Enhanced NPC Behaviour using Goal Oriented Action Planning

Edmund Long

A dissertation submitted in partial fulfilment of the requirement for the award of the degree of MSc in Computer Game Technology

University of Abertay Dundee

School of Computing and Advanced Technologies

Division of Software Engineering

September 2007

University of Abertay Dundee

Permission to copy

Author: Edmund Long

Title: Enhanced NPC Behaviour using Goal Oriented Action Planning

Qualifications: MSc in Computer Game Technology

Year: 2007

Signature ……………………..

Date …………………………..

Abstract

As the complexity of modern games increases, there has been a movement away from traditional artificial intelligence (AI) technologies as developers search for AI that is more scalable, flexible and provides diverse behaviour in non-player characters (NPCs). Recent innovations in the field of AI Planning have seen the first Goal Oriented Action Planner (GOAP) developed for a commercial game. GOAP has been hailed as the next step in evolving AI in games and can offer several advantages over traditional AI techniques. This dissertation investigated, analysed and compared a traditional AI technique, finite state machines, with Goal Oriented Action Planning when applied in games to discover whether GOAP is indeed an improvement over FSMs and what the relative merits of each system are.

As part of the dissertation, a GOAP and FSM system was developed and placed in a game scenario where experiments were carried out between the two. Comparisons were made on three levels, the first was based on the results of the games i.e. how the systems performed against one another in the Domination, Capture the Flag, Deathmatch and Last Man Standing game modes. The second comparison examined the two systems from a technical perspective and investigated how each system performed from a memory management, CPU usage and efficiency standpoint. The final comparison considers the merits of the two systems with respect to ease of management, flexibility and re-usability.

The results of the experiments clearly highlighted that GOAP is a superior AI system for many reasons however there are certain drawbacks associated with it which may require some consideration before choosing GOAP as a primary AI technique for a commercial game title.

No other previous piece of work has taken a GOAP system and compared it with another AI technique so the findings of this dissertation add to the limited pool of knowledge associated with AI planning in games.

Foreword

Thanks to all family and friends who gave encouragement and support throughout the project. Thanks to Dr. Colin Miller and Dr. Colin Cartwright for their help and advice from the proposal stage all the way through to the submission. Finally, thanks to my girlfriend Mary without whom this project would never have been finished and for keeping me sane in times of difficulty.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1. INTRODUCTION

Video games are constantly evolving. Up until recently there has been a surge of research in the field of graphics in the video game industry with the overall goal being real-time photo realistic graphics. However graphics have now become realistic enough that companies are looking at other ways in evolving games and bringing them to the next level. Apart from the visuals, the other main feature of a game that grabs a player's attention is the gameplay. Gameplay is made up of several different components with one of the most important being artificial intelligence (AI). AI in a game controls the decision making processes of all non-player characters (NPCs) and provides players with challenges they must overcome to progress in the game. With the so-called 'graphics race' effectively finished, the field of AI has become a hot-bed of research with new games using more and more advanced AI than ever before. Therefore this dissertation was motivated by the desire to investigate an exciting new AI technology and compare it with what already exists to determine if this new technique really is an improvement.

Finite state machines (FSMs) have long been a faithful servant to the AI programmer due to their ease to program and robustness. NPC's controlled by FSMs have a number of states and can only be in one of these states at any time in a game. For example, a simplistic FSM could have three states: Roam, Attack and Hide. If the agent is in the Roam state and it sees an enemy it could change to the Attack state. If upon executing the Attack state the NPC discovers that it is low on health, it could change state to Hide and go and find a hiding spot. The AI programmer must decide what states an NPC is going to have and what conditions cause the NPC to move from one state to another. However there are difficulties associated with FSMs that have brought about a desire for cleaner and more flexible technologies.

One new approach to NPC AI is the field of AI planning. AI Planning involves monitoring the current state of an NPC and planning out in advance what actions it is going to perform in order to reach a particular goal. The AI planning process is somewhat similar to the way in which humans reason when looking to satisfy a goal or task. Suppose a human wishes to satisfy their thirst. They know that to satisfy this desire they require a drink. To get a drink the human could first go to the fridge. If the human knows there is nothing to drink in the fridge they can go to the shop and buy a drink. Once the drink has been obtained then the human must open the drink and drink it. So the human would subconsciously build two plans of action, the first to go to the fridge, open the drink and drink it. The second would be to go to the shop, buy the drink, open the drink and drink it. AI Planning takes a similar approach. If an NPC has a goal, 'KillEnemy' for example, it has a collection of actions available that it can execute. The KillEnemy goal can be satisfied by shooting at a target but its weapon first needs to be loaded. If the weapon isn't loaded, the NPC can reload. If there is no more ammunition the NPC must either change weapons or find ammunition somewhere. Finally, once the NPC has a loaded weapon it can fire. So depending on the state of the world and the NPC's goal, the AI planning process chains actions together (e.g. GoToAmmo->Reload->Fire) in the correct combination in order to satisfy their goals.

Goal oriented action planning (GOAP) is a type of AI planning that is beginning to gain more support within the games industry. It is based on the STRIPS (Stanford Research Institute Problem Solver) language which is a planner made up of goals and operators. The planner finds solutions to STRIPS goals by combining actions in sequence and executing them one by one.

The task of this dissertation was to investigate, analyse and develop both a GOAP and a FSM system. These two systems are then to be tested against one another in a game scenario to evaluate the relative merits and drawbacks of the two systems and satisfactorily answer the research question.

Chapter 2. LITERATURE REVIEW

2.1 AI Planning and GOAP

AI planning is a field of classical AI that has been around for quite some time and has been extensively researched in academia.

GOAP is based on a modification of the STRIPS language which is defined using goals and operators. Dr. Bridge outlines the basics of the STRIPS language and some of the issues faced by AI planners in the field of classical AI planning in his lecture slides (Bridge, 2007). STRIPS goals describe some desired state of the world to reach, and operators are defined in terms of preconditions and effects. An operator may only execute if all of its preconditions are met, and each operator changes the state of the world in some way through its effects which are carried out by the add and delete lists. A classical AI planning problem known as the Sussman Anomaly is also discussed along with how it can be overcome using partial-order planners or total-order planners.

