FAST – Behaviour Modelling with Expert Systems

M I C H A E L  N A B B

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2008
FAST – Behaviour Modelling with Expert Systems

MICHAEL NABB

Master's Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2008
Supervisor at CSC was Stefan Arnborg
Examiner was Stefan Arnborg

TRITA-CSC-E 2008:044
ISRN-KTH/CSC/E--08/044--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.csc.kth.se
Abstract
The collection of techniques that control non-player characters in computer games are typically Artificial Intelligence methods. The graphics and physics simulations that are used in computer games has made great progress the last decade but that is not the case for Artificial Intelligence. This project studied alternative methods for simulating computer controlled soldiers in narrow spaces such as corridors and small rooms. A prototype was developed with the help of an expert system tool.

Sammanfattning
Den mängd tekniker som kontrollerar datorstyrda karaktärer i datorspel är oftast Artificiell Intelligens-tekniker. Den grafik och de fysiksimulationer som används i datorspel och har gjort stora framsteg det senaste årtiondet men det har inte de tekniker som kallas Artificiell Intelligens. Detta projekt tittade på alternativa metoder att simulera datorstyrda soldater i trånga utrymmen så som korridorer och små rum. En prototyp utvecklades med hjälp av ett expertsystemverktyg.
Preface

Artificial Intelligence is an area that I got interested in because of one simple reason. AI in games was not convincing, I thought more could be done. After taking a few AI courses my interest only grew. One course in particular, Machine Learning, really got me hooked on the subject and I knew that I wanted to write my Master’s Thesis on the subject.

I was lucky to meet a representative from DICE at D-Dagen at KTH and the project grew from there. I’m very happy I got the opportunity to do my Master’s Thesis project at DICE and I would like to thank everyone at DICE that made this possible.

Michael Nabb, June 2007
**Contents**

**Introduction** ........................................................................................................................................1

**Problem Specification** .......................................................................................................................2

**Theory** ..................................................................................................................................................3

<table>
<thead>
<tr>
<th>AI Engine</th>
<th>Inference engine</th>
<th>Knowledge-base</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Related Work** ...................................................................................................................................5

<table>
<thead>
<tr>
<th>Tacair-Soar</th>
<th>Soar Quakebot</th>
<th>MOUTBot</th>
<th>Anticipation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Methods and Solution** .....................................................................................................................7

**Expert Systems** ..................................................................................................................................7

<table>
<thead>
<tr>
<th>CLIPS</th>
<th>Features</th>
<th>Rules</th>
<th>Agenda</th>
<th>CLIPS vs. Soar</th>
<th>FAST</th>
<th>NPC Tactics</th>
<th>Positioning</th>
<th>Environment Features</th>
<th>Implementing Anticipation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

**Results** ..............................................................................................................................................15

**Conclusions** ......................................................................................................................................16

<table>
<thead>
<tr>
<th>Expert system vs. Finite State Machine</th>
<th>Anticipation</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

**Summary** ............................................................................................................................................17

**References** ......................................................................................................................................18
Introduction

Artificial Intelligence is an important part of computer games, despite that, advances has not been made in the last decade like those in graphics or physics.

Artificial Intelligence (AI) in modern computer and video games has traditionally taken a back seat to other more prominent features like graphic fidelity and physics simulations. While graphics and physics in modern games are very well developed, AI has perceivably not improved as much. When reading a game review it’s not uncommon to find comments about “stupid AI”.

AI is a quite broad field of techniques but one could sum them all up with one sentence: AI strives to give computers or machines the impression of being intelligent. Many examples of AI techniques can be found in everyday things but one of the areas in which is has become very common is computer games. Many different AI techniques like for example path finding, which is a collection of techniques to decide the optimal path of moving a unit in a certain space given the limitations of the unit and the space, can today be found in computer and video games. According to some, in particular Laird and van Lent (2001), computer and video games are also an excellent platform for developing AI.

The idea for this project came about when DICE at a meeting expressed that they had problems with computer controlled characters, also called bots or non-player characters (NPCs), navigating in narrow spaces and to have the bots support the player in a helpful way in their as of yet unreleased video game Battlefield: Bad Company. After further discussions it was decided that the project would focus on the behaviour of friendly combatants in small groups, or squads. Because of this focus, the project was given the name Friendly Automated Support Team, or FAST.

It was also decided during the meeting where the first idea was spawned that the project was too big to be done by one person, so it was split into two parts. In the beginning it was not clear exactly how the project would be split, but it was agreed that further reading would be necessary before the project could be split into two equally demanding and separate projects.

After the first initial weeks of research the FAST project was divided into two parts, the FAST Extension and the FAST Foundation. This report is concerning the part of the project that is called the FAST Foundation. The Foundation is the underlying part which controls and decides what each NPC in the mentioned squad should do when faced with different situations. The FAST Extension is then responsible for using this information to control entities in a simple graphical interface.

The reason for studying AI related problems like the one that DICE is facing is because of the perceived lifelessness of modern computer game AI. Bots will be considered smart as long as they don’t stand out, therefore, the slightest odd behaviour will immediately take away the illusion of intelligent behaviour. One major focus of the FAST project is therefore to make sure that NPCs controlled by the module developed as a part of FAST makes sound human like decisions.
Problem Specification

The perceived support provided by friendly Non-Player Characters in modern First Person Shooter (FPS) games is the focus of this report as are suggestions for solutions to the problem. This chapter provides a background to the problem.

