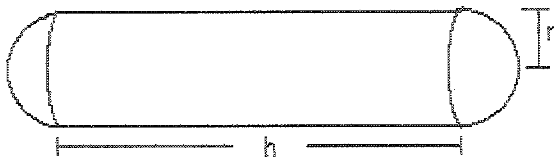


Figure 11. Collision capsule constructed of a cylinder and two semi-spheres.

The collision detection itself is fairly simple, and sacrifices some accuracy for speed. In particular, the methods used only detects whether two collision objects are overlapping each other. The main problem with only performing an overlapping test is that the collision objects could pass through each other between frames. This problem could occur if there are very small collision objects, or if the objects were moving at high speeds. Despite this problem, the overlapping collision detection method is sufficient for *Alpha Fighter*, where the problem stated is extremely unlikely to happen, except perhaps when the frame rate is extremely low. However in that case the game is unplayable anyway.

5. IMPLEMENTATION

The complete source code for *Alpha Fighter* can be found on the accompanying CD or on the website (See Appendix F). This section will describe some of the more notable implementation details of the AI system and the game in general.

5.1 Game States

At the highest level of the program is the CGame class, which contains a nonempty stack of CGameStates throughout the course of the game. The top of the game state stack holds the game state currently in control. The detail of how a game state is updated is presented in Section 5.3. Here, the control flow of a game through the various game states is presented.

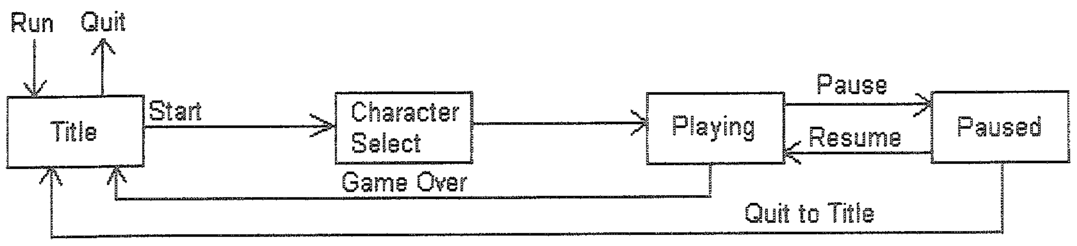


Figure 12. Game state transition diagram.

Figure 12 shows the possible transitions between the various game states. When a transition occurs, the new game state is placed on top of the stack, which implicitly marks it as the current game state. There is an option of whether to keep the old game state or discard it. In certain transitions, such as from the title screen to the character select screen, the old game state, title game state in this case, is discarded. This allows the program to free up any memory used by the old game state. However, in certain situations the old game state must be kept. For example, this is the case when transitioning from the playing state to the paused state. In this case, the playing state cannot be destroyed, since the player expects to be able to resume the game with everything intact. This is the reason that a game state stack is used, so that the game can transition from the paused state back to the playing state simply by popping the stack. It should be noted that when going from the paused state to the title state, not only must the paused state be discarded, but also the playing state that is further below the stack. In other words, the entire game state stack is cleared, and then the title state pushed.

5.2 Stage and Fighter Files

Two types of files were created to facilitate in the initialization of the stages and fighters. These files allow new stages and fighters to be added with ease. Basically, the stage file specifies the various textures to be used, the music for the stage, and the size of the fighting area. The fighter file contains the name of the fighter, the .x file of the model, a scale factor for the model, the collision objects specifications, and a list signifying which body parts each attack and block affects. For more detail, see Appendix C and D for the stage file and fighter file respectively.

5.2.1 The CFighter Class

The CFighter class represents a fighter. Although the player uses the CFighterPlayer subclass, and the NPC uses the CFighterNpc subclass, most of the functionality is in the base CFighter class. There are two things of particular importance for the fighter class. First, CFighter has a CModelSkinned as its visual object. A CModelSkinned is a skinned model, in other words, a skeletal mesh that provides key-framed animation functionality. Second, CFighter does not handle collisions itself, and in fact, has no collision object. Rather, it delegates that task to its set of CBodyParts. The exact list of body parts for a particular fighter is specified in the fighter data file in the collision object specification lines. Each body part corresponds to a particular bone in the fighter's skeleton, and its position and orientation are updated according to the skeletal animation of the fighter. See Appendix A for a UML class diagram depicting the relationships between CFighter, CModelSkinned, and CBodyPart, and their relations to the game as a whole.

5.2.2 Collision Detection

There are two types of collision objects in *Alpha Fighter*, spheres and capsules. Collision spheres are used to bound areas such as the head and fists, while a capsule is used to bound areas such as the arms and legs. A capsule was used as opposed to a cylinder because it is easier to test capsules for overlap than cylinders. This is due to the spherical caps in a capsule as opposed to the hard edges at the ends of a cylinder. When a collision is detected, a SCollisionInfo object is created and placed in the collision queue of each entity involved. For *Alpha Fighter*, the collision information was kept very simple and small, and contains only a pointer to the entity one collided with. SCollisionInfo can easily be extended to contain more information, such as the point and time of contact, for more sophisticated systems.

5.2.3 Collision Reaction

Recall that if a collision is detected, a SCollisionInfo object is placed in each of the colliding entities' collision queues. Later in the frame, each entity is given the chance to look at its collision queue and process any detected collisions as it sees fit. In *Alpha Fighter*, the only possible collision is between two body parts.

