Using an Inference Engine for AI in the Office Tactics Video Game

Arturo I Concepcion[1], Edward Munoz[2], Matthew Hawkins[3], and Diane Balane[4]
[1]School of Computer Science & Engineering, California State University, San Bernardino, CA, United States
[2]iMedRis Data Corp., Redlands, CA, United States
[3]School of Computer Science & Engineering, California State University, San Bernardino, CA, United States
[4]Department of Art, California State University, San Bernardino, CA, United States

Abstract - *FSM (or its improvements) is the most common method employed when implementing AI on video games. Its major advantages are its simplicity and the ease of implementation but its greatest disadvantage is the predictability of the next state, which could lead to the player predicting the next step the game will take. Although there are some improvements done in FSM to alleviate this predictability, the inference engine allows a reasoning process and could come up with a strategy or move that the player might not be expecting. The inference engine consists of three parts: the knowledge base, the agenda, and the working memory. This paper developed an inference engine using the scripting language of UDK and applied to a video game, OfficeTactics. The resulting AI is very diverse and provides a lot of options that makes the game more exciting and enjoyable to play.*

Keywords: expert system, inference engine, knowledge-base, AI agents, and video game.

1 Introduction

Office Tactics is a video game designed by Danny Vargas when he was an undergraduate student in the School of Computer Science & Engineering, California State University, San Bernardino, in 2011. The game is a turn-based strategy and involves office workers who were laid off from their jobs causing them to retaliate in rebellion against management.

The paper describes the use of an inference engine to implement the AI of the enemy and friendly NPCs in the video game. Finite-state machines are the most common structures that are used for AI because of its simplicity and ease of implementation that produces good results. In spite of its advantages, FSM has many disadvantages: subject to unbounded growth of states, hard to maintain, and predictability of actions by the NPCs.

There were extensions made to FSM to overcome the disadvantages mentioned above. One is extending the states to offer Enter/Exit blocks and allowing event notifications to produce random or probabilistic transitions. Another is to allow FSM to have a stack-based history to track past states [6]. This was also extended to allow a state to transition to an entirely new FSM, making the FSM hierarchical. Even with these extensions, FSMs cannot perform pathfinding, reasoning or learning and so we propose the use of inference engines to overcome these deficiencies [7].

2 Office Tactics Video Game

2.1 Story

Office Tactics is a game that circles around the conflict of choosing between human and machine labor. The story starts off with three of our main characters, the Friendlies: the corporate employees who serve as our protagonists as well as your playable characters in the game. The three Friendlies are called into a conference meeting, where it is announced by the Boss that they are all fired (Figure 1).

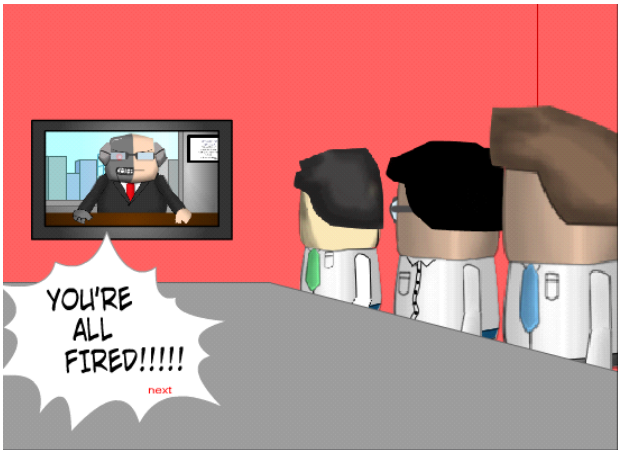

Figure 1. The corporate boss announces to the Friendlies that they are fired.

In a state of shock, the Friendlies are speechless as the Boss welcomes into the room their new replacements. To more of the Friendlies' shocks, they find that their replacements are, in fact, exact copies of themselves but in robotic form. The Boss

gladly explains to the Friendlies that the Robots are beneficial to both him and the company since they are more reliable, productive, obedient, and extremely cheap (Figure 2).