The traditional approach to AI in computer games has been to make finite state machines (FSM) that always react in a certain way when faced with a certain situation (Krajewski, 2006). This solution is relatively easy to develop but won’t produce very realistic behaviour since it will always react in the same way when faced with a particular situation.

Although AI controlled NPCs have, with more work, been able to do more complex actions their actions seldom take into account the vital dimensions that humans do; the past and the future. Anticipation of future actions of other characters in a game, both computer controlled and player controlled comes natural to humans but is vary hard for NPCs that rely on traditional FSMs. The reason for this is that in an FSM each distinct state of an NPC can be in has to be hard coded, this is hard enough with just taking into account static environments so if you would want your NPC to also take into account dynamic information such as other NPCs or a player controlled entity and have the FSM anticipate their intentions that would add a lot of complexity to the problem. This paper will among other things present a solution to the problem of adding anticipation to NPCs without adding as much complexity.

Another problem in current generation games is that friendly NPCs can be perceived as a burden rather than a help, for example, they may position themselves in a way that hinders the player from advancing or maybe place themselves in the line of fire. Adding anticipation to an NPC could greatly improve the projected image of the NPC helping rather than disrupting the play of the game.

Early on in the project it was decided that the scope will be limited to movement and positioning in narrow indoor spaces. This was for several reasons, mainly because DICE expressed that they’ve had trouble with these kinds of situations in the past, also it seemed to limit the project enough to fit with the time constrains.

This resulted in the following project scope, concerning the complete FAST project:

- Have access to both static and dynamic information about the player (e.g. position).
- Have static game domain knowledge (e.g. tactical and environmental knowledge), limited by the knowledge needed for the use cases.
- Have dynamic game knowledge (e.g. state of the environment) based on the same information that is available to the player.
- Be able to estimate the player’s intentions and goals based on the NPC’s own knowledgebase.
- Adapt NPC behaviour, both individually and as a coordinated group, to support player’s intentions and goals.

Out of these five points the goal of the FAST Foundation would be to handle three. The project would strive for FAST Foundation to have static game domain knowledge, would be able to estimate player’s intentions and would adapt its behaviour. The other two points would be handled by FAST Extension which would in turn forward that information to the FAST Foundation.
Theory

This chapter presents the available work that has been the basis for solving this problem. Also previous work that is similar in nature to this project is presented.

As mentioned earlier, FSMs have traditionally been used to control NPCs in computer games. The main drawback is that the decisions can become very erratic if the current situation is changed frequently. What can also happen is for NPCs to get stuck in indecision loops, that is, first they decide to do one thing and then right after decide to do another and finally deciding that the first thing is the best, and so on, because the given situation presents many options. One solution to this problem has been to script more and more of the NPC behaviour. Scripts on one hand can perform very complex tasks suited for a particular situation but can not react to changes in the situation that require new actions in an adequate way. Once a script fires it runs for a predetermined set of time until finished. Another problem is that if the game requires a big amount of scripts the work needed to make scripts for all situations can be overwhelming.

This gives us two types of AI engines that have traditionally been used, the stimulus-response machine (FSM) and the script-based machine. What these two solutions and other AI engine solutions generally have in common though is that they have both advantages and drawbacks. The stimulus-response machine is very reactive in that it is good at adapting to the current environment but fails to keep recent sensor information in mind when making decisions. The script-based machine on the other hand is very good at producing context specific reactions that are perceived as realistic but is not as reactive as a stimulus-response machine. This can be illustrated by a simple example.

Say that an NPC decides to kick in a door; such action would probably have to be scripted but let’s disregard that. Now if an enemy appears and starts firing at the NPC while the NPC is approaching the door a stimulus-response machine would immediately acknowledge the threat and fire back instead of kicking in the door. After the fight is over the stimulus-response NPC will however not remember that he was kicking in the door before the fire fight and might now decide to do something else. Behaviour such as this is hardly realistic, a human being is context aware and will remember what he was recently doing and can therefore take this into consideration when deciding what to do next.

Now consider the script-based NPC. He would continue kicking in the door regardless of what happens and will ignore the fact that he is fired upon and that returning fire would be more appropriate than kicking in a door. This behaviour is also unrealistic.

The proposed solution is to provide an AI engine that combines the strengths of the mentioned solutions without keeping its limitations.

AI Engine

It has been suggested by Laird and van Lent (1999) that an AI engine should have the following design goals:

- Reactive
- Context specific
- Flexible
- Realistic
- Easy to develop
The engine needs to be reactive so that NPCs will change their behaviour quickly when a new situation arises or if the current situation changes enough to warrant another strategy. Context specific means that the engine always produces actions that are appropriate for every situation and also that the past actions of the agent and past sensor information should be consistent with these actions. Flexible agents means that they both have a choice of high level tactics and a choice of low level behaviours with which to carry out the tactics. Flexibility is not, according to developers at DICE, as important as other aspects, but it is still considered because it will probably result in more interesting behaviours. Realistic simply means that the agents behave similarly to what we humans would expect other humans to behave in the situations the agents are faced with. Although it is debatable if the high level of ease at which you can develop AI behaviour is a requirement it certainly will be beneficial to the development of better and more complex behaviours if development is easy. It almost goes without saying that each component in an AI engine must be designed carefully to fulfil all of these requirements. These design goals where chosen for the overall FAST project since they corresponded well with what the project definition stated.