Each body part has an independent state: attacking, blocking, or neutral. In collision reaction, the body part checks to see if it has damaged the other colliding body part. Note that a body part does not check whether it has taken damage itself. Whether a body part has damaged the other colliding body part is determined by the states of the two body parts involved. Table 3 summarizes the consequences of every possibility.

Table 3. All possible collision scenarios and consequences.

B A	**Attacking**	**Blocking**	**Neutral**
Attacking	Both are damaged	No damage	A damages B
Blocking	No damage	No damage	No damage
Neutral	B damages A	No damage	No damage

The state of a body part is set when the fighter performs a new action. Each particular attack action keeps a list of the body parts it affects, in other words, the body parts that are doing the hitting. This list is specified in the fighter data file for every possible attack that fighter can do. When an attack action is executed, it simply sets all the body parts in its affected body parts list to the attacking state. In the same way, block actions signify which body parts are involved in the block. For the other actions that are neither an attack nor a block, all the body parts of the fighter are set to the neutral state. Figure 13 shows a screenshot with the debug collision object display turned on. Body parts with red collision objects are attacking, green is blocking, and yellow is neutral.

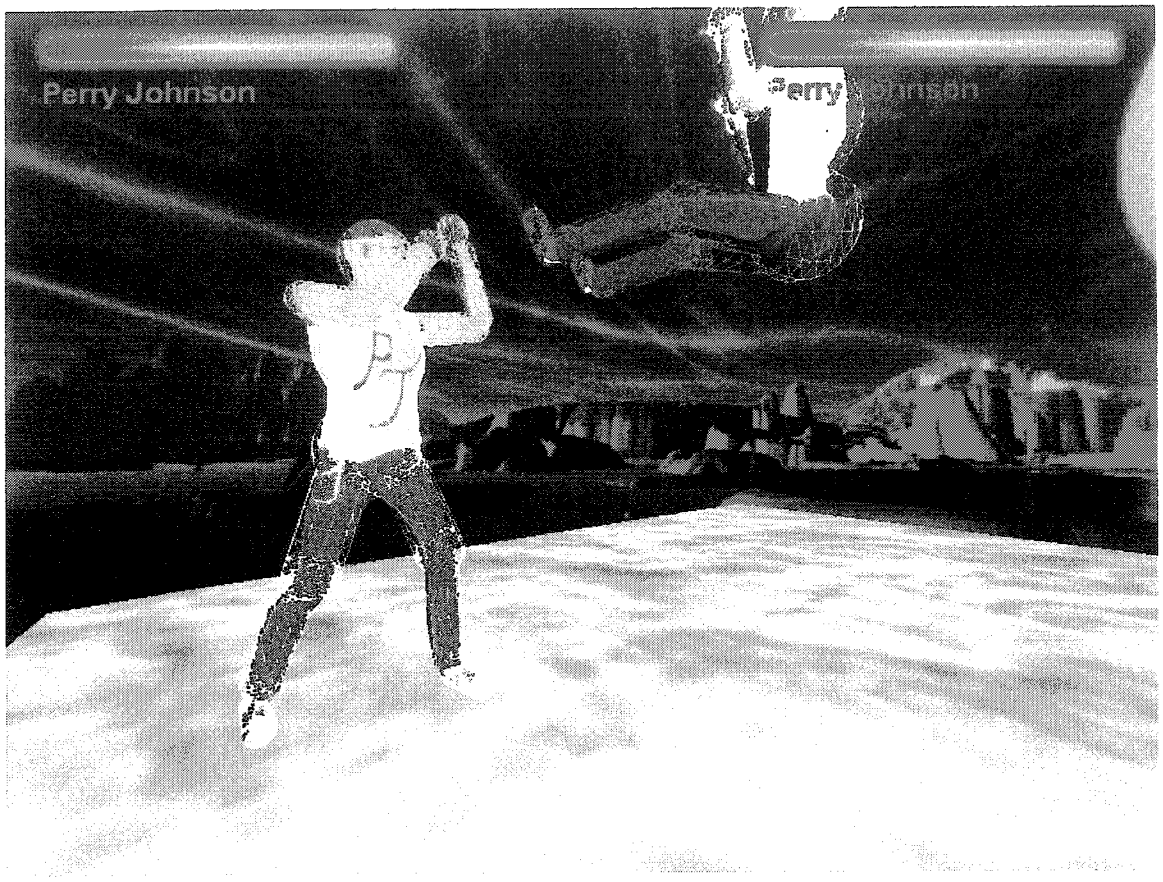


Figure 13. Collision object display showing body part states.