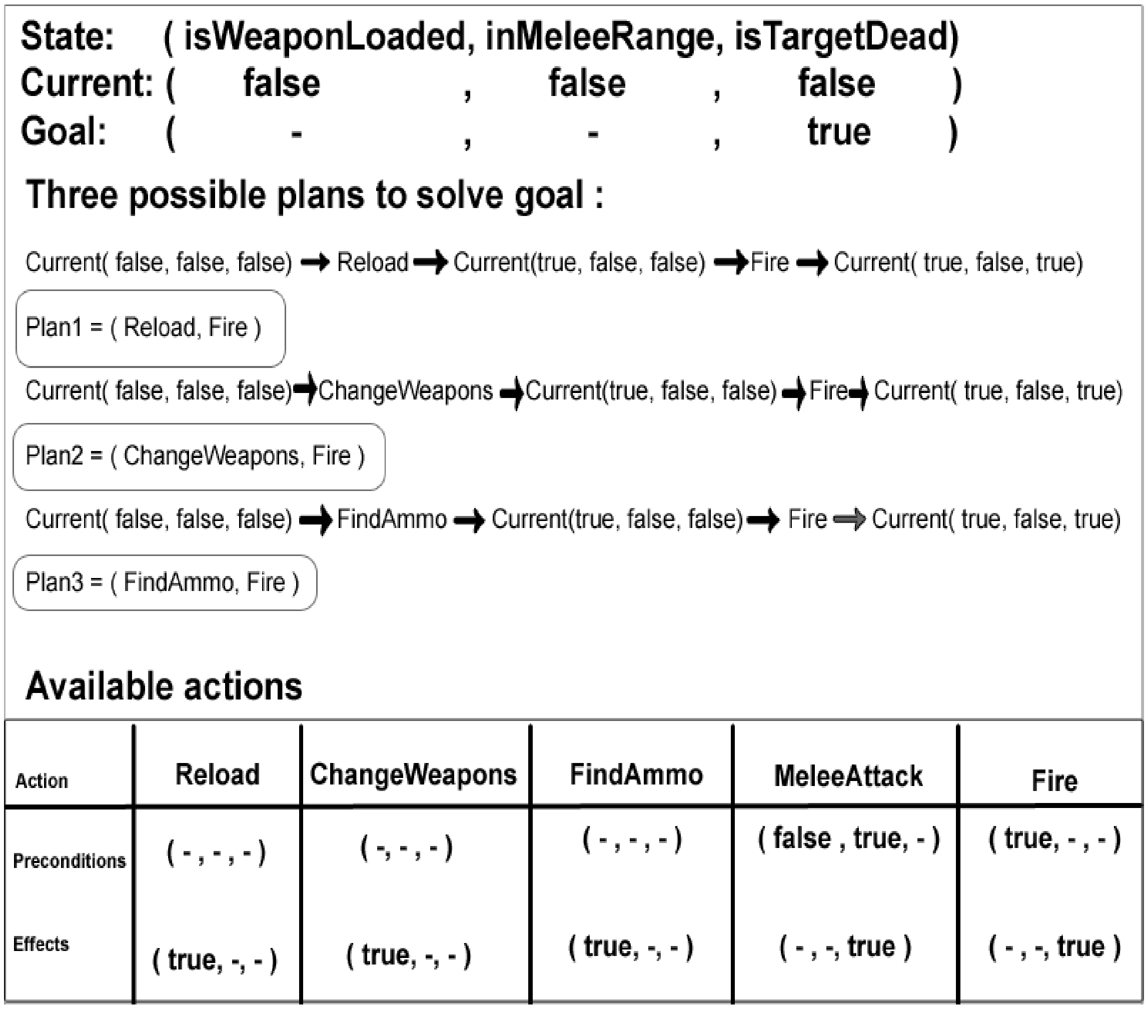
A game produced by Monolith productions, First Encounter Assault Recon or F.E.A.R., is the first AAA commercial title known to use a real-time planner system in games and won several awards due its ground-breaking AI. The lead architect of the AI system, Jeff Orkin, has written several articles that describe the agent and system architecture used in F.E.A.R.. His work and publications provided the inspiration for this dissertation as GOAP is such a new AI technology and the challenge of creating a state-of-the-art system whilst discovering its possible benefits over existing traditional technologies was quite appealing from the outset.

A presentation given by Orkin at the Game Developers Conference 2006 (Orkin, 2006) provoked the first interest in the field of AI planning in games. The difficulty in managing the complexity of FSMs was cited as the primary reason as to why GOAP was chosen for F.E.A.R.. Orkin specifies exactly how the system used in the game was based on the STRIPS language and clarifies the modifications that were employed in order to make it more applicable to games. The STRIPS operators are renamed to become actions in a GOAP system. The primary differences were assigning costs to these actions so that they can be used alongside the A* algorithm, the removal of add and delete lists, procedural preconditions and procedural effects. The procedural preconditions and effects are added to actions so that they can query and have an effect on the virtual world outside of the planner. The actual planning process developed for F.E.A.R. is detailed and this includes a brief explanation as to how a generic A* engine can be used for both pathfinding and planning by changing A* nodes to be GOAP world states and A* edges to GOAP actions.

Orkin continues to explain that each NPC has a collection of symbols (called world state symbols) that represent the agent's current perception of the world and set of goals and actions. Each symbol represents information about the world, e.g. if an agent is thirsty it could have an 'isThirsty' symbol which would be a Boolean value, true or false. The collection of world state symbols is known as a world state. Each goal has a world state which represents the satisfaction criteria for that goal. An action's precondition is also a world state that the agent's current world state must satisfy before an action can be triggered. An action's effect is also a collection of world state symbols and they are applied to the NPC's current world state upon execution of the action. Based on the agent's current perception of the world, the NPC decides which of its goals is the most relevant. If this new chosen goal is different to its current goal, the GOAP planner system attempts to find a combination of available actions that can take the agent from its current world state to satisfy the goal world state. The A* algorithm is used to plan the correct sequence of actions that form an NPC's plan. The NPC then proceeds to step through and execute the plan, one action at a time, until there are no more steps left.

Based on what was learnt from this article, a simple planning problem along with possible solutions is illustrated in figure 1 to demonstrate exactly how AI planning works as defined by Orkin. The goal chosen for this example is the KillEnemy goal which is satisfied if a single world state symbol, isTargetDead, is set to true in the agent's current world state. There are five actions available to the agent, Reload, ChangeWeapon, FindAmmo, MeleeAttack and Fire. Each action has preconditions and effects which can be set or unset (Reload has no world state symbols set for its precondition). The planning system can produce three potential plans and either one of them solves the goal satisfaction condition. At first, it may appear that there is no difference between the three actions, Reload, ChangeWeapon and GoToAmmo from the viewpoint of the planner but the difference arises in each action's costs, context preconditions and functionality when activated. The action's associated cost is used to guide an A* search, the context preconditions validate certain action specific conditions before an action can be chosen by the planner and each action has differing effects on the virtual world when actually selected for activation after planning has completed.

When examining the GoToAmmo action, if the agent has any ammunition then the context preconditions of the GoToAmmo would fail and plan3 wouldn't be created. When the planner is inspecting the Reload action, if there is no ammunition for the current weapon its context preconditions fail, thus making plan1 impossible. If the current weapon has no ammunition and there exists another weapon with ammunition, then plan2 would be built. If the inMeleeRange world state symbol was set to true in the current state of the agent then the planner would simply perform a melee attack.



State: (isWeaponLoaded, inMeleeRange, isTargetDead)
Current: (false , false , false)
Goal: (- , - , true)

Three possible plans to solve goal :

Current(false, false, false) $\rightarrow$ Reload $\rightarrow$ Current(true, false, false) $\rightarrow$ Fire $\rightarrow$ Current(true, false, true)

Plan1 = (Reload, Fire)

Current(false, false, false) $\rightarrow$ ChangeWeapons $\rightarrow$ Current(true, false, false) $\rightarrow$ Fire $\rightarrow$ Current(true, false, true)

Plan2 = (ChangeWeapons, Fire)

Current(false, false, false) $\rightarrow$ FindAmmo $\rightarrow$ Current(true, false, false) $\rightarrow$ Fire $\Rightarrow$ Current(true, false, true)

Plan3 = (FindAmmo, Fire)

Available actions

Action	Reload	ChangeWeapons	FindAmmo	MeleeAttack	Fire
Preconditions	(- , - , -)	(-, - , -)	(- , - , -)	(false , true, -)	(true, - , -)
Effects	(true, -, -)	(true, -, -)	(true, -, -)	(- , -, true)	(- , -, true)

Figure 1 A Sample planning problem

Dynamic re-planning is another feature of the GOAP system as highlighted by Orkin. The planner constantly monitors the agent's current plan checking if it is still valid. If invalidated, it is possible for the GOAP system to either plan towards the same goal selecting different actions or it can select a new goal and plan towards that instead. This dynamic replanning that can bring about new and sometimes unexpected behaviour that is highlighted as one of the main features of the GOAP system. For example, if an agent is attacking an enemy and it discovers its current weapon is out of ammunition, instead of checking the relevancy of each goal once more, the same goal is planned towards and another way of satisfying it is discovered by changing weapon, going searching for ammunition or performing a melee attack.

FSMs and GOAP are briefly analysed but only on the basis of how the decision making process of each is carried out; Orkin states that FSMs are procedural while planning is declarative. Three general benefits a planning system offers to AI programmers are outlined which are the decoupling of goals and actions, dynamic problem solving and layering of behaviours.