Laird and van Lent also suggest that the AI engine should be comprised of three parts, the inference engine, the knowledge-base and the interface.

**Inference engine**

The inference engine applies the knowledge and behaviours stored in the knowledge-base to the current situation stored in memory. The job is to select which knowledge that is relevant to the current situation. A general description of the inference engine cycle is:

1. Perceive – Accept sensor information
2. Think – Select and execute knowledge
3. Act – Execute actions

The inference engine defines what is and isn’t possible for the other parts of the engine. For example, the inference engine defines how the knowledge-base must be represented.

**Knowledge-base**

The knowledge of the AI engine is modelled in a hierarchical way with an agent’s goal and tactics at the top and more specific actions further down in the hierarchy. Each goal or behaviour is called an operator and at each given level only one operator can be selected in each decision cycle. This is highly context specific because each operator is selected depending on the current situation and the current goal but it is also very reactive since if the situation changes other sets of operators are immediately selected to better fit the situation. It is also flexible because the agent could potentially respond to a given situation in different ways as long as the programmer provides the option in the form of different behaviours. A very similar type of knowledge hierarchy was also suggested by Schlenoff et. al (2005) in an unrelated paper.

**Interface**

The interface, called the framework in the FAST project, is the layer between the AI engine and the system that uses the AI engine, called FAST engine in this project. The framework is however outside of this paper’s scope and a detailed discussion about the framework can be found in the FAST Extension paper (Wedlund, 2007).
Related Work

There have been implementations of systems with an approach similar to the one described here, mainly in the form of a project called Tacair-Soar which is based on the Soar architecture. The Soar architecture will be discussed in a later chapter. Other implementations with similarities to the FAST project are the Soar Quakebot and the MOUTBot.

Tacair-Soar

Tacair-Soar (Jones et al., 1999) is a rule based system based on the Soar architecture that is able to generate large scale distributed military simulations. It was the first system developed using the inference engine architecture suggested by the University of Michigan Artificial Intelligence Lab. The system contains more than 5 200 rules which control the fixed wing aircrafts that are simulated and these rules are organized in the hierarchical way previously described. The goal of the project is to make the autonomous entities behave indistinguishably from a human controlled aircraft in a simulated environment.

Soar Quakebot

The Quakebot (Laird, 2000) was developed with the Soar architecture to interface with the video game Quake II. The rule base stored in the Soar inference engine communicates through an interface with the game to control an NPC that plays a deathmatch game, a kind of one on one duel, against a human player. Also added to the Quakebot is a rudimentary ability to anticipate an opponent’s actions, with just the sensory information that would be available to a human player. The design of the anticipation part used in the Quakebot project is the basis for the anticipation solution suggested in the FAST Foundation.

MOUTBot

Military Operations on Urban Terrain Bots is a system similar to the Soar Quakebot also developed with the Soar architecture to simulate opponents in an urban environment (Wray, 2004). Here the computer game Unreal Tournament II was used to interface with the Soar inference engine. In the MOUTBot project, emphasis was more towards making the bots behave in a plausible and tactically correct way. The system has been used by the US military to train personnel in close quarter urban environment combat.

Anticipation

In the article “It knows what you are going to do: Adding anticipation to a Quakebot” the author discusses why and how anticipation was implemented into the Soar Quakebot. While presenting his current work to game developers the author continually got the comment: “Does it anticipate the human player’s actions? If it did, that would be really cool”. This also ties back to one of our initial design goals, realism. For an NPC to appear realistic it would greatly benefit to have some type of anticipation since human players have this ability.

Laird notes that adding anticipation to his design was more straightforward than first thought. The solution is divided into three parts with one precondition. First off, don’t anticipate the player’s intentions all the time. Anticipation isn’t always necessary to make good decisions, for example, if an NPC is crouching besides a player and the player is standing still and both are firing at opponents anticipating player intentions are unnecessary, he will most likely continue to fire anyway. Anticipation should only be performed when applicable, partly for performance reasons. When anticipation is deemed necessary three steps are performed; internal representation of player state, prediction of player intentions and implementation of prediction into goal.
During the first step, internal representation tries to decide what state a player is in. Where is the player, what health does he have, what is he facing and so on. The NPC needs to know everything it in itself needs to know to be able to make decisions. If not all information is available, either the prediction will be cancelled because of lack of knowledge or the missing information is set to default values to simulate a guess as to what state the player is in. When the first step is over the NPC has an approximate or definite view of the player.

The next step is the actual prediction. Using the player’s current state the NPC forward projects what he would do, using his own knowledge-base, if he was in the exact same situation as the player is in. This way the NPC puts itself in the players shoes and tries to find out what he would do in that situation. This continues until one of two outcomes occur, either the NPC produces a prediction that is useful, for example, it predicts that the player will go outside and pick up a gun that is lying there, or it ends up in a dead end, maybe the situation is uncertain enough that a prediction is impossible. However, if a prediction is successfully done the bot can move on to the next step.