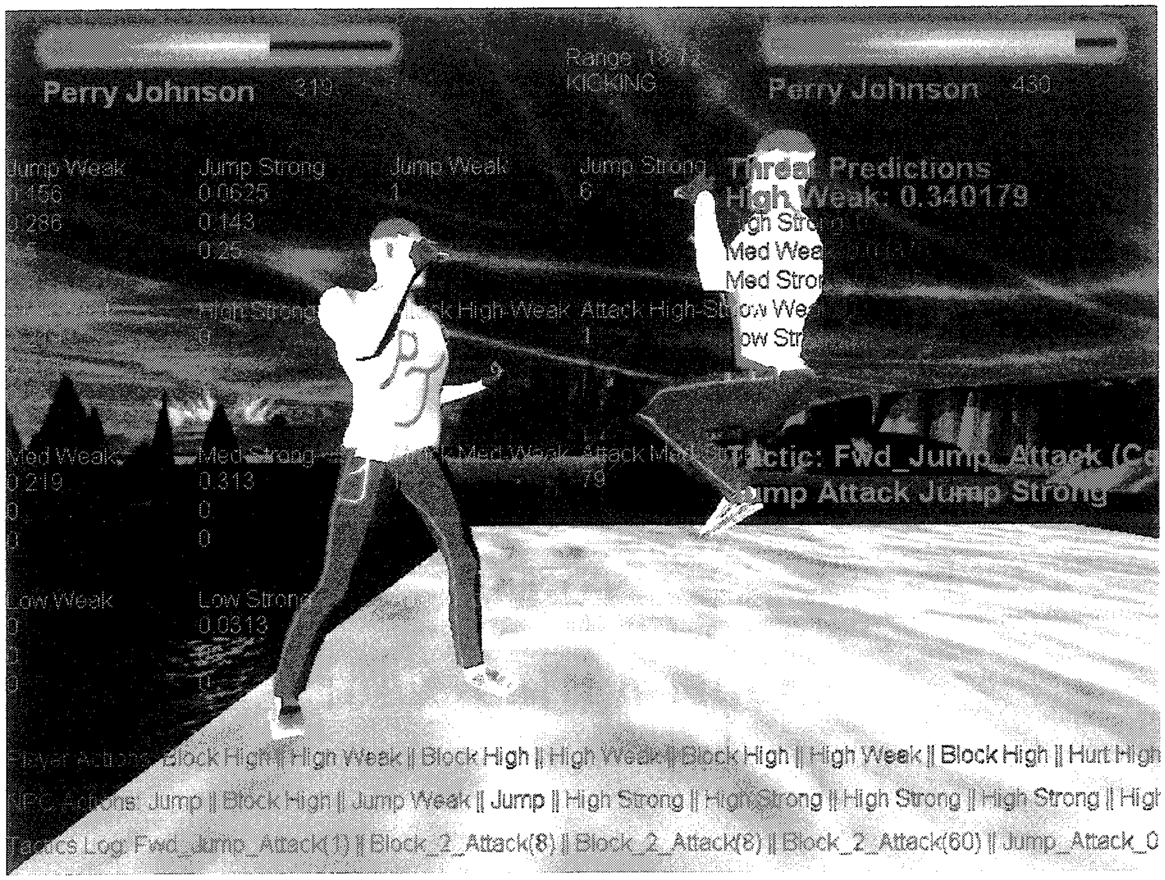
5.3 Processing a Frame

While the CGame class controls the flow of the entire game, the CGameState class is controlling the processing details for each frame. In each frame, the step method of the current game state is called by the game class, this in turn sets off a hierarchy of calls to update the state of the world and render the frame. The updating includes processing input, detecting collisions, updating all the entities, and moving all the entities. Appendix B shows a sequence diagram of the important high level calls made to process a frame during the playing game state.

One interesting note is that the camera is not updated until after all other entities have moved. This ordering is important since the camera follows the fighters, and if the camera were to update before the fighters moved each frame, the camera would fall behind. Another important point is that all the entities perform a specific step before moving onto the next step. One should contrast this approach to where an entity moves through all the steps before the next entity is processed. This latter approach, called the serial approach, can lead to problems because some entities may base its behavior on information from other entities. In the serial approach, the order in which the entities are processed will have a significant effect on the behavior of the entities that are dependent on others. Since this is not desirable, the parallel approach first described is used.

5.4 Data Structures for the AI Models

The following two subsections explain how the four AI models were implemented in *Alpha Fighter*. Figure 14 shows a screenshot with the AI debug display turned on.



```
Jump Weak     Jump Strong    Jump Weak          Jump Strong
0.456         0.0625         1                  6
0.286         0.143
              0.25

              High Strong    ck High-Weak       Attack High-St
              0                                 1

Med Weak      Med Strong     ck Med-Weak        Attack Med-St
0.219         0.313          1                  79
0             0
0             0

Low Weak      Low Strong
0             0.0313
0
0             0

Player Actions: Block High || High Weak || Block High || High Weak || Block High || High Weak || Block High || Hurt High
NPC Actions: Jump || Block High || Jump Weak || Jump || High Strong || High Strong || High Strong || High Strong || High
Tactics Log: Fwd_Jump_Attack(1) || Block_2_Attack(8) || Block_2_Attack(8) || Block_2_Attack(60) || Jump_Attack_0
```

Figure 14. In-game AI debug display.

Going from top to bottom and left to right, the numbers next to the fighters' names are their numeric health values. A range is displayed at the center top, showing that the fighters are currently within kicking distance. The two green columns on the left are the statistics for the player attack model. Below the name of each attack, the unigram, bigram, and trigram probabilities are displayed, given the current situation; in this case that the fighters are within kicking range. The next two green columns show the NPC attack model. The numbers there are the current point values for each attack in the current state. The meaning of these point values will be desribed in Section 5.4.2. In this particular scenario, the medium strong attack, with the highest point value of 79, has worked the best for the NPC, and is its most probable form of attack. The red display on the right indicates there is a threat, otherwise the display is green. It shows each of the possible attacks the player can do in the given situation, and the probability of each one. The most probable attack is in bold font, in this case it is a high weak attack. The blue display right below shows the tactic the NPC is currently executing, along with the steps involved. In this case, it is a counter tactic with a single step of jumping forward with an attack. Using the NPC attack model, the NPC decided to do a strong jumping attack, as opposed to a weak one. Finally, the bottom three rows display a log of the player's action history, the NPC's action history, and the NPC's tactics log along with the point values of each tactic.