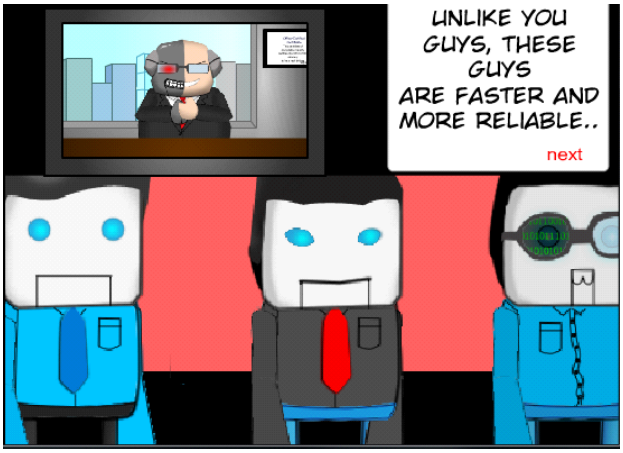

Figure 2. The corporate boss explains the advantages of having robotic employees versus human

Infuriated, one of the Friendlies, Joe Bob, stands up against this proposal and is challenged by the boss to stop him, thus forcing Joe Bob and his partners to fight for their right to voice against this decision as well as for the sake of keeping their job (Figure 3).

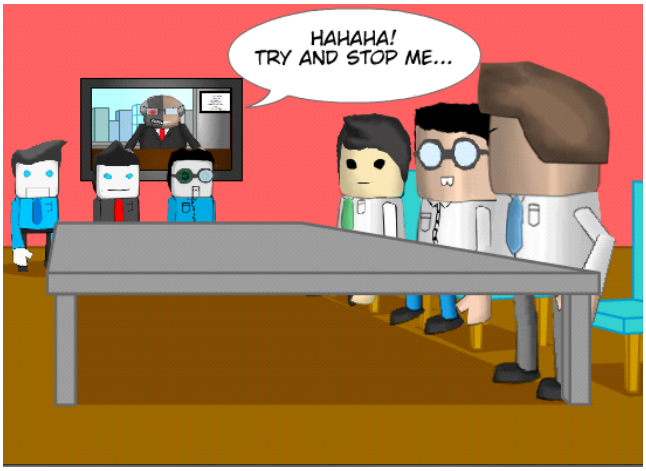

Figure 3. The Friendlies stand up to the corporate boss, opposing his plans

The game takes you through three levels, and in each level you play as the Friendlies, battling the Robots with each Friendly's different abilities. Some of the characters include:

- **Joe Bob (Brown hair, blue tie)** A regular employee that is also the most dedicated employee to the company. His abilities include Bash, Rush, Self Motivation, and Genuflect.
- **Slick Back (Black Hair, green tie)** A young man who is only interning for the company. Due to his young age and minor attachment to the company, he can come across as naïve or even irresponsible at times. His abilities include Bear Trap, Poison Dart, FlashBang, and Brain Drain.
- **John Doe (Black hair, glasses)** Although often picked on by other employees, John Doe is the IT Nerd as well as the brain of the company. The company would be nothing without his technical knowledge and skills and the employees know it. His abilities include Focused Shot, Rush, Self Motivation, and Genuflect.

Right after the final boss battle, the game switches to a cut scene of the infuriated, beaten Boss kicking the Friendlies out of his office. As the Friendlies stumble out of the office, the John Doe's glasses are knocked off onto the floor. His vision extremely blurred, John Doe drops down to his knees to look for the glasses, only to stumble upon the handle of a hidden file cabinet. The Friendlies gather together to force open the cabinet where they find what looks like a steering wheel to a ship. After the Boss leaves his office for a bathroom break, the Friendlies sneak into the office to figure out where the steering wheel belongs. Slick Back finds its rightful place by accidently locking the steering wheel into the Boss's desk. The Friendlies gather around to find out what it does by turning the wheel, which causes the building break off the ground and levitate into the distance, ending the game.