Incorporating the prediction into the bots decisions is much like adding other facts to the system. If you predict that the player will move up the stairs behaviours should exist that gives the bot the option to move up the stairs before the player so that the area at the top of the stairs can be cleared before the player enters it.

Strategies similar to this are for example used in chess games where the computer uses its own knowledge of the game to predict what the human player’s next move (or moves) will be. This technique can be implemented to perform well, seeing that a computer has been able to beat or draw human chess masters, referring to when the IBM built computer called Deep Blue beat then world champion Garry Kasparov in 1997 in a widely publicized event.
Methods and Solution

An expert system called CLIPS was used as the backend of the solution. How an expert system works, its benefits and drawbacks is analyzed as well as how it was utilized in the project. The chapter ends with a review of what was done and a theoretical situation that would lend itself well to a similar implementation as suggested in this work.

The majority of the work in this project was based on the previous work done by the University of Michigan Artificial Intelligence Lab. Throughout their work they have used an expert system called Soar as the foundation for their work.

Expert Systems

Expert systems is a field in AI research which focuses on developing systems with a high level of knowledge in a certain well defined problem domain (Giarratano and Riley, 2004). The first computer language developed that followed this approach was OPS5 (Wikipedia, 2007) developed at Carnegie Mellon University in the late 1970s. The knowledge may be represented in numerous ways, but a common way, and the way used in this work, is to represent knowledge as rules, so called rule-based system, in the form of \textit{IF–THEN} statements.

As depicted in Figure 1, an expert system is mainly comprised of two parts; the knowledge-base (also sometimes referred to as the rule-base) and the inference engine. These two components have two very distinct purposes. The knowledge-base is the collection of all the expertise available to the system and is therefore very specific to the problem that the knowledge-base is designed to solve. The inference engine on the other hand is a component that is completely independent of the problem the system is designed to solve. Its job is to reason given the knowledge in the knowledge-base and the facts provided by the user (user in the sense that it can be a person, one or many sensors or maybe both) and to derive answers from this reasoning. For example, if an expert system is designed to choose suitable wines for a multitude of different dishes then the knowledge-base contains the data needed to decide which wine to pick to serve with a Sirloin steak served with plums and dehydrated wine vinegar but the expert system could not decide whether the sun will shine or if it will be rain tomorrow. An inference engine on the other hand could be used in both of the applications, wine selection and weather forecasting, since it operates on the knowledge stored in a knowledge-base regardless of what kind of knowledge it is.

Another characteristic of expert systems is that they are data driven and not procedural. This means that the code isn’t executed in a predetermined way but the order of execution is decided by the facts stored in the system at run time. Some systems however, including CLIPS (which will be covered in the next section), do provide the developer with basic ways of writing procedural code too.
The basic difference between an expert system and a traditional problem solving program is how the knowledge is represented. In a traditional program both program and data structures are used to represent the knowledge while in an expert system only data structures are used. Because of this difference and that IF-THEN statements are used to represent the knowledge in the knowledge-base it’s very easy to understand the data structures for an expert of the problem domain that is represented by the system. Much of the knowledge possessed by humans can be represented as a simple IF-THEN rule (Giarratano and Riley, 2004). For example, IF the light is red THEN stop, while you are in your car driving.

CLIPS

In previous work done by various people connected to the University of Michigan Artificial Intelligence Lab (see Related Work) an inference engine called Soar was used to develop the different AI engines. In this work an expert system tool called the C Language Integrated Production System (CLIPS), which was originally developed by the American space agency NASA, was chosen instead because it is written in C, easily integrated into other C or C++ code (as well as software written in other languages) and is public domain software, which means it’s free to use in commercial applications. CLIPS is available for download at http://www.ghg.net/clips/CLIPS.html. Furthermore CLIPS has a high level of documentation available online in the form of the Reference Manual (Culbert et. al, 2006).

Features

CLIPS consist of the two elements that most expert systems contain; a knowledge-base and an inference engine. On top of this several other components are a part of the package. Most important is the working memory which stores all facts that are currently active. In the example of the traffic light the fact that the light is red, green or yellow would be stored in the working memory before any rule that matches this fact can fire. Another convenient and available feature is a graphical user interface and a wealth of debugging commands. As mentioned earlier CLIPS also provides its users with ways of writing procedural code which can ease some parts of the development that can be more easily written procedurally. Finally CLIPS contains the CLIPS Object-Oriented Language, more commonly referred to by the acronym COOL. COOL provides the user with the ability combine facts into objects, something that surely appeals to object oriented programmers.

Rules

As explained in the description of expert systems the knowledge stored in CLIPS is represented by rules. The rules contain a left hand side and a right hand side. The left hand side details what facts that need to exist for the rule to be activated, so for example, the light has to be red for the traffic light rule in the previous section to be activated. Each part of the left hand side is called a pattern, and so all patterns have to match a fact in working memory. The right hand side details what should be done if a rule is activated and then fired, this can include removing facts from working memory, adding facts to working memory or performing other actions, such as writing a message to a file. The traffic light rule could be written in CLIPS like this:

```
(defrule traffic-light-red
  (light red)
=>
  (assert (stop))
```

The first line defines that this is a rule and the name of the rule. The second line is the sole pattern of this rule, for this rule to be activated and placed in the agenda a fact that is exactly (light red) has to exist in working memory. The arrow on the third line marks the end of the left hand side and the start of the right hand side. Finally a command is issued, the assert
command means that the fact provided as an argument to the command should be added to working memory. If the traffic-light-red rule fires, rules that have the fact (stop) as a part of its left hand side will be activated as long as all other patterns in the left hand side also does match other facts. This, execution order of CLIPS, can be roughly broken down into three parts:

1. The rule at the top of the agenda fires.
2. The commands specified by the right hand side of the rule that fires are executed.
3. Working memory is updated and rules become deactivated or activated and removed from or added to the agenda depending on the changes made to working memory and the left hand side of the rules.