This particular situation is interesting in that it shows the AI is using probabilistic production, as opposed to deterministic production. Note that the point value for a Block_2_Attack counter tactic was most recently at 8, while the Fwd_Jump_Attack counter tactic only has one point. In spite of this, the latter counter tactic was chosen. If a deterministic approach was taken, the former tactic would always be chosen in this scenario.

5.4.1 The HMM Tree Array

The player attack model uses an array of trees. The index of the array is the range between the two fighters, which is the only world state parameter taken into account for this model. There are three values for range: punching, kicking, and out. Punching means the fighters are within punching distance, kicking means they are within kicking distance, but not within punching distance, and out means they are not within striking distance. Recall that the HMM is actually a collection of n-grams with values of n from 1 through some positive integer MAX_N_GRAM. The tree is structured such that Level i of the tree represents the i^{th} order n-gram with the exception of Level 0, which is the root (Figure 15).

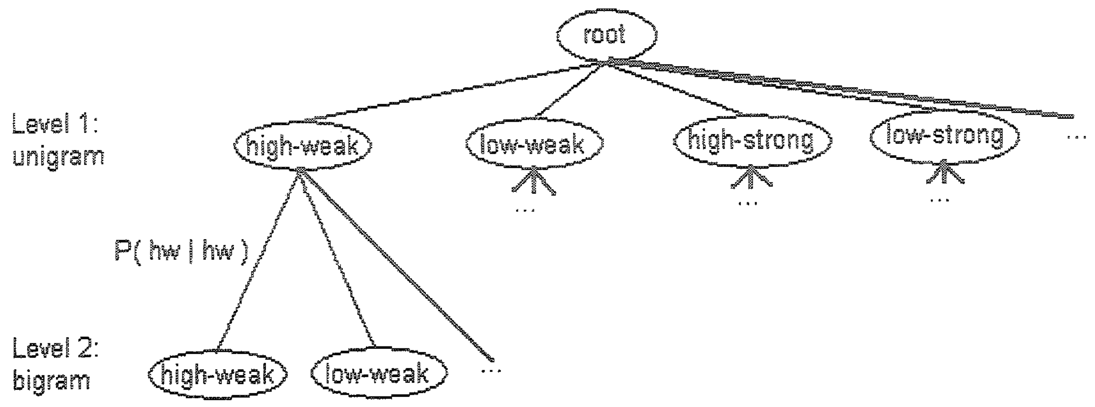


Figure 15. HMM tree structure.

Each node in the HMM tree is a CActionProbNode, and contains a count, pointers to its children, and the sum of its children's counts. The count of a node indicates the number of times that particular sequence of attacks has occurred. For example, the count of the low-weak node at Level 2 in Figure 15 is the number of times that a low-weak attack has occurred after a high-weak attack. Training the model simply involves traversing to the appropriate node, n_{train}, given the history of the player's attacks, and incrementing the count for n_{train}. In order to make a prediction, the probabilities of each possible next attack must be determined. To do this, the tree is again traversed given the recent history of the player's attacks. Let this node be called m. Note that m is always the parent of n_{train}. The children of m are the possible next attacks of the player. Let C be the summation of the counts of the children of m, which m maintains. Let c_i be the count for child i. Then the statistical probability of the attack represented by child i is given by P(attack = i) = c_i / C. This tree structure allows the HMM to be scalable to larger values of MAX_N_GRAM, but the size of the tree grows exponentially.

5.4.2 Matrix of Sets

The regular tactics model, counter tactics model, and NPC attack model all use a matrix of sets as their data structure. This is implemented in the class C2dStatModel, which is actually a template so that it can be used for both tactics (CTactic) and attacks (CActionAttack). Each of these models takes into account exactly two discrete parameters, although not the same two parameters. Regardless of these parameters, the fact that there are two allows for the use of a matrix, where the row index is the first parameter and the column index the second. Each cell in the matrix contains a set of either CTactics, in the case of the tactics models, or CActionAttack in the case of the NPC attack model. Each element of a set has a point value. A high point value indicates the corresponding element is favorable. Point values are capped between a minimum and maximum value. It is important for all the point values to be always greater than 0 so that every element is at least always possible, although not necessarily probable. This prevents the NPC from ever discarding any tactic or attack. Capping the points below a certain value has a more subtle benefit. Suppose a particular tactic was not capped, and it was positively reinforced so much that its point value became much larger relative to those of all the other tactics. If the player suddenly switches to a new strategy, one that the NPC's favorite tactic is no longer effective against, the NPC will most likely continually choose this ineffective tactic until it has been negative reinforced to the point where its point value is on par with those of the other tactics. This process of unlearning can take a very long time if the point value is not capped at a maximum. Furthermore, if the point value of one tactic is much larger than the others, the NPC is much less likely to explore its other options. Thus the maximum point value provides a way to adjust the amount of exploration by the AI system.

Recall that these three models are used for production, as opposed to prediction. To pick an element, a random number, r is generated between 1 and the total points in that set. The set is then traversed starting from the first element, and a running sum of the points of each element traversed so far is kept. At any time, if r is less than the running sum, the current element is picked. This is a method of performing probabilistic production, where the element with the highest point value has the most likelihood of being picked, but it will not always be picked.