2.2 Game Design

Office Tactics is a turn-based strategy game for Windows PCs, similar to other games such as *Ogre Battle* and *Final Fantasy Tactics*. It is designed as a single player game centered around unit management and strategy. The game is built using the public version of Unreal Engine 3. *Office Tactics* challenges the player's ability to use a limited number of units to beat the ten planned levels of the game. Each of the player's units will level with each map completed and each enemy they defeat, but are gone for good when taken out by an enemy, making it so that the player has to be careful with how they deploy their units and how they approach each situation. There are six different unit types, with the AI having access to special Boss types as well. The player has a choice between either a melee or ranged physical type, a direct damage or control 'magic' type, and a healer or a buff/debuff type. The player starts with one of each, along with an extra ranged and melee character, though they cannot use all of them at once. Each level has a set limit on how many characters a player can have, determined by the cost of the units and the cap on spending for each level; typically around 2500 with each unit being 500.

Each level has a different mix of enemy units based on both the classes that the player can use, and Boss enemies that are either a combination of 2 of two class types, or a unique NPC that has a whole different move set from a normal class. The maps themselves are designed around a cartoony image of an office in chaos, with cover and obstacles based on cubicle walls and other things that may be found in an office, or whatever odd ball thing that might seem interesting or comical from a visual standpoint.

The characters are based on different character tropes and archetypes that would be found in something like *Office Space* or *The Office*, while the main antagonist can best be described as Gordon Gekko-1000. Enemies are designed to look like mechanical versions of the base classes, but as if built by a modern tech company; sleek, smooth, and cheaply made.

All the in-game audio and music is completely original, created specifically for the game. The music is meant to be light and comedic, with 5 unique tracks that play over each map, with a specific song for certain unique events like the tutorial and the final boss battle. The audio includes the effects that play with each unit's actions, and different office ambiance cues as the player looks around the map.

2.3 Game Mechanics

Mechanically the game is a turn-based system where, starting with the human player, each side takes turns controlling their units. On that player's turn, each unit may move and perform up to one action in any order. The game is played on a grid board, with the size based on the size of the map; the average map is a 64x64 sized grid, with each square being a 1 unit by 1 unit square in Unreal. At the start of a game, the player selects the units they wish to use from a list that lets them see their stats and abilities, along with the costs of each unit and the limit they have on that map. Units are deployed one at a time, until the player has spent the amount of points they want up to the cap.

Every unit has a set of seven stats that determine their effectiveness in combat. These stats are their Hit Points (HP), Ability Points (AP), Physical Power/Defense, Ability Power/Defense, and Movement speed. Each of these except for Movement is determined by a unit's class and level, with a unit's level determined by how much Experience it has gained. Ability Points (AP) are spent in units to use their special attacks, which are different depending on the unit using them, and any effect they have is based on the unit's Ability Power; if it is an offensive ability, it deals damage based on a comparison of the attacker's Ability Power to the defender's Ability Defense. Every unit has a basic attack that is either melee or ranged, depending on the unit, and deals damage similar to an Ability, except using the Physical stats. Movement speed is what determines how many squares a unit can move each turn. Movement can be blocked by other units and obstacles, making placement and movement order important.

Every attack and ability has a set of stats as well that, when combined with the each unit's base stats, determines how powerful they are. The stats are Range, Power, Area of Effect, which has either a cone, line, or burst shape; and AP cost for Abilities. Most units have a basic attack range of one square, and can attack in any direction around them in either a horizontal or vertical line, but not diagonally. Attacks and Abilities that have a range beyond one are targeted in the same manner as movement, with the action's Area of Effect shown on the grid. Bursts are centered on the player's cursor and can be centered anywhere in the action's range; meaning that units outside the actual range can be hit if they are still within the Area of Effect's radius. Line and cone actions are centered on the unit using it, and fire out in the selected direction similar to a Range one attack, but either a line of squares in that direction if a Line, or in a cone of squares in that direction if a Cone. As an example a basic ranged Attack, and certain abilities, is effectively an attack with an Area of Effect Burst of zero.