Finally, Orkin describes how squad behaviour was integrated alongside GOAP in F.E.A.R.. Squad behaviour relies for the most part on the individual GOAP system of the agent. Orders, which simply prioritise certain goals, are issued to the individual GOAP agents and they can decide whether to obey these orders or not. If another goal has higher priority than the one that satisfies the squad command, then it gets chosen and the squad command doesn't get carried out. Complex squad behaviour actually arose due to the dynamic interplay between the squad decision making system and the individual decision making. An ad-hoc approach to squad behaviour was implemented in F.E.A.R. but Hierarchical Task Networks (HTNs) are suggested for squad planning in the future.

Munóz-Avila's paper in which HTNs are explored as a means of encoding strategic game AI (Munóz-Avila and Lee-Urban, 2005) was quite influential on what type of game was chosen to be created for the dissertation and what tests would be run between the FSMs and GOAP systems. The article outlines a project undertaken that uses HTNs to extend GOAP to strategically control agents or bots in a game called Unreal Tournament 2005 (UT2005). The project 'modded' UT2005 using Javabots which involved taking the existing commercial game code and modifying it to create new custom AI behaviour. HTNs control teams of bots in a series of 'Domination' games in UT2005. Domination games involve teams of bots fighting over two strategic locations on a map and only one team can control a location at a time. If all the strategic locations or 'Domination points' are controlled by a single team for a certain period of time then the team scores a point. The results of the project highlighted that the HTN out-performed the non-HTN team and that they are well-suited to encoding strategies to coordinate teams of bots in first-person shooter games.

Another article published by Orkin discusses design decisions taken for the GOAP system and the AI agent created for F.E.A.R. (Orkin, 2004). The planning process is once more dealt with, but in a little bit more detail. Along with the prioritisation of goals, actions can also be prioritised to resolve cases where actions have the same effects. There exists no explicit mapping between goals and actions, the planner finds the valid sequence of actions to solve a goal at run-time and there is no link between goals and actions.

When looking for a solution to a goal, the planner must examine which world state symbols are unsatisfied between the agent's current world state and the goal world state and look for actions that have effects that bring the two states closer to one another. It is possible that there can be a combinatorial explosion when searching for a valid sequence of actions for a goal but Orkin explains how the F.E.A.R. system overcame this problem by using both symbolic and non-symbolic preconditions, hashing actions by their effects and using heuristics to guide the A* search.

The issue of symbolically representing the virtual world to the agent is also addressed in this article. One of the difficulties involved with implementing a planner based on STRIPS is that STRIPS functions using Boolean symbols. This may be inadequate for a game scenario as not everything in a game world can be represented by just true of false. Before an action can be chosen by the planner, its symbolic preconditions must first be satisfied (e.g. the weaponLoaded world state symbol must be set to true in the agent's current world state before the Attack action can be chosen by the planner). However it is often the case that extra checks may be required that can't be performed by the symbolic preconditions. The F.E.A.R. system used context preconditions to perform these checks where each action can query the subsystems and run validity checks to see if the action can be eliminated from the planning process or not. There are two benefits in using context preconditions. It improves efficiency by reducing the number of potential candidates when searching for viable actions during planning and it extends the functionality of the actions beyond simple Boolean world state symbols.

The hashing of actions by their effects involves storing a list of all actions that have an effect for a specific world state symbol in a lookup table. If the planner needs to find actions that will solve a world state symbol, it can look at all the actions for that symbol in the hash table without having to build a new list every time.

The heuristic value used in the A* search is the number of different world state symbols between the agent's current world state and goal world state. This means that actions which bring the current and goal states further away from each other (i.e. more symbols different between the current and goal states than before action execution) are given a higher heuristic value and thus receive a higher 'f' value for their insertion to the open list during the A* search.

The International Game Developers Association (IGDA) has a working group on Goal Oriented Action Planning whose task is to create an interface standard specification for real-time GOAP in games (Long et al., 2003), (Nareyek et al., 2005).The working group is still in its infancy and more work needs to be completed before the group can recommend a full standard specification to industry however. In the IGDA's AIISC 2004 report (Orkin et al., 2004), a requirements specification for GOAP goals and actions is detailed. Although quite high-level, this specification indicates what goals and actions should be able to do and the primary functionality that should be included. While there isn't a finished interface available as of yet from the group, this specification provides a solid base to start off with.

Orkin's article in the AI Game Programming Wisdom book (Orkin, 2002) focuses on the design considerations when creating GOAP actions and goals and builds upon the specification from the GOAP working group. It states that goals have a certain number of properties, only one goal can be active at any time, each has a relevancy which is constantly updated, knows when it is satisfied and doesn't contain a plan – only conditions that must be met. A plan is also defined as being "any valid sequence of actions that move the agent from some starting state to a state that satisfies the goal". There can be multiple plans to solve a goal and so the planner needs to be directed by giving costs to actions. Each GOAP agent has a reference to a planner which searches the set of actions for the correct sequence that takes the agent from the start state to the goal state.

A* is not only used to search for the correct sequence of actions to satisfy a goal when planning but it is also used to find a path through a virtual world when an agent is navigating. In separate articles, Lester describes the basic theory behind A* pathfinding (Lester, 2007) and Higgins outlines how to create a generic A* machine (Higgins, 2002). The generic A* machine can power A* searches used for navigation and planning. To create a generic A* machine there are four primary components needed: A* node, A* storage, A* goal and A* map. The modifications needed to get each of these components to function for different purposes are also explained however GOAP isn't specifically mentioned. Orkin recommends using a generic A* machine in his paper (Orkin, 2005) but with different A* goal, A* map and A* node structures for both the navigation and planning systems. The benefit of this system is that it facilitates the re-use of the A* engine code within the game.

2.2 Agent Architecture

A FSM or GOAP system is just a decision making module, it determines what actions should be carried out and when. However this decision making module is only one part of the agent that resides within the virtual game world and it needs to be able to interact with some, if not all, of the other components of an agent. One of the major tasks in creating any AI system is developing a robust agent architecture to plug the AI system into. The architectures explored were again heavily influenced by Orkin's work with F.E.A.R. as it is the only real-time planner known to have been developed for games that has any documentation about its architecture publicly available at present.

The GOAP system agent architecture considerations are discussed at length in another of Orkin's articles (Orkin, 2005). The F.E.A.R. agent architecture employs a modified version of the cognitive architecture as described by the Synthetic Characters Group or C4 from MIT (Burke et al., 2001). The C4 architecture consists of six primary components, the sensory system, a working memory, the action selection system, an internal blackboard, a navigation system and motor system and is represented in figure 2 below.