When one of these elements, either a tactic or an attack action, is executed, it is placed into a log, which is another class called CStatNodeLog. The tactics models share a single tactics log, while the NPC attack model has its own separate attack log. Recall that a reinforcing event is when either fighter has taken damage, or when the NPC has performed an unsuccessful attack. When a reinforcing event occurs, the CStatNodeLog's reinforce method is called. This method takes a single parameter, which is the amount of reinforcement. The most current element in the log is reinforced by the given amount, that is to say, that element's point value is incremented by the given amount. The amount is then multiplied by a falloff factor, which is within the range [0, 1]. The next most recent element in the log is then reinforced by this reduced amount. This continues until the end of the log is reached. Figure 16 shows this idea with a falloff factor of 0.5. Note that since points are integers, they are only reinforced by integer amounts.

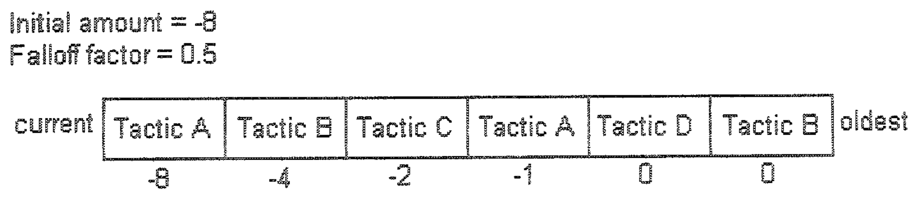


Figure 16. Reinforcement with falloff factor.

The idea behind this is that the more recent an element is, the more likely it had a high influence on the outcome that led to a reinforcing event. However, older elements may have had some influence as well, but less. Therefore, the amount of reinforcement is reduced proportionate to the time since the element occurred. Note that a particular element, for example a high-weak attack, could have multiple occurrences in the log, and therefore be reinforced multiple times. Since this process of reinforcement directly modifies the point values of the attack actions or tactics involved, the probabilities of performing those affected elements, which is dependent on their point values, are instantaneously changed.

6. RESULTS

6.1 Adaptive AI Effectiveness Survey

In order to measure the effectiveness of the adaptive AI system of *Alpha Fighter*, a small survey was conducted where ten people were asked to play two versions of the game. The only difference between the two versions was that version A had the adaptive capabilities enabled, while version B was non-adaptive. All other aspects of the game, including the other parts of the AI were exactly the same. The testers were not told which version was which. They did know the two versions were different in their AI, but did not know the details of how the AI was different. After playing through both versions, they were asked to answer a set of five questions, the averaged results and standard deviations of which are in Table 4.

Table 4. AI effectiveness survey results.

Survey Questions. Answers from 1 - 10 (1 = low, 10 = high)	A	B
Rate the intelligence of the AI.	7.7±1.1	5.7±1.5
Rate the difficulty of the NPC.	8±0.9	6.2±1.3
Rate the amount the NPC changed its tactics over time.	6.8±0.6	5.6±1.4
Rate how much the change helped the NPC.	7.3±1.2	5.4±1.0
How fun is the game?	7.75±0.9	5.5±0.4

As seen, version A with its adaptive AI, scored higher on every question. The adaptive AI seemed to make the most difference in the difficulty of the game, and the players' perception of the intelligence of the AI. It made less difference in their perception of the game-play, in other words, how fun the players felt the game was.

7. CONCLUSION

From the survey and the author's own observations, the adaptive AI in *Alpha Fighter* certainly added some value to the game. It made the AI seem more intelligent to the player, and slightly increased the game-play. One of the goals of this project was to help realize the practicality of machine learning in game AI. In that sense, the project was also a success. The fact that *Alpha Fighter* used four adaptive AI models in various ways and that the AI system and the game as a whole were enhanced because of it, shows that machine learning can, and should be a major consideration in the design of a video game AI system.

7.1 Future Work

There are still various ways on which the *Alpha Fighter* AI system can be improved. For example, the learning behavior itself can change over time [Manslow 2004]. One way to achieve this is to adjust the learning rate, α of the AI. For example, α can start with an initial value of 1, where the AI system will tend to adapt and change its behavior very rapidly. As time passes, α can slowly be decreased to 0, in which case the AI will no longer adapt, and simply use the knowledge it has already developed.

Another way to enhance *Alpha Fighter's* AI is to dynamically adjust the tactics of the fighters, or even create entirely new ones. This would require a lower level AI production model, so that the NPC can explore different combinations of individual actions, instead of being tied to the sequences of predefined tactics.

Yet another way to improve the AI is allow it to look ahead to find actions that will lead to rewards in the long-term. At present, the AI system takes a greedy approach and simply looks for the best chance for an immediate reward. However, it could be extended to follow more along the lines of Q-learning [Manslow 2004], where basically the AI will explore several paths internally, and try to find one that leads to the highest overall reward. This path is not necessarily the same as the one with the highest immediate reward.

REFERENCES

CHARNIAK, E. 1996. *Statistical Language Learning.* MIT Press, Cambridge, MA.

KAUKORANTA, T., SMED, J., AND HAKONEN, H. 2004. Understanding pattern recognition methods. In *AI Game Programming Wisdom 2*, S. RABIN, Ed. Charles River Media, Hingham, MA, 579-589.

LARAMÉE, F. D. 2002. Using n-gram statistical models to predict player behavior. In *AI Game Programming Wisdom,* S. RABIN, Ed. Charles River Media, Hingham, MA, 596-601.

LUNA, F. D. 2003. *Introduction to 3D Game Programming with DirectX 9.0.* Wordware Publishing, Plano, TX.

LUNA, F. D. 2004. Skinned Mesh Character Animation with Direct3D 9.0c. http://www.moon-labs.com/resources/ d3dx_skinnedmesh.pdf.

MANSLOW, J. 2002. Learning and adaptation. In *AI Game Programming Wisdom,* S. RABIN, Ed. Charles River Media, Hingham, MA, 557-566.