These three steps are then repeated until a predetermined number of rules are fired or the agenda becomes empty.

Agenda

The agenda defines in which order the rules that are currently activated should fire. The order of the rules is determined by two mechanisms, the conflict resolution strategy and each rule’s salience. CLIPS provides many different conflict resolution strategies, among them depth and breadth first and random. The strategy used in FAST is LEX (lexicographic ordering) which states that among rules of equal salience the rule that matches to the most recent facts or objects will fire first. This has the same functionality as the OPS5 strategy of the same name. Picking LEX among the available strategies is important because it enables the AI engine to be reactive; the most recent fact gets priority. Another good side effect of LEX is that if two rules have the same recency but on of the rules contains more patterns on the left hand side then the more complex rule gets priority. This is preferable since more complex rules define more specific behaviour and more specific behaviour will lead to more realistic behaviour. The salience of a rule is defined by a number between -100 and 100, with 0 being the default salience attributed to facts that don’t have a salience defined. The rule with the highest salience gets priority in the agenda, regardless of conflict resolution strategy. If two rules have the same recency, as defined by the LEX strategy and the same salience, the rules are placed arbitrarily, but not randomly, into the agenda (Culbert et. al, 2006).

CLIPS vs. Soar

One of the main differences of Soar and CLIPS is in the way rules are executed, more specifically how the inference engine reasons. In Soar, all rules that match the current state fire as if in parallel and then working memory is updated with the actions produced by these rules firing (Laird, 2000). However, in CLIPS rules that are currently activated fire in the order defined by the agenda and updates working memory after each single execution. This way rules that are scheduled to fire may become out of date and be removed from the agenda as rules that fire may alter working memory such as those rules become obsolete. Another difference between Soar and CLIPS is that in Soar, operators need to be organized in a hierarchical way. This is not a requirement in CLIPS but it is of course possible to organize rules in a hierarchical way and it greatly helps organize the knowledge.

These differences need to be taken into account, but it is still possible to use CLIPS in a manner similar as to how you would use Soar.

FAST

This paper focuses on the part of FAST which is called the FAST Foundation which aims to use an ontology to represent NPC controlled soldier knowledge in a small group that can be used in a general squad based FPS. The FAST Foundation’s goals were to have:
• Static game domain knowledge.
• Dynamic game knowledge.
• Player strategies, plans, intentions, tactics and goals.
• Player behaviour.
• NPC environmental sensing.

The first part of the work is to break down and capture knowledge so that it can be represented as rules. What should an NPC do when he is faced with a particular situation? This is an important part of the solution since it dictates the behaviour of an NPC. The final result will depend on the quality of knowledge acquired in this initial phase.

The next part is to design the rules to incorporate this knowledge, this is generally the most time consuming part of the work (Laird and van Lent, 1999). Finally the expert system must be integrated with the target application, which is outside of this reports scope, but is presented in the FAST Extension report.

**NPC Tactics**

Before any work can begin on the real knowledge, the rules, the first step is to sit down and to actually decide how you want the NPCs to behave, in other words, outline their tactics. Since the project was very limited in scope this part of the work was not as big a part as it could be in case of a project with a broader scope. The knowledge is usually collected by reading field manuals and interviewing Subject Matter Experts (SME) (Wray, 2004) if you are designing a knowledge-base that is designed to mimic human behaviour that follows strict guidelines. However, the case of the computer game is quite unique in that you can hardly state that there is a predetermined perfect way for an NPC to behave, it’s all up to the game designer’s vision. Because of that it was decided to let the rule of “Hollywood” determine what the behaviour would be like.

“Hollywood” is a term used internally at DICE, and presumably at other game developers, to describe the kind of behaviour you would expect from a big action movie produced by some of the big Hollywood film studios. After all, one could assume that most western people’s perception of how soldiers behave in war is based on big budget American war movies and not on actual behaviour in high stress situations like war. This will hopefully ensure that the behaviour is considered “cool” by players, which is an important design goal. This design philosophy was used throughout the FAST Foundation project and no SME was consulted, except for the input of employees at DICE.

With this simple rule in mind the work on how the NPCs should behave began. The NPCs should behave in a cool way and also position themselves so to give the impression that they are providing support to the player. From this guideline the first simple behaviour was decided upon, the NPC is a part of a team possibly lead by the player and the team work together by providing each other support and this support can most effectively be provided if every member of the unit stays in close proximity to each other. Team member proximity is therefore the first and arguably the most important behaviour to address in the design. The object of the project was to demonstrate team behaviour and having members close easily portrays this.