When a unit's HP reaches zero it is considered defeated, and if an NPC, the player unit that defeated it gains a set amount of Experience determined by the level of the unit compared to the type and level of the NPC defeated. When a unit gains enough Experience to level, they gain it immediately and have their stats recalculated, along with regaining health and ability points up to their new maximum at that level.

Between levels each unit's HP and AP is replenished, however in game each of the caster types has a way to replenish AP, and the healer class obviously can restore the HP of units on the player's side; either with a weak Area of Effect heal, or a stronger single target heal.

Finally, the current goal of each level is to clear it of all enemy units, allowing the player to advance to the next level of the game. There are plans to include different mission types, but those are time dependent.

3 Inference Engine

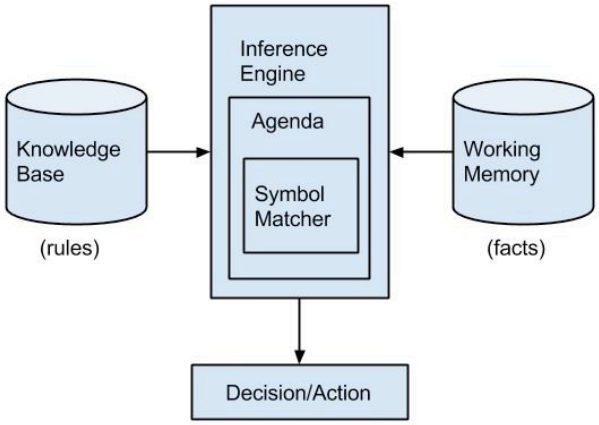

Figure 4. Architecture of an expert system showing the inference engine

An inference engine, see Figure 4, is designed to employ methods of plausible reasoning. Indeed when designing an inference engine there are many methods that can be implemented to perform reasoning, such methods include forward chaining, backward chaining, fuzzy logic, and bayesian logic, to name a few.

Among the many different methods of reasoning that could have been used for *Office Tactics,* forward chaining stood out as being the most practical and easiest method to implement into the game. Forward chaining allows for the inference engine to make use of the if-then structure of the rule base in order to locate justified rules that allow the engine to conclude the consequent (Then Clause), as a result new information is added to the games working memory [5].

For example, suppose the goal is determine the best possible move that the AI should make based on the following rules:

- **If** Target distance is between 6 and 10 **Then** Move to target
- **If** Target is 1 space away **Then** check Player's available action points
- **If** Player's available action points are less than 10 **Then** perform basic attack

Assuming the following facts:

- Target is 1 space away
- Player's available action points is 6

With forward chaining, the inference engine can ascertain that the best possible move is to perform a basic attack in three steps:

- Target is 1 space away

Based on that logic, the inference engine is then asked to check the Player's available action points.

- Player's available action points is less than 10

Based on rule 3, the inference engine can derive

- Perform basic attack

Just as the name "forward chaining" sounds, it is based on the fact that the inference engine starts with data and reasons its way to the best possible move.

3.1 Knowledge Base

A knowledge base is comprised of two key components: facts, and rules. Facts are assertions that change rapidly through the course of a program, and generally represent short-term information corresponding to rules that have been proven true. Rules form the representation of knowledge that originates from an expert on a specific domain. Unlike the short-term duration of facts, rules serve as long-term information about how to generate new facts. In most expert systems, rules are expressed in natural language typically following an IF conditional THEN consequent structure, for example:

Rule 1
IF the target is 1 space away, **AND** available action points is less than 10 **THEN** the AI should perform a basic attack.
Rule 2
IF the target is 1 space away, **AND** available action points is greater than or equal to 10 **AND** the target is below 50% health points. **THEN** the AI should perform a stronger ability.