MANSLOW, J. 2004. Using reinforcement learning to solve AI control problems. In *AI Game Programming Wisdom 2*, S. RABIN, Ed. Charles River Media, Hingham, MA, 591-601.

MOMMERSTEEG, F. 2002. Pattern recognition with sequential prediction. In *AI Game Programming Wisdom,* S. RABIN, Ed. Charles River Media, Hingham, MA, 586-595.

RABIN, S. 2004. Common game AI techniques. In *AI Game Programming Wisdom 2,* S. RABIN, Ed. Charles River Media, Hingham, MA, 3-24.

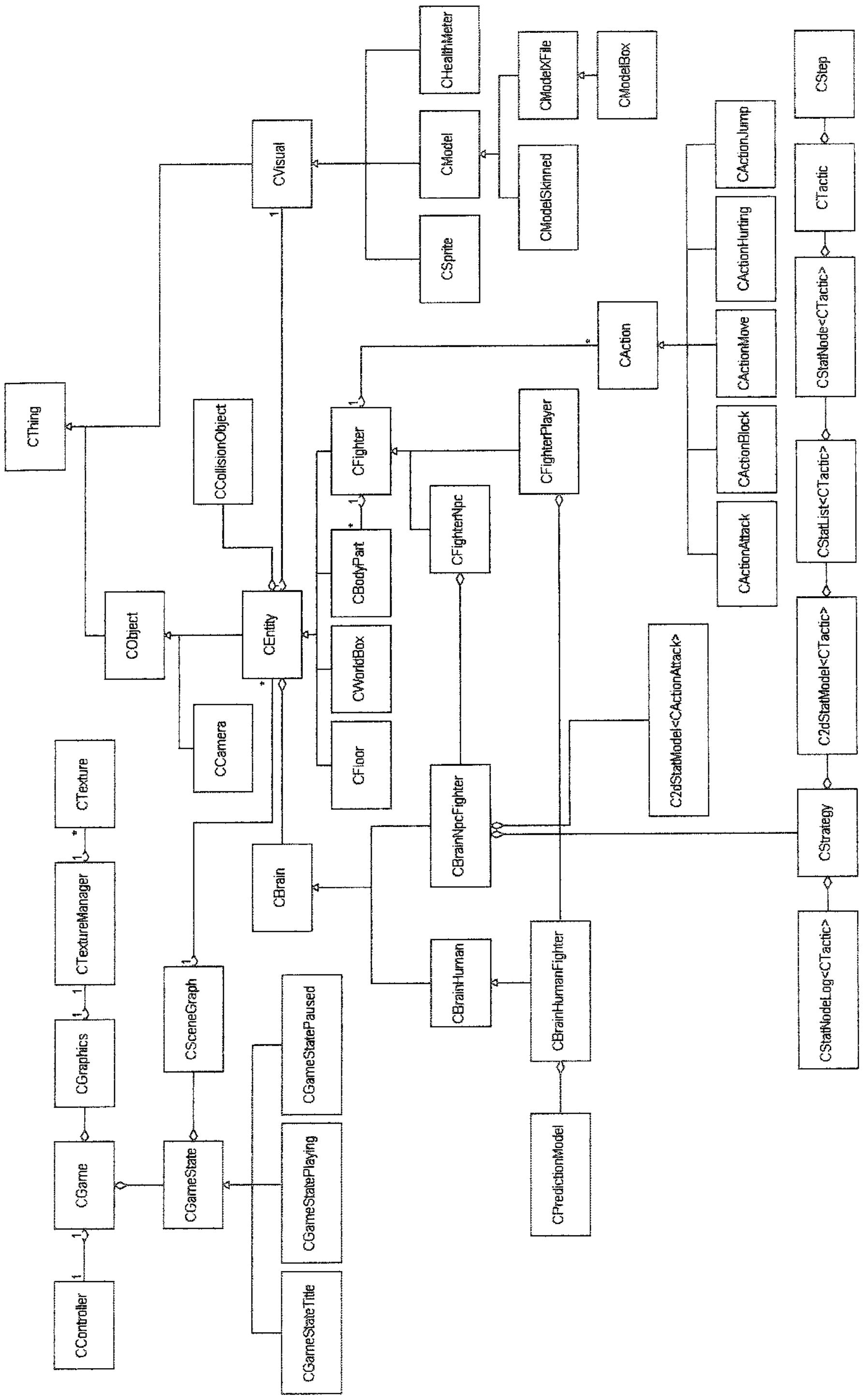
RABIN, S. 2004. Promising game AI techniques. In *AI Game Programming Wisdom 2,* S. RABIN, Ed. Charles River Media, Hingham, MA, 15-27.

RUSSELL, S, AND NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ.