Keeping the NPCs close however is not enough to portray intelligent behaviour. The player and NPCs live in an environment and this environment should obviously be considered when making decisions. The environment could present many different types of objects or features but to simplify the work they would all be handled the same, as points in a space. The NPCs would be aware of these points, coordinates in a 2D space really, and react to them depending on where they are and where the player is. The first behaviour was to stay close to the player so the second behaviour should be to use these objects or features when close to the player.
Some of these objects or features could possibly also be used in an intelligent way, so a third behaviour could be to use objects intelligently when possible.

These three behaviours are discussed in the following paragraphs more in depth along with details to how they were implemented.

**Positioning**

The prototype that was developed for the FAST project was a simple 2D world rendered in a 3D engine. Walls were represented as solid lines and doors as solid lines with a different colour compared to the walls. In this world four entities existed; one player controlled sphere and three computer controlled bots. It was decided that three different environment features would be represented; corners, doors and windows. The NPCs would be aware of where corners were, where the centre of a door was and finally where windows were. With just this information and the location of the player and other NPCs the bot would be able to place himself in the environment to provide sufficient support to the player and not appear artificial. A simple use case was developed to incorporate all the environmental features that were identified. The room has a rectangular shape (see Figure 2) and has two doors, one to the east and one to the west. In the south east part of the room there is a window. Also, inside the room are two big square columns that divide the room into junctions and corridors. This layout presents many good situations for positioning well.

![Figure 2: The use case room](image)

The default position for an NPC was decided to be somewhere around the player, as earlier discussed. The group consisting of the player and the three NPCs should be considered a well trained squad so keeping the squad together is a reasonable goal. This, in practice, resulted in three separate but closely related behaviours. Below a series of rules will be presented, these are not complete rules but have been simplified so to be more readable but at the same time convey the core the knowledge stored in each.

The first behaviour was called **stay-close-to-player-when-idle**; this behaviour basically dictated that when an NPC was standing still, having no current goal in mind, he should stay very close to the player if the player is moving. So, if the player moved far enough away (currently set at 2 game distance units) the NPCs would try to follow the player around and pick a random position around the player’s current position.
(defrule stay-close-to-player-when-idle
    (Player moving)
    (NPC idle)
    ((distance NPC Player) > 2)
=>
    (move-to NPC Player)
)

“distance” and “move-to” are utility functions that among other things could help make rules more readable. You could, for example, on the right hand side of the rule (the parts below the =>) set the goal of the NPC to move to the Player, but here a function was used instead that should be read as “NPC move-to Player”. The same goes for distance, it returns a number of the distance between the NPC and the Player, which could have been calculated in the rule but for readability is encapsulated in its own function.

Behaviour two, stay-close-to-player-when-far-away, is very similar to the last behaviour but with the significant difference that this only applied to a bot which had a current goal, maybe to guard a particular spot, but that was a certain amount of units away from the player (currently at 4 units). This behaviour would assure that even though a bot was guarding a door, it would leave the door and try to place itself closer to the player if the player left the area where the bot was currently positioned.

(defrule stay-close-to-player-when-far-away
    (NPC ~idle)
    ((distance NPC Player) > 4)
=>
    (move-to NPC Player)
)

Notice that we now added a “~” character in front of idle. This acts as a logical NOT. So the NPC can be in any state except “idle” for that pattern to match.

The third and final player positioning behaviour was the stay-close-to-player-when-far-away-idle behaviour. This behaviour is almost equivalent to the first behaviour with the added difference that the player does not have to be moving for this behaviour to be activated.

(defrule stay-close-to-player-when-far-away-idle
    (NPC idle)
    ((distance NPC Player) > 2)
=>
    (move-to NPC Player)
)

Environment Features

Since the main focus of the project was to simulate behaviour and movement in close quarters the behaviours mentioned in the previous section aren’t the primary source of interest, the other features like doors and corners are more so. First you need to decide what you want the NPCs to do when they have specific features available to them in the immediate surrounding. It was decided that because the demo environment wouldn’t lend itself to detailed animations it would be sufficient for the NPCs to just move towards a feature and place itself at it.
Corners constituted the majority of the features available in the demo environment. When a bot was within a certain distance of a corner it would be made aware of this and given the possibility of moving towards it and when there stand guard at it, giving the impression that it was using the corner to partially cover itself.

With doors and windows the behaviour was similar. If an NPC is close to a door or window and decides to move to it then it will move towards the closest side of the door. If the bot would have moved to the opposite side of the feature or if it would have moved to the centre of the feature it would have given the impression that it was not being careful. Moving towards the closest side has two benefits, first it appears to be using the wall as cover for onlookers from the other side but it will also not, in the case of doors, block it completely, giving the player the option to move through the door.

The behaviours related to corners, doors and windows shared several patterns so they were somewhat combined. This combined behaviour is called find-avail-feature. It was decided that the closest idle NPC would always be the one to stand guard at a particular feature. This will avoid strange behaviours like the bot furthest away moving past his team mates to stand guard at a corner or door, that behaviour might be good in some instances, but in the narrow scope this project worked under it never made sense to have someone else than the closest idle NPC move to a feature close by. Furthermore only features that are close to the player’s current position is of interest as this will result in a team more closely together. When an NPC decides to move to a feature that feature becomes taken by that NPC and no other NPC can then move to it with the intention to guard it. This booking lasts for as long as the bot stays at the feature, as soon as it leaves it another NPC can book it and guard it. This was done so several bots wouldn’t guard the same corner when other corners are available nearby.

```
(defformula find-avail-feature
  (NPC idle)
  ((distance feature Player) < 2)
  (closest NPC feature)
  =>
  (move-to NPC feature)
)
```

Which features that are available in the prototype and exactly what to do with them is outside of this paper’s scope and can be found in the sister project FAST Extension.