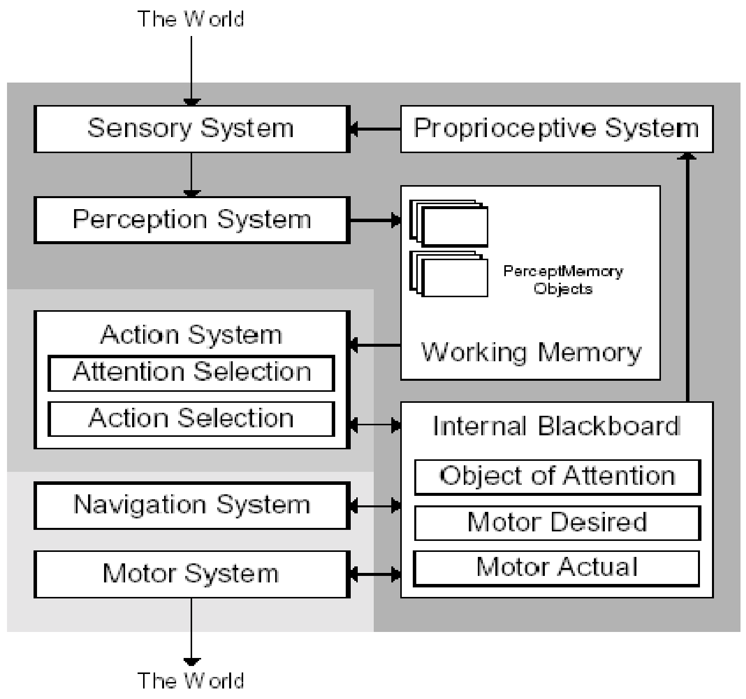


Figure 2 C4 Agent Architecture

The motivation for using the C4 group's architecture is justified by Orkin through outlining some of the performance issues associated with evaluating preconditions of actions which can sometimes require CPU intensive operations such as ray casts, A* pathfinding etc. However, there are some alterations required to the C4 agent architecture to allow for its use alongside a real-time planner. The subsystems are extended to include targeting, animation and weapons combined with the inclusion of the real-time planner instead of the action system used in figure 2. Each agent has a sensory system that monitors the virtual world and records data in the working memory. There can be a number of sensors, some of which are updated every frame while others may only be updated every few seconds. The working memory is described as being a memory location where information about the virtual world is stored for later use. It is implemented in the agent architecture as it can facilitate distributed processing and caching of information which can help reduce the load on the CPU when evaluating some action's context preconditions. Orkin continues to explain the working memory concept in greater depth indicating that the working memory is made up of a collection of working memory facts, each of which have the same general structure. Each fact can have a confidence value associated with it which is a value between zero and one and represents the confidence or certainty the agent has regarding that certain working memory fact. The concept of an internal blackboard is introduced, it is likened to a central scratch-pad that all the subsystems use to communicate with one another.

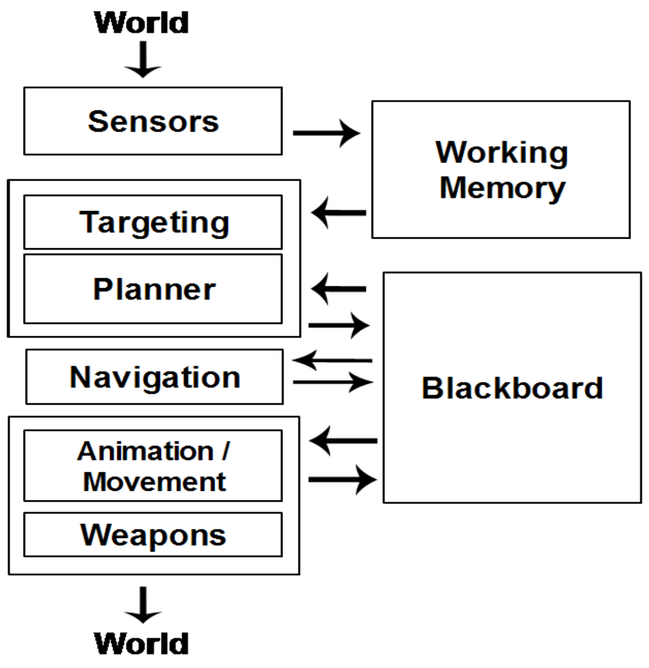


Figure 3 F.E.A.R. Agent Architecture

Further information about the planning process is illustrated and in particular what steps were taken to optimise the A* search for F.E.A.R.. The planner uses the preconditions to force the A* search to only take fruitful branches and again the hashing of actions by their effects is mentioned. Orkin indicates that the action context preconditions work in tandem with the blackboard and working memory so that they don't need to directly access the subsystems. The A* heuristic employed in F.E.A.R. is again described as being the number of unsatisfied world state symbols between the current and goal states.

The considerations involved with developing agents which implement blackboard systems in first-person shooters are analysed in another of Orkin's articles (Orkin, 2003). It details more clearly how to go about designing and creating blackboards for games along with their benefits. There are two types of blackboard, inter-agent and intra-agent. The inter-agent blackboard is used to coordinate between agents (relates to the working memory in figure 3) while the intra-agent blackboard is used by sub-systems to communicate and share information (the blackboard in figure 3). While there is no formal structure to the blackboard set out, Orkin suggests it could be just a collection of getters and setters making it into a central repository of information. The task of the inter-agent blackboard or working memory is to store information about the world as sensed by the sensory manager but also to coordinate agents in teams by storing orders as received from the squad manager.

2.3 Finite State Machines

Some investigation into FSMs revealed that there are several different methods of implementing them in games. Three options are presented by Houlette (Houlette, 2003), the first is to code up the FSM directly in C++, the second is to use a macro-assisted FSM language and the last way is to create a custom FSM scripting language that defines the machine. Creating a custom FSM scripting language involves defining the FSM in an external text file and on being read into the program it is transformed into C++ or bytecode. Houlette continues to explain that this system requires a custom vocabulary along with a native condition and action library. The condition and action library is a vocabulary subset of the scripting language for game-specific actions and conditions. The state transitions must also be encoded in the external scripting file. Houlette warns that defining a scripting language requires a lot of man-power and it can be tricky to design a good scripting language. The macro assisted FSM language is only briefly explained and as there isn't any source code or examples of its use available and combined with little previous experience with macros in C++, made it a risky undertaking. The traditional C++ approach is described as being problematic when coming up with state transitions, occasionally troublesome to debug and can increase compile times, however it is one of the most tried and tested means of creating FSMs in the games industry.

Buckland approaches FSMs by using C++ templates to represent FSM states along with an overall state machine class to control state transitions. Buckland's article (Buckland, 2005) describes a quite neat implementation with sample code of the actual state machine in action provided. The entire state machine itself is relatively straight forward and is contained within the single header file. Each state extends a base C++ template state and has the same general structure.

There have been some criticisms levelled at FSMs. Isla states that with ever increasing game complexity FSMs can become unwieldy and difficult to manage due to the number of states and the transitions between these states (Isla, 2002). Another criticism of FSMs is that they are deterministic. Given an input and if the current state is known, the state transitions can be predicted (Brownlee, 2002). This is often used to players' advantage as they often 'learn' AI behaviour and can predict what actions the AI is going to perform. Brownlee discusses two approaches to bring about non-deterministic behaviour by extending FSMs to use Fuzzy State Machines and by using random numbers. This would help make the system a little less predictable. Just like Isla, Brownlee notes that FSMs can be difficult to manage and maintain as they increase in size.

2.4 Statement of Research Question

The question being posed by this dissertation was to evaluate whether GOAP is a superior AI technology to FSMs. To answer this question, it was necessary to analyse the two systems on several levels of comparison. The first comparison was to develop the two AI systems and pit them against one another in a game scenario i.e. a Domination game and determine which system performs better. This also includes monitoring how each system can incorporate squad tactics, just like the HTN project (Munóz-Avila and Lee-Urban, 2005). The inclusion of squad tactics helps to highlight the flexibility of each system along with how well the AI system can integrate squad behaviour. Technical tests form the basis of the second comparison, i.e. what are the differences in memory usage, frame rate, CPU utilisation etc. between the two systems. The re-usability, flexibility, ease of management and issues faced during development are the final criteria by which the two systems are evaluated.

The advantages provided by GOAP or AI planning in general have been outlined by several authors but none have performed an in-depth comparison between GOAP and another AI technology to prove these claims or come up with any direct comparisons. Orkin stated the benefits of the GOAP system developed for F.E.A.R. without actually comparing with any other system. Munóz-Avila builds upon GOAP by using HTNs to encode team strategies whilst leaving individual UT bot behaviour to be determined by FSMs and pits them against improved UT bots. So although the work undertaken was similar to these other projects, none actually perform any direct comparisons using GOAP. There has been relatively little work involving GOAP in games, especially for research purposes. Hence this dissertation adds to a knowledge gap in the field of AI planning in games and provides a stepping stone for further work in this area of research.

As FSMs have been so prevalent in games and are a known and stable technology, moving away from them to such a new and unproven technology may not be so appealing for games companies. However the findings from this project not only contribute knowledge to the field of AI planning but also look to show that the benefits of GOAP far outweigh the disadvantages and uncertainly surrounding it.