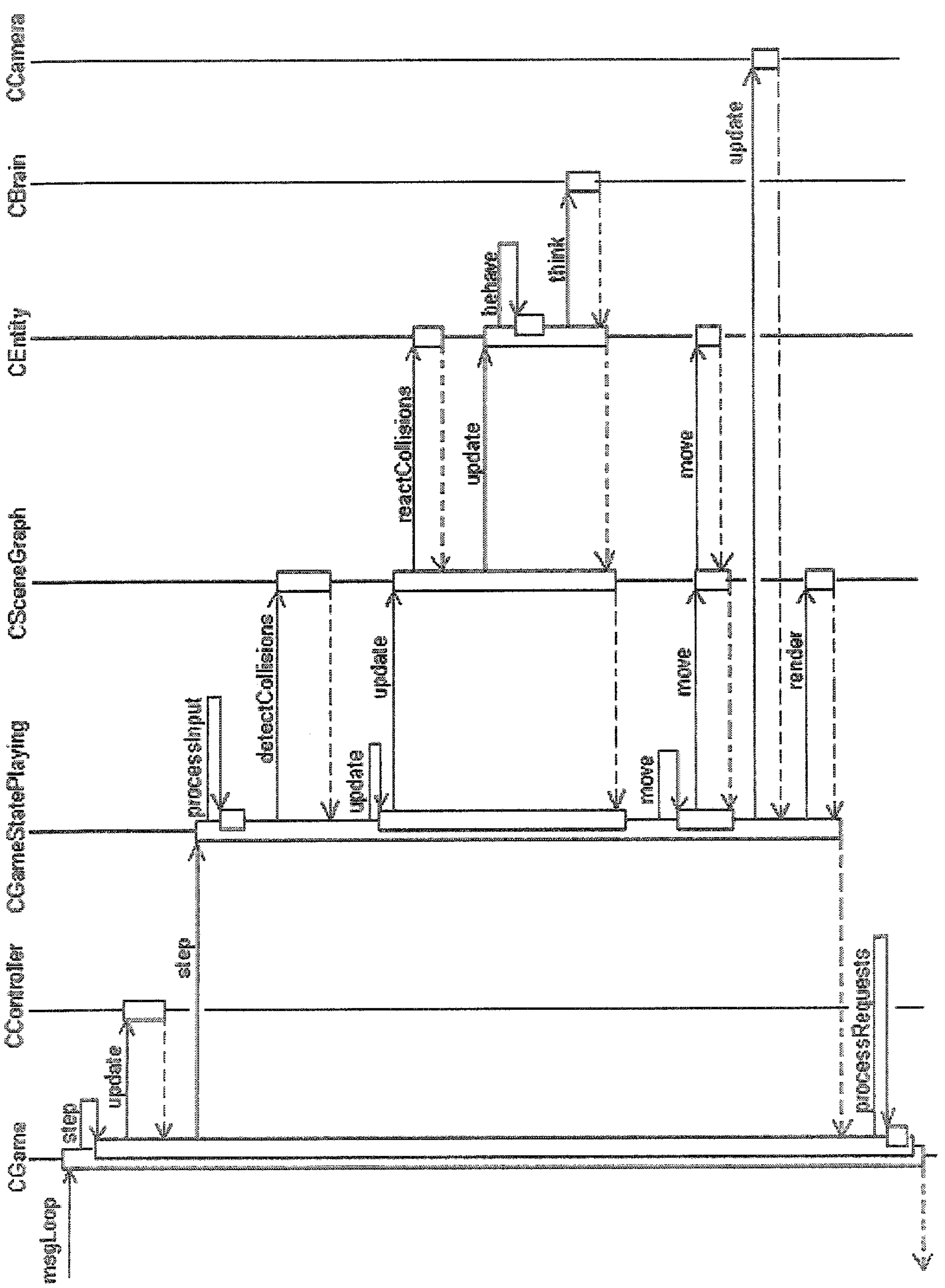
SANCHEZ-CRESPO, D. 2005. Tutorial: Machine Learning. http://www.gamasutra.com/gdc2005/features/ 20050307/ postcard-sanchez-crespo.htm.

TOZOUR, P. 2002. The evolution of game AI. In *AI Game Programming Wisdom*, S. RABIN, Ed. Charles River Media, Hingham, MA, 3-15.

APPENDIX A: GAME ARCHITECTURE UML CLASS DIAGRAM



APPENDIX B: SEQUENCE DIAGRAM OF PLAYING STATE FRAME



APPENDIX C: STAGE DATA FILE FORMAT

The stage data file has the extension .stg and specifies the parameters needed to construct a CStage object. The stage file contains exactly nine lines. The first six lines specify the textures to be used for the world box. The sample file below indicates which line is for which side of the world box. The seventh line specifies the texture for the floor of the stage. The eighth line is the size of the stage, which is always a square. The final line specifies the music file for the stage. The following is a sample stage file.

```
textures\\set 5\\neg_x.bmp
textures\\set 5\\pos_x.bmp
textures\\set 5\\neg_y.bmp
textures\\set 5\\pos_y.bmp
textures\\set 5\\neg_z.bmp
textures\\set 5\\pos_z.bmp
textures\\rock007.jpg
100
sound\\music.mid
```

APPENDIX D: FIGHTER DATA FILE FORMAT

The fighter data file has the extension .fgt and contains the information necessary to construct a CFighter object. The first three lines of this file contain the fighter's name, filename of its character model, and a scale factor. The next section of this file should contain lines beginning with "CO" indicating the line is a collision object specification. Recall that *Alpha Fighter* uses two types of collision objects, spheres and capsules. The first seven parameters of a collision object line are common for both types of collision objects. The format is:

CO BodyPartName Region Type Posx Posy Posz Radius

- bodyPartName: name of this body part.
- Region: 'U' for upper body or 'L' for lower body.
- Type: 'S' for sphere or 'C' for capsule.
- Posx: x coordinate in fighter's local space.
- Posy: y coordinate in fighter's local space.
- Posz: z coordinate in fighter's local space.
- Radius: radius of the sphere or of the capsule's spherical caps.

The capsule collision object type has four additional parameters:

Height Majorx Majory Majorz

- Height: height of the cylindrical portion of the capsule.
- Majorx: x component of the major axis vector.
- Majory: y component of the major axis vector.
- Majorz: z component of the major axis vector.