These rules cannot directly be embedded in program code due to the nature of natural language; instead they can be represented by decision trees, semantic nets, or predicate calculus.

3.2 Agenda

An Agenda is a prioritized list of rules prepared by the inference engine [8]. Rules put onto the agenda are satisfied by the facts from working memory. When the inference engine locates facts that satisfy the conditional portion of a rule it adds the rule to the agenda. In order for a rule to be put on the agenda the entire conditional portion of the rule has to be proven true, even when there are multiple patterns that need to be satisfied, for example:

Rule 2
IF the target is 1 space away, **AND** available action points is greater than or equal to 10 **AND** the target is below 50% health points. **THEN** the AI should perform a stronger ability.

In this example, Rule 2 has two conditions: the target has to be 1 space away, and available action points need to be greater than or equal to 10. If both of these conditions are met then this rule gets added to the agenda. When the inference engine is ready to fire rules from the agenda there are several methods that can be applied, such methods include FIFO (First In First Out), precedence, and number of antecedents.

- **First In First Out** works by adding rules to the agenda in the order in which they were proven true in the knowledge base, and the inference engine fires those rules in the same order in which they appear in the agenda.
- **Precedence** is a system in which rules are given precedence values, and rules with higher precedence are fired first.
- **Number of Antecedents** is a system that fires rules based on the number of antecedents. According to this kind of logic, rules that have more antecedents have more requirements, which are likely to be more accurate, and more likely to provide the shortest path to the goal.

3.3 Working Memory

Working memory consist of a collection of facts that are stored to be used later on by the rules. The inference engine uses working memory to retrieve known facts in an attempt to satisfy the conditional portion of a rule. Facts that are applied to rules generate new facts that are added to the working memory forming a continuous cycle.

4 Implementation Using UDK Scripting Language

All the components of the inference engine used by the Office Tactics video game were written in the Unreal Development Kit's native scripting language UnrealScript. UnrealScript is used for authoring game code, and game events. Similar to high-level programming languages such as Java, and C++, UnrealScript is object-oriented. The language was designed to be simple, yet powerful enough for high-level game programming.

4.1 Working Memory Implementation

The goal of working memory is to provide new facts that have been generated by previous calls to the inference engine. The engine constantly updates simple facts, such as an enemy units current health, throughout game play. For example, a call to working memory is made in the beginning of each of the AI unit's turns. This call is also done before any reference to knowledge base to ensure that the facts being retrieved are those that have been updated by the previous turn.

```
var CorPIE_WM  WorkingMem;
WorkingMem = new ()  class  'CorpIE_WM' ;
```

Figure 5. The above snippet of code illustrates a call to working memory in UnrealScript.

Most of the information needed by the working memory is data regarding the player units that are targeted by AI units. Information such as ability points and health points are of primary concern. Before executing any rules in the knowledge base calls to working memory are made to retrieve facts regarding the AI's target. Some examples of functions used by working memory to retrieve data include:

```
/ / TODO:  Get pawn's hit points accessor function
function int getCurrentHealthPoints (CorpPawn pawn)
{
  return pawn.CurrentHealthPoints;
}
```

Figure 6. The above snippet of code is an accessor function that returns a pawn's current health points

```
/ /  TODO:  Get pawn's ability points functions
function int getCurrentAbilityPoints (CorpPawn
pawn)
{
  return pawn.CurrentAbilityPoints;
}
```

Figure 7. The above snippet of code is an accessor function that returns a pawn's current ability points