Chapter 3. DESIGN

3.1 The Game Scenario

It was decided from Munóz-Avila's article (Munóz-Avila and Lee-Urban, 2005) to perform the experiments between the two types of AI in a Domination game for this dissertation. This game is an ideal test scenario for the two types of AI as it puts one team against another and the scores and kills from the game can be easily measured and compared. A certain degree of squad tactics are required for a team to be successful so this also enables the evaluation of how well each type of AI can integrate squad behaviour. HTNs were used to control the squad coordination in Munóz-Avila's work however the extra complexity they bring meant that it was decided not to implement them for this project. An ad-hoc approach to squad coordination and tactics was designed in its stead, similar to the approach taken by Orkin in designing the squad behaviour for F.E.A.R.. The squad system prioritises agent goals but each agent uses its own individual GOAP system to make decisions.

Ideally, this project would have modded UT2005 to create the Domination game simulation just like the HTN project. However due to the time constraints and man-power available, this wasn't an option. So instead it was decided to develop a 2D game using Microsoft's DirectX API that mimics the UT2005 Domination game mode as closely as possible. DirectX is quite mature, widely used in industry and minimized any learning curve involved with making the game. It also allowed more focus on the actual AI rather than having to get to grips with a large codebase and complex systems that would have occurred if the modding of UT2005 was undertaken. The option of extending the Domination game to include other game modes was left open in the design of the game.

3.2 Agent Architecture

As the agent architecture described by Orkin and shown in figure 3 is flexible enough to incorporate almost any type of AI system and is known to work with an existing planner, it was decided to model the agent's architecture for this project on this architecture, with a few changes. There are five major components of this project's agent architecture which are the working memory, the blackboard, the subsystem managers, the AI module and the squad manager. The subsystem managers include the target manager, the weapons manager, the navigation manager and finally the sensory manager. The animation manager, which is part of the F.E.A.R. architecture, is left out for this project. The AI module contains references to all the various managers and is charged with setting them up and updating them regularly. The AI module can be either the FSM or GOAP system depending on the AI system chosen. Figure 4 illustrates the flow of information through the architecture for every update and which components communicate with the others.

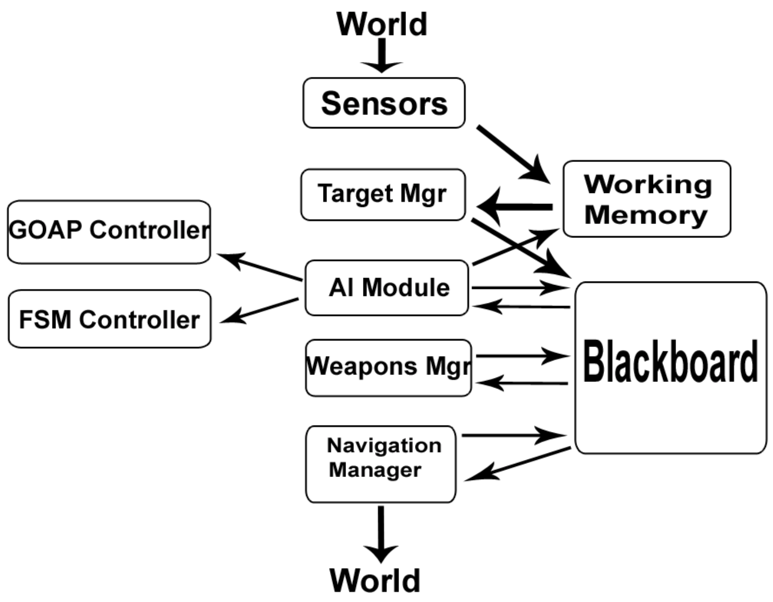


Figure 4 High level agent architecture

The squad manager triggers an update of the entire agent. Just like the F.E.A.R. system, the sensors read in details about the world and store information in the working memory as working memory facts. The target manager reads the working memory, decides what to target and deposits its decision on the blackboard. The AI module updates the GOAP or FSM controller which in turn reads information from the working memory and the blackboard. It is within the GOAP or FSM controllers that the actual AI processes take place. On the back of the information read by the AI module, it makes decisions as to what the agent is going to do and deposits appropriate data on the blackboard or writes to the working memory. The weapon and navigation managers read information from the blackboard and cause effects in the virtual world such as moving around the agent or firing at a target. Finally the agent itself is rendered. The different components of the architecture are explained in detail in the following paragraphs culminating in the description of how the two different AI modules (GOAP and FSM) were designed.

3.3 Managers

The managers within the game control the various subsystems and perform most of the ground work needed for the AI module to carry out its decisions. The sensory manager consists of a number of sensors that record information about the world in working memory for later use. There were five sensors designed in all, two pickup sensors, an enemy sensor, a friendly sensor and a node sensor. The sensors aren't updated every frame but are offset for different time periods depending on the sensor type. Sensors that are critical to the agent, such as the enemy sensor, are updated more regularly than the pickup sensor for example as more up to date information is required for the target manager to make correct decisions.

The target manager decides what the agent's current target is at any time. The target manager examines the working memory and gets the Enemy working memory fact with the highest belief (explained in the next section) and writes to the blackboard what this target is.

The weapons manager controls the updating of the agent's weapons. In every frame the weapons manager queries the blackboard and checks what tasks is it required to carry out such as reloading or changing weapons. The weapons manager also fires at the agent's target if there's a request on the blackboard to do so. Each of the agent's weapons are updated which involves rendering and updating all the weapon's shots. Once finished attacking, the weapons manager updates the attack status on the blackboard.

The navigation manager coordinates the agent's navigation around the virtual world. The A* algorithm is used to find paths as it is an industry standard algorithm for pathfinding and is used for the GOAP planner. The navigation manager is also in charge of finding patrolling nodes, valid attack positions, dodging, finding pickup locations and performing flocking. As the process of navigation using A* it directly linked to the actual GOAP planning, the more involved details of it is left for a later stage. However when updated, the navigation manager checks if its current navigation target is the same as the navigation target on the blackboard, if not then it requests for the new path to be created using A*. It then steps through the sequence of path nodes in the navigation plan one node at a time, moving the agent along until it reaches its final destination. When finished, the navigation manager updates its navigation status on the blackboard.

3.4 Working Memory

When the sensory manager discovers something about the world it deposits a working memory fact in working memory. At a later stage if the AI module is looking for information about the world, it searches all valid working memory facts looking for the best fact which is used to help make a decision or execute an action. The benefit of using the working memory is that it allows for the storage of results (caching) and sensors may not need to be run every frame, they can be delayed (distributed processing).

The working memory is a collection of working memory facts. Each fact was designed to have a fact type, an ID and several other optional values such as belief, owner ID, target ID, position, distance, radius, direction etc. In all there were six different types of facts designed: Enemy, Friendly, PickupInfo, Event, Order and NodeType.

A fact's belief (or confidence) can represent the agent's belief that something exists in the virtual world. For example, if an enemy target is in the agent's field of view then a working memory fact of type Enemy would be created and set to have a belief of 1. If the enemy was obstructed or not seen in some time the belief of the fact would be lower. The target manager chooses the enemy fact with the highest belief as its current target.

The working memory also provides a querying facility whereby a working memory fact can be sent in as a query and used to check if any facts match it. Associated with each fact is a standard template library bitset that indicates which of the working memory fact attributes have been set. When two facts are being compared, the first step is to test each fact's bitset. If this passes then the query's attribute values are tested against each of the existing working memory facts' attribute values. Also designed within the working memory was a facility to count the number of fact types and find facts with the highest belief for a given fact type.