Following the "CO" lines are the attack list, or "AL" lines of the following form:

AL BodyPart1 BodyPart2 ...

Following the string "AL" is simply a list of the names of the body parts that are part of that attack. Likewise, the block list, or "BL" lines have the same format. There should be exactly two "BL" lines immediately following the "AL" lines. The first "BL" line specifies the body parts affected when blocking high, while the second "BL" line is for a low block. The next page is a sample fighter data file used for the character Perry Johnson.

```
Perry Johnson
models\ \perry.x
0.2
CO neck           U S  0.0    118.0  -2.5  9.5
CO sternum        U S  0.0    92.0   -3.0  13.0
CO pelvis         L S  0.0    68.5   -3.0  13.0
CO left_wrist     U S  -16.0  57.0   7.0   4.0
CO right_wrist    U S  16.0   57.0   7.0   4.0
CO left_shoulder  U C  -16.0  92.5   -5.0  4.5   17.0  0.1    1.0    -0.1
CO left_elbow     U C  -17.0  69.7   0.8   3.5   17.0  0.0    1.0    -0.5
CO left_hip       L C  -6.0   50.0   -2.5  6.2   24.0  -0.05  1.0    0.15
CO left_knee      L C  -5.0   21.0   -7.2  4.3   25.0  -0.05  1.0    0.0
CO left_ankle     L C  -4.3   1.5    -0.5  3.0   11.0  -0.1   -0.15  1.0
CO right_shoulder U C  16.0   92.5   -5.0  4.5   17.0  -0.1   1.0    -0.1
CO right_elbow    U C  17.0   69.7   0.8   3.5   17.0  0.0    1.0    -0.5
CO right_hip      L C  6.0    50.0   -2.5  6.2   24.0  0.05   1.0    0.15
CO right_knee     L C  5.0    21.0   -7.2  4.3   25.0  0.05   1.0    0.0
CO right_ankle    L C  4.3    1.5    -0.5  3.0   11.0  0.1    -0.15  1.0
AL left_hip left_knee left_ankle
AL left_hip right_hip left_knee right_knee left_ankle right_ankle
AL left_shoulder left_elbow left_wrist
AL right_hip right_knee right_ankle
AL left_shoulder left_elbow left_wrist
AL right_hip right_knee right_ankle
AL left_shoulder left_elbow left_wrist
AL left_hip left_knee left_ankle
BL neck sternum left_wrist right_wrist left_shoulder left_elbow right_shoulder
right_elbow
BL pelvis left_hip left_knee left_ankle right_hip right_knee right_ankle
```

APPENDIX E: TACTICS FILE FORMAT

The following shows the format for a tactics script file followed by a simple example. In a tactics file, any text not within a BEGIN_TACTIC and END_TACTIC pair is not parsed. A tactic definition has the following form:

```
BEGIN_TACTIC Name CounteringFlag Points
StepType Param1 Param2 ...
...
END_TACTIC
```

- BEGIN_TACTIC and END_TACTIC are the string literals.
- Name: The name of the tactic.
- CounteringFlag: 0 = regular tactic, 1 = counter tactic.
- Points: An integer of the initial point value.

Each line between BEGIN_TACTIC and END_TACTIC specifies a step for that tactic. The order that the steps are specified is the order the steps will be executed. The form for specifying each type of step follows:

MoveToRange range
Range is an int within [0, NUM_DISTANCE_RANGE - 1].

MoveWithinRange range
Range is same as for MoveToRange.

SideStep direction time
Direction is 'L' for left, 'R' for right, or 'A' for either.
Time is the seconds to side-step for.

Attack
No parameters.

Block maxTime
maxTime is number of seconds to block at most.

Wait time
Time is the seconds to wait.

Jump direction
direction is character in {'F', 'B', 'R', 'L', 'N'}, to indicate which direction to jump: forward, back, right, left, or none respectively.

JumpAttack direction
direction (see Jump above).

Below are two sample tactic definitions, a regular tactic and a counter tactic.

```
-------------------------------------------
Get within kicking range and do two attacks in a row.
-------------------------------------------
BEGIN_TACTIC Double_combo 0 1
MoveWithinRange 1
Attack
Attack
END_TACTIC
-------------------------------------------
Blocks for at most 2 seconds then throws a counter attack.
-------------------------------------------
BEGIN_TACTIC Block_2_Attack 1 1
Block 2
Attack
END_TACTIC
```

APPENDIX F: ABOUT *ALPHA FIGHTER* AND CONTENTS OF CD

Alpha Fighter uses Microsoft's DirectX 9.0c library. The DirectX SDK documentation and the work of Luna, [Luna 2003] and [Luna 2004], were consulted as primary references for issues concerning DirectX. All other design and development are solely the work of the author. The included CD contains the items summarized below. These resources are also available at the *Alpha Fighter* website at http://www.leolees.com/alphafighter.html.

Directory	**Contents**
/source	This directory contains the complete source code for *Alpha Fighter*, including the project files for Visual Studios 7.1, and all necessary resources to compile and run the game.
/thesis	This directory has an electronic copy of this thesis report in Microsoft Word (.doc) format.
/userguide	This directory holds the user's guide for *Alpha Fighter*.
/game	The directory contains the executable of *Alpha Fighter* and all necessary resources to run the game.
/prepprojects	This directory contains the small programs developed while research various AI methods for *Alpha Fighter*. These programs are mentioned in Section 2.