4.2 Knowledge Base Implementation

The rules in a knowledge base are expressed in natural language, but unfortunately computers don't have the capabilities to understand natural language due its ambiguity. However natural language is an exceptional start when designing the expert rules that will be included into the knowledge base. Once a list of expert knowledge has been accumulated it is only a matter of translating those rules from natural language into high-level computer language such as UnrealScript. For example, when determining the actions of a support unit, several facts are needed to ascertain the best move that should be made. Consider the following rules as an example:

Rule 3
IF the target is 1 space away **AND** available action points is greater than or equal 20, **AND** the target has 50% or less health, **AND** only allies are in the area. **THEN** perform a Group Heal.
Rule 4
IF the target is 1 space away **AND** available action points is greater than or equal to 10, **AND** the target has 50% or less health. **THEN** perform a heal on target.

Rules 3, and 4 can be translated into UnrealScript, the following is an example of the kind of an if-then structure taken from the knowledge base.

```
If(TargetPawnDistance == 1)
{
  if(UnitActionPoints  >== 20 &&
                TargetPawnObjective.CurrentHealthPoints
                <=  (0.50
                *float(TargetPawnObjective.MaxHealthPoi
                nts)) &&
                GroupHealArea.Length > 1 &&
                AllyOnlyArea(1))
                {
                  / /GroupHeal
                AgendaQueue.AddItem(Ability);
                }
                else if(UnitActionPoints  >= 10 &&
                  TargetPawnObjective.CurrentHealthPoin
                ts <= (0.50
                *float(TargetPawnObjective.MaxHealthPoi
                nts)))
                {
                  / /Heal
                  AgendaQueue.AddItem(Ability0);
                }
}
```

Figure 8. The above snippet of code represents rule 3 and rule 4 in UnrealScript.

Even though the original design for the rules are in natural language the statements can easily be expressed through the use of comparative operators, variables, and objects in UnrealScript. The 'then' portion of each rule is equally as imperative to the knowledge base as the conditional portion, because it is within this block of code that new rules enter the life cycle of the game. When a rule has been justified, a keyword, or symbol gets pushed onto a data structure for later use. For this particular kind of expert system a **symbol** represents various problems concepts, actions, and is used for applying strategies to reach a certain conclusion. In UnrealScript a symbol is represented as a constant, and stands for some concept related to the Office Tactics video game. Examples of the symbols used in the Office Tactics video game include:

- **Ability [0-5]**: This symbol is used to represent the type of ability a unit should use in a certain situation, in which case the AI goes into an Ability Standby State.
- **BasicAttack**: This symbol is used notify the system that a basic attack should be used on a target, in which case the AI goes into an Attack Standby State.
- **MoveToTarget**: This symbol is used to notify the system that the AI should move toward a target, in which case the AI goes into a Move Standby State.
- **Wander** This symbol is used to flag the system that there are no moves to be made, in which case the AI is allowed to make to traverse the map randomly.

These symbols were given names that represent the actions that they embody. These rules once proven to be true will cause symbols to be added onto a data structure that serves as the agenda of the inference engine.

4.3 Agenda Implementation

The agenda implementation for Office Tactics works a little differently than other expert systems. Generally an agenda would fire rules base on their precedence level. Rules with higher precedence leave the agenda quicker than those that have lower precedence. However for Office Tactics the agenda works off a FIFO principle. The agenda for this implementation is a queue that allows the inference engine to fire rules in the same order in which they appear in the agenda. A queue based implementation allowed for easier manageability, and development. The set of symbols representing different rules can be added onto the agenda in many different combinations, allowing for the AI to perform actions that are less predictable, and more likely to lead to greater results than a static finite state machine.

```
Function Agenda ()
{
  local int Action;
  NumberOfActions = AgendaQueue.Length;

  Foreach AgendaQueue(Action)
  {

    if(Action < 6)
    {
      ThisPawn.SetAbility(Action);
      If(NumberOfActions == 1)
      {
        GotoState('SpecialAbilityStandby');
      }
                  }
    else if(Action == 6)
    {
      if(NumberOfActions == 1)
      {
        GotoState('AttackStandby');
      }
    }
    else if(Action == 7)
    {
        GotoState('Standby');
    }
```

```
    else if(Action == 8)
    {
        WanderState();
    }
  }

  AgendaQueue.Remove(0, AgendaQueue.Length );
}
```

Figure 9. The above snippet of code represents the structure of the agenda used in the Office Tactics video game.