3.5 Blackboard

Each agent in the game has a blackboard which serves as a public scratch-pad for all the various subsystems in the game. As suggested by Orkin (Orkin, 2003), the blackboard was designed to be simply a list of getters and setters for the different subsystems. For example, there is a list of getters and setters for the agent's navigation such as setting a new navigation location, setting the speed of movement and the status of the navigation. There are also getters and setters for targeting, following of agents, weapons, planning and agent's characteristics. Blackboards can benefit an agent's architecture in several ways. It facilitates centralised communication between subsystems thus reducing coupling of the subsystems, see figure 5 for an illustration of this fact. Suppose an agent wishes to move within firing range of a target. Without a blackboard system the navigation manager would need a reference to the target manager and vice versa to know where to move to. When using a blackboard both systems just need a reference to the blackboard which they can read and write to and from.

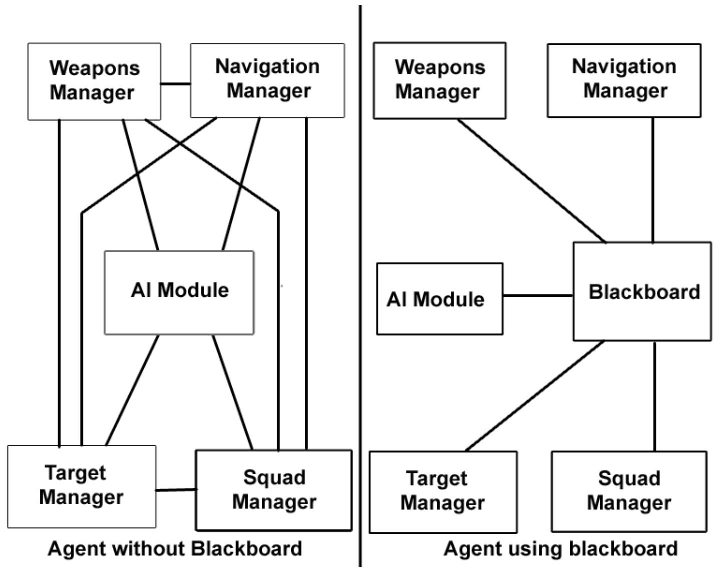


Figure 5 Blackboard comparison

The use of the blackboard becomes advantageous for later projects as systems can be migrated across to new projects without having to re-write much of the system. Without a blackboard system the various subsystems would have references to the other systems scattered within them which would have to be removed before migrating them. Also the blackboard is extremely useful for the squad manager as it can monitor the status of a subsystem or task is by querying a single central location rather than the various subsystems. Orkin also states that blackboards assist developers in that they can provide improvements in performance, maintainability and flexibility and there is less to debug when using them (Orkin, 2003).

3.6 Squad manager

The squad manager designed for this application was simply an abstract class that was implemented at a later stage by the squad managers for the different game modes. Each squad manager is in charge of deciding what squad an agent is placed in and what tasks each of these squads should perform. The agents are first placed in a pool of agents and then a squad manager must decide whether to create squads or not. The primary function of each squad manager is to coordinate these squads so that they aren't all performing the same tasks. When a squad manager wishes to set the squad a task, it issues it a command. Upon activation of a command, agents have working memory Order facts assigned to them. The squad manager monitors the status of the agent's blackboard. Squad coordination is then brought about by creating new agent orders or modifying the squad's task according to the status of the individual agents or according to events in the game.

Both the FSM and GOAP systems share the same squad manager system for each of the game modes as both types of AI attempt to perform the same tasks within the game. The difference arises in how the FSM or GOAP react to the orders that are stored in their working memory. Squad behaviour can be disabled for a team and when this occurs squads are never created and all agents are left in the pool of agents which never receives orders.

3.7 AI Module

The AI module provides a virtual class that the FSM and GOAP controllers implement and extend. Each agent in the game is controlled by an AI module which can be to be either a FSM or GOAP controller. The AI module provides virtual setup, update and cleanup functions that the GOAP and FSM controllers overwrite using polymorphism to create the different behaviour. The setup and updating of the individual AI modules is carried out by the squads which are in turn updated by the overall squad manager which is updated from the main game loop.

See figure 6 for a UML diagram that illustrates the relationships between the different subsystems, squad manager, the AI module and the game itself.

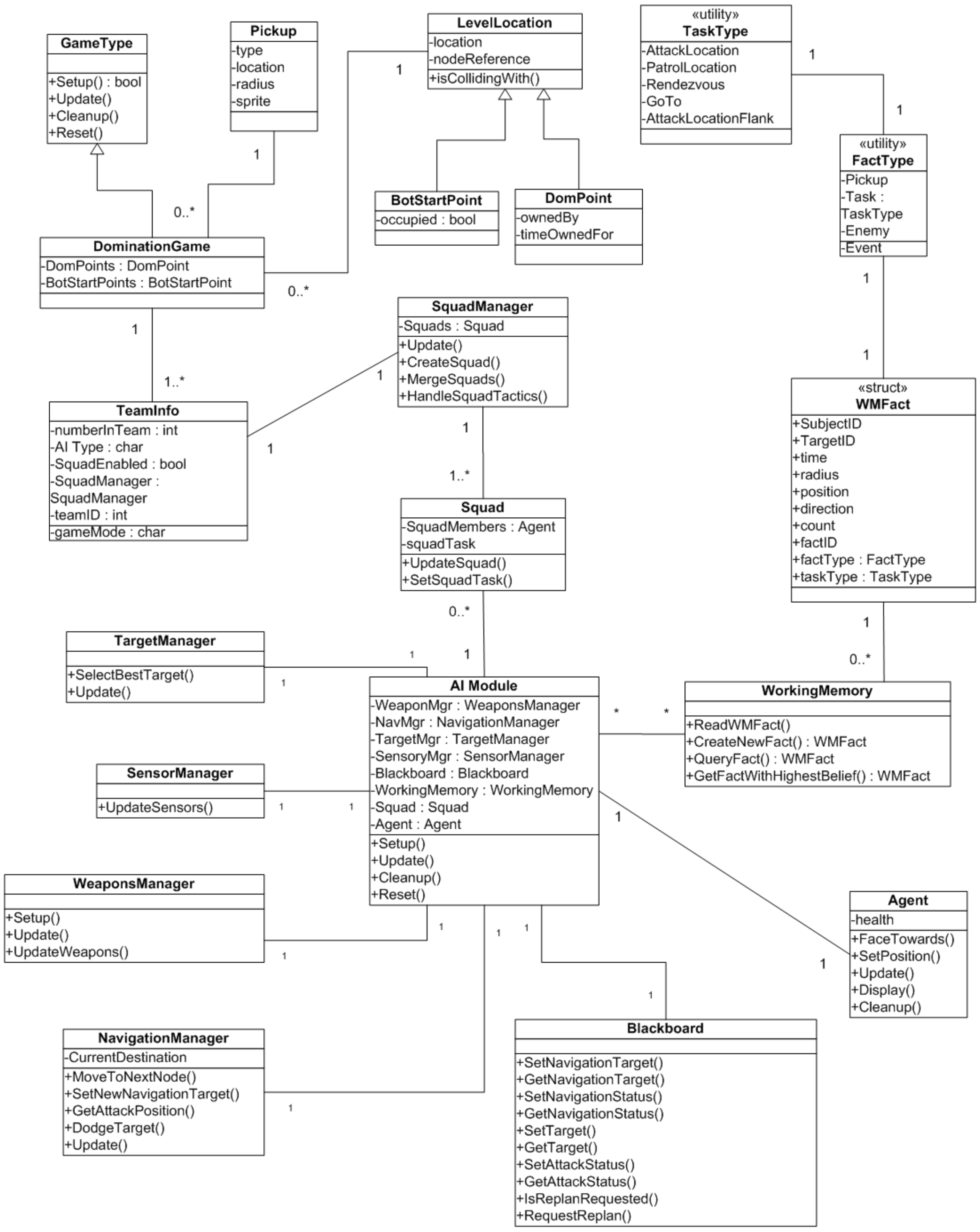


Figure 6 Agent Architecture UML

3.8 GOAP Design

All of Orkin's previous articles relating to GOAP indicate that within a planning system each GOAP agent should maintain a world state which is a collection of world state symbols that relate to the virtual world.