One final behaviour has not been discussed, dont-stop-in-door. This behaviour was added so that NPCs wouldn’t decide to stop in the middle of the door and therefore block a potential exit. NPCs who are guarding a door will never stop inside of the door but NPCs that are following the player and currently has no feature to guard could potentially decide to move to the door and stop inside it and this simple behaviour stops this from ever happening.

## Implementing Anticipation

Adding anticipation to NPC behaviour has been suggested by Laird (2000). The proposed solution does however assume that NPCs have such complete behaviour that they know about their environment to the extent that they can navigate it themselves. In the case of the FASTbots this was not the case. Their behaviours only specified where to go but not how to get there leaving out a very important aspect, path finding. The FAST Foundation only provided the interface with data about where it wanted the NPC entities to go and the interface forwarded this information to the simple game engine described in FAST Extension. The engine then moved the graphical representations of the NPCs and sent the new location of the NPCs back through the interface to the expert system. The lack of internal path finding made it hard to build an anticipation mechanism that worked as intended. On the other side the strict scope of the project
would not provide with too many interesting opportunities for strong anticipation that would actually be noticeable, except for maybe behaviours concerning doors.

Therefore a special very simple behaviour was created to give the NPCs an added behaviour while guarding the door. The behaviour, `go-through-horizontal-door-if-predicted`, would fire if the player was moving towards the door which an NPC was guarding. When fired the NPC would move through the door before the player reaches it so that it would give the impression that it was scouting the area on the other side of the door before the player reaches it while also staying out of the way so that the player can move through the door whenever he wants to.

```lisp
(deffn go-through-horizontal-door-if-predicted
  (NPC door)
  (predicted Player door)
  (same-side NPC Player)
  =>
  (move-through-door NPC door)
)
```

This rule contains two new utility functions, “predicted” and “same-side”. The first one, “predicted” returns true if it is predicted that the player is moving towards the door. It basically looks to see if the player is closer to the door now than it was just a moment ago and if the Player is close to the door. The second function “same-side”, returns true if the player and the NPC is on the same side of the door.

If a proper implementation of anticipation, as suggested by Laird, had been incorporated into the project a much higher standard of knowledge would have to be added. Anticipation that gives the NPCs enough reasoning ability to predict where a player is headed before the player has even started heading there relies heavily on the NPCs to have behaviours that are very similar to a player’s behaviour. One possible solution would be to have a separate knowledge-base to use for anticipation. The drawback is obviously that it increases the workload and will not be as general as a solution that relies on a complete NPC behaviour. Building a separate knowledge-base won’t solve the biggest problem though, the problem that arises because there are probably as many playing styles as there are players. Consider the following example; a player enters a room with two doors that are located at an equal distance from the player. Some players might choose to go to one particular door a high percentage of the time while others might choose to approach the other. Finally, some players might prefer to let any companions they have in the game approach the doors first. This obviously requires the AI engine to anticipate differently depending on what a particular player’s tendencies are, a general approach to anticipation won’t be enough.

However, although the behaviours implemented in this project are simple they provided a surprising level of perceived anticipation which speaks to the strength of the methods used.
Results

The project suffered from time constraints but was ultimately able to provide a proof of concept that future work can evolve from.

Together with the FAST Extension project a prototype utilizing the architecture discussed in this report was developed. The result of this prototype is outside of this report’s scope and can be found in the FAST Extension paper but a short summary is presented here.

The FAST Foundation project combined up to date AI research into a viable design model that is capable of simulating basic human decision making and that has the potential to do much more complex decision making. The suggested design allows the AI engine to have the desirable characteristics of reactive, context specific, flexible, realistic and easy to develop. As a part of the project a knowledge-base was developed that fulfils all of these goals except flexibility, but flexibility can easily be added with more behaviours for similar situations. The project also suggests a way of implementing anticipation into the suggested design and also discusses how a much more complicated approach could be made to continue work on different anticipation behaviours.

The developed prototype showed good results with the intended behaviours. Using the use case room depicted in Figure 2 modelled in a 3D world three NPC characters were placed in the world with a player entity. With the use of a mouse the user can click on a point in the environment and the player entity will start to move towards this point, taking the shortest path.

At first the player and NPCs are placed outside the east door of the use case room. Since the door is close by an NPC immediately approaches the door and positions itself there while the remaining NPCs stand guard around the player.

If the player is moved into the room the three NPCs will move with the player and take position in close proximity to the player, and at an environment feature if any is close by. This behaviour certainly looked “cool” to the author so one of the design goals was fulfilled.

Also, if an NPC is positioned at a door it will move through the door if the player is approaching from the same side of the door as the NPC, further giving the impression of cooperation.

The work produced can be used to compare and evaluate the suggested design with other ways of solving similar problems. Anticipation can not be easily implemented in a finite state machine based AI framework and that makes the AI engine approach suggested in this work interesting for games development. The description of how anticipation, or player prediction, could be implemented in an expert system driven design and the ease of adding it to such a system is the key takeaway from FAST Foundation.
Conclusions

This project showed that it is relatively easy to develop simple NPC behaviour that is easily extensible and easy to maintain. But is that enough to convince game companies that this approach is better than the ones they use today?