Based on the order in which the rules are added onto the agenda, and the unique constant value that each symbol is represented by, the agenda directs the system to the appropriate state until all the rules have been fired.

5 Conclusion

Historically expert systems have been less than enthusiastically received by the software engineering community [2], and has led many designers to favor finite state machine in their designs. However, FSM have their limitations such the lack of knowledge manipulation, predictability, and rigid design. These pitfalls in FSM are generally resolved through the use of Inference logic provided by an expert system. Some notable benefits of expert system integration include [1,3,4]:

- **Overcoming toy domains:** When integrated into software the expert system is able to be targeted at specific components, therefore, a small scale inference engine can make a meaningful contribution to large scale projects.
- **Ability to manipulate knowledge:** Experts systems allow of the manipulation of knowledge, this differs from conventional programming that generally only manipulates data.
- **Use of symbols:** A symbol is simply a string of characters that can represent various problem concepts that apply to strategies, and heuristics to reach a conclusion.
- **Quality of analysis:** Due to the fact that an expert system's knowledge base is comprised of information from multiple specialists in a specific domain it has been proven to perform better than their human counterparts.
- **Control:** Experts systems allow for improved control over various aspects of operations. Through the use of symbols to represent facts and ideas, users of the system are able to identify and monitor the progress of each inference result, which is unlike traditional systems that require the backtracking of chained events.

Inference engine logic is most effective in applications that require vast amount of expert knowledge and in cases where the best choice out of many permutations of outcomes is required. For these reasons inference logic is most often integrated into video game artificial intelligence to handle the high demand of possible outcomes that a complex game can produce. Such an example includes the application in Chinese chess, where the aim of the inference engine is to find the best move in the game's large decision tree. The endgame knowledge base for Chinese chess aids in determining infrequent winning nodes or inevitable draw nodes, which are then removed from consideration. As a result, the chances of determining a winning node go up as well as the engines chances of finding the best way to win. When dealing with large complex systems, it is apparent that FSMs have their limitation, as the possible permutation of outcomes become too large they become increasingly more difficult to manage. For reasons such as this, it is imperative to take advantage of the control, and expertise that an inference engine can provide for complex game systems.

References :

[1] Forsyth, R. (1984). Expert Systems Principles and Case Studies. New York, NY: Chapman and Hall Computing.

[2] Gillies, A. C. (1991). The Integration of Expert Systems Into Mainstream Software. New York, NY: Chapman and Hall Computing.

[3] Goldenthal, N. (1987). Expert Systems and Artificial Intelligence. Cleveland, OH: Weber Systems, Inc.

[4] Laswell, Lawrence. K. (1989). Collision Theory vs. Reality in Expert Systems. Wellesley, MA: QED Information Science, Inc.

[5] Robin. "Rule Based Expert Systems." *Artificial Intelligence: Articles On Artificial Intelligence,* November 1st, 2010. September 29, 2013. <http://intelligence.worldofcomputing.net/expert-systems-articles/rule-based-expert-systems.html#.Uku--IashcY>.

[6] Tozour, Paul, "Stack-Based Finite-State Machines," AI Game Programming Wisdom 2, Charles River Media, 2003.

[7] "Introduction to Game Development," 2nd Ed., Edited by Steve Rabin, Charles River Media, 2010.

[8] "Expert Systems/The Agenda." *Wikibooks*, January 3rd, 2008. September 29, 2013. <http://en.wikibooks.org/wiki/Expert_Systems/The_Agenda>.