3.8.1 World state

Each agent maintains a world state which is basically a fixed-size array of world state symbols. Following on from Orkin's advice (Orkin, 2004), each world state symbol is a key-value pair and includes a name, type and overloaded operators to allow for testing against other symbols and for setting values of the symbols. Figure 1 illustrates a simple world state for an agent with three Boolean world state symbols, weaponLoaded, isTargetDead and inMeleeRange. It was determined from designing the actions and goals that the GOAP agent required eight core world state symbols for this project: targetIsDead, covering, weaponLoaded, atTargetNode, atDomPointOne, atDomPointTwo, inMeleeRange and inTargetSights.

A world state doesn't require all the symbols to be set. For example, the Reload action in figure 1 has a precondition world state with just the symbol weaponLoaded set to true for its effect, all other symbols are unset as they are not relevant or needed. The world state class provides methods to evaluate one world state against another, find out how many world state symbols are different (i.e. test each symbol against every other symbol regardless if the symbol has been set or not) and check how many symbols are unsatisfied (i.e. how many symbols that are set in each world state are different). This functionality was required for the planning process as different world states are often tested against one another. Also a means of getting, setting and resetting world state symbols was included in the design so that world states could be changed and queried if necessary.

3.8.2 Agent action and goal sets

Different agents in the system can have different capabilities. Each type of agent has its own goal and action set which define what goals and actions the planner can use when planning. Rather than hard-code this into the application it was decided to keep this data external to the C++ code. XML was chosen as the means to represent what actions and goals an agent could have. An external XML file is processed using an open source plugin written in C++ called XPathParser that can read XML files and provides functions to parse them quickly and easily. When an agent is first created, this XML file determines which actions and goals can be allocated to it. A sample XML file for a basic agent is outlined below. The agent has a single goal in its goal set, Idle along with GoToCover, GoToAmmo, GoToHealth and Idle actions available in its action set.

```xml
<?xml version="1.0" ?>
<agent>
<actions>
<action type="Attack">false</action>
<action type="ChangeWeapon">false</action>
<action type="GoTo">false</action>
<action type="Patrol">false</action>
<action type="Idle">true</action>
<action type="Reload">false</action>
<action type="GoToAmmo">true</action>
<action type="GoToCover">true</action>
<action type="GoToHealth">true</action>
<action type="Dodge">false</action>
</actions>
<goals>
<goal type="KillEnemy">false</goal>
<goal type="Dodge">false</goal>
<goal type="Idle">true</goal>
<goal type="Patrol">false</goal>
</goals>
</agent>
```

The benefit of using this system is that if agents with different goal and action sets need to be created then the program just needs to read in the XML file for that agent type. For example, an agent with the XML file defined above could be just a rat-like enemy in the game which can't attack and just wanders aimlessly while another agent type could have more or all of the action and goal types enabled thus giving it different behaviour.

3.8.3 Actions

The design of the structure for the GOAP actions was influenced by the requirements specification document set-out by the working group on GOAP (Orkin et al., 2004) along with other considerations taken from Orkin's suggestions in his various articles. GOAP actions have the same general characteristics according to Orkin (Orkin, 2002) so an abstract action was designed that implements the basic functionality required for all actions. All the subsequent actions created build on-top of this and extend its functionality. For an action to trigger, the preconditions and effects of the action must be satisfied. These preconditions and effects are each represented by a world state which has a subset of the world state symbols actually set (see figure 1, the Reload action has no precondition world state symbols set and has a single effect world state symbol set). Each action was designed to have functions that test the action's preconditions against an input world state to determine if they're met and also to apply the effects of the action to an input world state. To direct the A* search, actions require costs associated with them, cheaper actions that solve a goal are selected over more costly actions. If there is a tie during an A* insertion (i.e. if it turns out that an action has the same f value as something already on the open list during A*) there needs to be conflict resolution. To this end, each action has an integer that represents the precedence of the action. Actions with higher precedence are inserted before the lower precedence actions in the open list if such a tie occurs. This means high precedence actions will later be selected from the open list before the lower precedence actions.

Once an action has been selected for execution, a means of sending commands to the blackboard or subsystems was required in order for it to have an effect on the virtual world. This was performed by the ActivateAction function which every action implemented differently. A mechanism to determine when an action has completed was provided by the IsActionComplete function which returns true if the action has finished and false otherwise. When finished, an action must be deactivated which can also place information on the blackboard. As recommended by Orkin (Orkin, 2004) action's context preconditions and effects were designed to be functions that return true or false by querying the working memory or blackboard.

When the planner is searching to find a solution to a goal, two world states are maintained. The goal or desired world state along with the current world state are stored, both of which are altered as actions are applied during planning. If the planner recognises that an action can solve an unsatisfied world state property, the action's preconditions are added to the goal world state and the action's effects are applied to the current state that is being maintained.

To illustrate this concept and to introduce the actual planning process involved in the GOAP system a little further, figure 7 below depicts how the planner's world states are affected when planning. The current goal for the planner is selected to be KillEnemy (how this is done is explained in the next section). When a KillEnemy goal is selected, the planner then knows that the goal is satisfied if the world state symbol targetIsDead is set to true in the current state. The planner's goal world state is setup having just a single world state symbol defined, targetIsDead = true. This is performed in step 2 in figure 7. The planner's current state begins not having any world state symbols defined. Whatever symbols that are in the goal world state and aren't in the current world state of the planner are then added to the current world state. So the targetIsDead symbol is added. However the value applied to this symbol in the current world state is actually set to be the agent's *actual* world state symbol value (step 3). The planner then looks at the effects of the various actions the agent has available to it and sees that ActionAttack has an effect that solves the unsatisfied world state symbol i.e. by bringing the targetIsDead symbol to a true value (step 4). Upon executing the action within the planner, the targetIsDead symbol becomes satisfied (step 5). However the ActionAttack's world state preconditions are now added to the goal world state of the planner (step 6). This adds a single world state symbol in this case, weaponLoaded, which is set to true. As before, the planner adds whatever symbols are in the goal world state that weren't previously in the current world state. The new symbol's value is then set to be the same as the agent's actual current value for this symbol (step 7), which in this case sets weaponLoaded to be false (if the agent's actual world state symbol value for weaponLoaded was initially true then the planning process would finish as the goal and current states would be detected to be the same).

The goal and the current world states still aren't the same so more planning needs to be carried out. The planner looks for an action that has an effect which sets weaponLoaded to true and finds another action, ActionReload (step 8). The ActionReload action has no preconditions and only a single effect, which sets the value of the world state symbol weaponLoaded to true. Applying the action's effects cause the goal and current states to now be the same (step 10). The ActionReload has no preconditions to add to the goal state so the planner quits out and creates a plan. The plan includes the Reload action followed by the Attack action.

What was left out from this planning snippet was that for the ActionAttack or ActionReload to be even considered, their context preconditions must be satisfied which might check for things like if the agent has a valid target in view, a long range weapon etc.. Also this process takes place entirely in plan space and none of the actions are actually executed until the agent begins to step through its newly created plan.