This project provided lots of opportunities to compare the common way of producing AI in games, as described in earlier chapters, to techniques that so far have only been used in the academic world on a research basis. One natural question therefore is; does it provide improvements?

Expert system vs. Finite State Machine

One interesting comparison is whether developing NPC behaviour with the approach suggested by FAST Foundation provides any benefits with regards to development and implementation compared to the way AI behaviour is traditionally developed. The short answer is no. But this conclusion is drawn after developing behaviour that could easily be developed with the object oriented FSM approach commonly used in modern games like Battlefield: Bad Company.

None of the features implemented in the project is very difficult to implement. The positioning relies on ordinary comparisons of positions of the bots and of the player, hardly difficult to do. The feature selection also would be easy to develop using common methods.

What would be hard to implement using a standard FSM would be anticipation. As an example, it is currently impossible to add anticipation to the AI design implemented in Battlefield: Bad Company (Hedberg, 2007). If the AI engine is properly implemented the addition of anticipation is rather straightforward.

Another drawback is the overhead created if development were to switch from an object oriented design to an expert system approach. Developers would have to learn a new way of programming and that would result in a cost for companies since they would have to educate its employees first. It is possible that this could happen, but a company would need to believe the new way of working would provide significant benefits for it to take the plunge and invest in new knowledge.

Anticipation

A working NPC ability to anticipate player intentions would probably be the strongest reason to move away from a basic object oriented FSM design towards the approach suggested in this work. Anticipation in current generation games are more of the script type where a certain situation will alert an NPC to a certain thing that could appear like some sort of anticipation is done when in fact is pre-programmed. A general approach would be much more flexible and could potentially provide anticipation abilities in every situation it could be successfully applied and would add to the perceived intelligence of computer controlled bots.

However, it was shown in this work that simple forms of anticipation can easily be added with just one or a few simple rules in the knowledge base of the expert system. An NPC currently standing at a door and blocking the way of the player can easily sense that the player is approaching and therefore reason that it is standing in the way of the player and move away or to scout ahead on the other side of the door. Expert systems provide a powerful way of easily adding such behaviour.
Summary

Artificial Intelligence in modern computer games has taken a back seat to computer graphics and physics simulations at least from a developer point of view, despite the fact that video games are some of the most suitable applications for various AI techniques.

EA Digital Illusions Creative Entertainment AB was interested in supervising an AI project and after discussing various possible problems it was decided that simulating small combat units in tight areas was the most interesting project to pursue. The FAST project was born, Friendly Automated Support Team. However, such a project would be too big for one single person so it was divided into two parts, FAST Foundation, the part which this report details, which is to handle the low level decision making of members of a FAST team. The other part was the FAST Extension which integrates FAST Foundation with a 3D graphical interface for easy viewing of the actions decided by the FAST Foundation module.

For the entities controlled by FAST to behave as natural as possible it was decided that they would have access to the exact same information as a human player would have such as information about where other entities are located, information about what the environment the entity is in looks like and the kind of tactical information that would be available to a player. If a non player entity has more information than a player would have in the same situation it can seem like it is cheating and it will take away from the perceived intelligence of the entity. The problem the project focused on was movement in narrow spaces such as corridors and small rooms.

There has been some research put into developing computer controlled entities in military simulations, much like those in FAST, that behave in a smart and human way. An institution which has worked extensively on these techniques is the University of Michigan Artificial Intelligence Lab. Tacair-Soar, Soar Quakebot and MOUTBot are all examples of projects developed with the use of an expert system called Soar which is specifically designed to solve such problems as is solved in the various implementations mentioned here. The basic idea is to first specify a number of different behaviour levels, the one at the top of the hierarchy contains the broadest behaviours and the one at the bottom the most specific. For example, the top level might contain a behaviour called “engage enemy” which in turn leads to more specific behaviours to finally arrive at the bottom of the hierarchy where the behaviour might be “fire”. In between there might be behaviours like “locate enemy”, “aim at enemy” and “turn right”.

The part which decides what behaviours the entity should have is called the knowledge-base and the part which decides what decision to do is called the inference engine. Behaviours, or knowledge, in so called expert system are often stored as rules in the form if “if-then”. For example, if a knowledge-base contains knowledge about what wine to drink to a special dish then one rule might be “if red meat and spicy then recommend red wine”.

In the FAST project an expert system called CLIPS was used. This expert system tool is written in C and is easily integrated into other software almost regardless of which programming language this other software is written in. It contains the necessary parts of building a knowledge base such as an inference engine and a knowledge-base building tool.

FAST Foundation was developed with CLIPS and a number of rules that both directly specified entity behaviours and worked as a number of utility methods were implemented. The behaviour rules specified made sure entities stayed close to a player controlled entity, made sure to guard important environmental features and did not stop in the middle of doors so the player couldn’t pass through.

The results were as expected. Entities stayed close to the player and held guard at such important features as doors, windows and corners. They rarely stood in the way of the player and provided support.
References

HEDBERG, M (2007) EA Digital Illusions CE AB, Personal communication