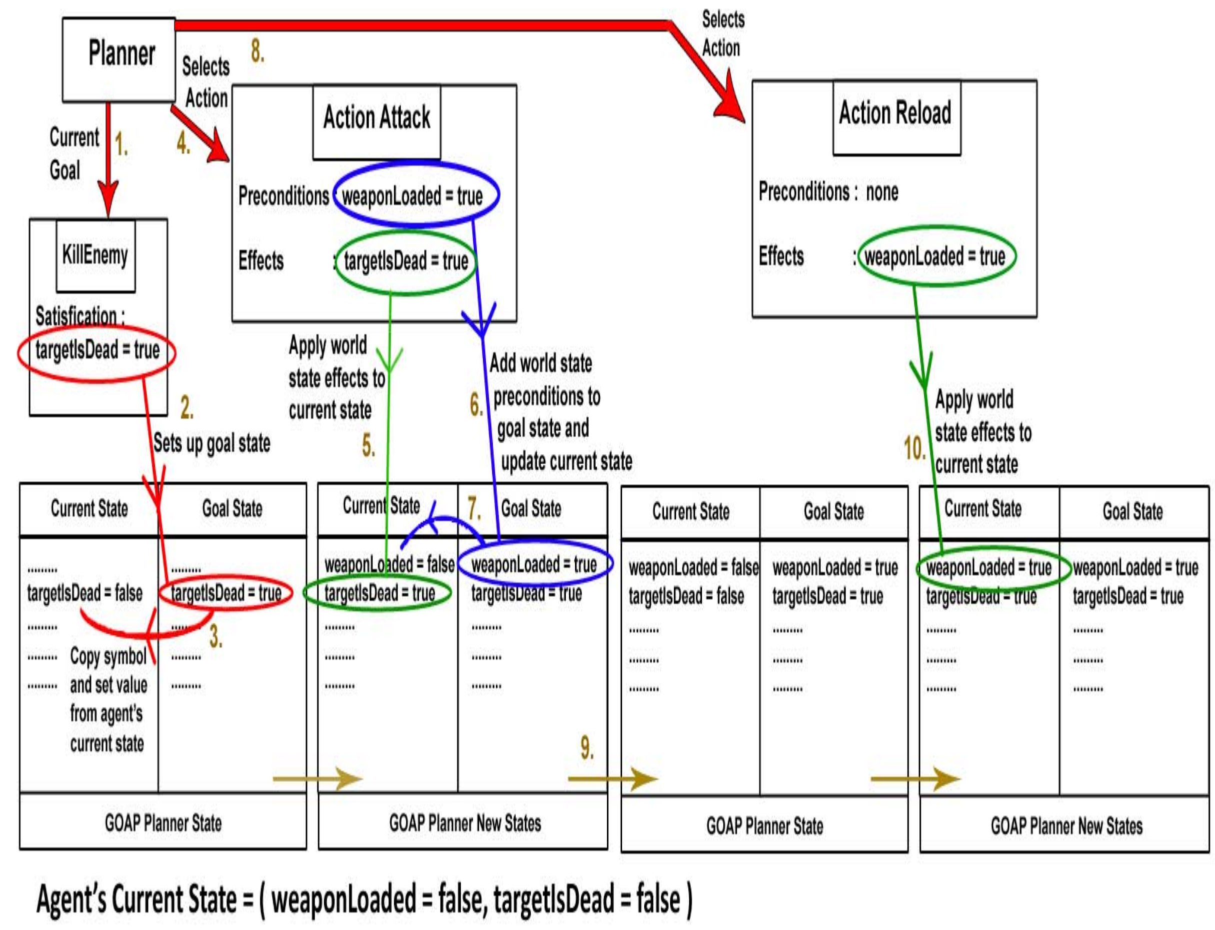


Figure 7 Applying an action when planning

As the plan formed by the planner is being executed, it needs to be regularly validated because if it becomes invalid, replanning is required. A plan is validated by just querying the action that is currently executing and checking if that action is still valid. Hence each action has a ValidateAction function that performs checks to ensure the action is still valid for execution, if any of these fail the action is stopped, the goal deactivated and the plan destroyed.

One of the key features of GOAP is that actions can be built upon one another i.e. layering of behaviours (Orkin, 2005). For example, a basic ActionAttack was designed which just gives basic attacking functionality but it was possible to build on-top of this by creating extra actions such as ActionAttackLongRange or ActionAttackShortRange. These extra actions make use of most of the same functions as the ActionAttack but bring about different behaviour by their activation, validation and completion criteria. This feature can simplify the addition of new behaviour as variants of the base action's behaviour can easily be built up.

3.8.4 Actions Container

The action container is a global singleton class that creates and encapsulates all the GOAP actions within the game. All the actions are shared across every agent and are created upon the initialisation of the program and stored in the actions container. When an agent is being created, its action set is loaded up from the XML file. Each of the agent's actions are obtained by requesting a pointer to the action from the container which is then stored in the agent's action set.

3.8.5 Goals

Goals are as important as actions in a GOAP system as without them the planner has nothing to plan towards. Orkin states that, in a similar way to actions, every goal should have the same basic structure (Orkin, 2002). Every goal should know when it is satisfied, be able to calculate its relevance and doesn't include the actual GOAP plan but just conditions that must be met.

Each goal has a float value between zero and one which represents its relevancy. When the planner is searching to find a new goal, each goal is requested in turn to update its relevancy. Goals relevancy values are determined by whatever is going on in the game at that time. For example, if the agent has a target in its sights then the KillEnemy goal could have a relevancy of 0.7, if it doesn't the relevancy would be 0. Once all the goals have updated their relevancies, the goal with the highest relevancy is requested to build a plan.

Most of the goals share the same functionality so it was decided, just like the action design, to create an abstract goal to encapsulate this. Each goal in the project extends this abstract goal and overwrites functions wherever new custom behaviour is required. For example, the functions that calculate a goal's relevancy and determine if a goal is satisfied are overloaded by every goal in the project.

World state symbol checks are used during planning to determine if a goal has become satisfied. However during the execution of a plan (i.e. after planning has finished), there can be a mixture of checks from the blackboard or from the agent's current world state. For example, during planning the GoalAttackDomPointOne is satisfied if the atDomPointOne world state symbol is set to true but during execution the goal is satisfied if the team overtakes Domination point one or if the navigation is complete.

Every goal has a handle to the global GOAP planner class which is used to create plans. When a goal is selected as an agent's current goal, it is instructed to create a plan. The goal requests the planner to run an A* search and find a sequence of actions that satisfy the goal. The plan created isn't stored alongside the goal, instead it is sent to the GOAP controller where it is stored and can be obtained every time it's needed thereafter. A goal is also in charge of activating, deactivating and advancing plans. When a plan is first activated the first action in the plan is executed. Every time the goal is subsequently updated the current action in the plan is polled to see if it is finished or has become invalid. If it is finished then the plan is advanced to the next action. Once the plan has finished executing (i.e. no more actions left to execute) then the goal is deactivated and the plan is discarded.

3.8.6 Goal Manager

The goal manager is the GOAP controllers' main interface with the GOAP system. When an agent is being initialised, the goal manager creates and stores each goal that is listed in the agent's goal set defined in the external XML file.

Every update of the goal manager updates the agent's current goal. This obtains the agent's current plan and either advances it or determines if it is invalid or finished. If the goal's plan is finished, invalid or if there is a request on the blackboard to replan, the goal manager searches for a new goal. It requests each of the stored goals to update their relevancies and picks the most relevant goal to build a plan. If a plan is successfully built then the goal is set to be the current goal and the plan is activated. If a plan fails to be built then the goal manager requests the next highest relevant goal and attempts to build a new plan. If all the goals with relevancies greater than zero fail to build plans then the goal manager has no goal and just waits until there is a request to replan.

3.9 The A* machine

The A* machine is an important part of almost every game nowadays as it is used to calculate paths through terrains as part of the navigation of an agent. The A* system designed for this project was based on the generic A* machine as described by Higgins (Higgins, 2002). To allow an A* machine to be used for different purposes, an A* storage mechanism is required to store the open and closed lists, a generic A* node along with custom A* goal and A* maps are needed for the different systems using the A* machine. Discussion of how the A* algorithm works is left for the reader's own investigation but pseudo code of the algorithm used in this project is detailed in Appendix A. There are however several important components of the A* machine that need further elaboration.

Whenever the A* machine is requested to run an A* search, the planner (whether it be navigation or GOAP) must first setup the A* map and A* goal that are to be used in the A* search. Different A* map and A* goals were required for the navigation and GOAP system. The A* map and the A* goal provide the exact same function-set to the A* machine but the navigation and GOAP system implement them differently so when they are called from the A* machine they bring about the required different functionality. However the A* machine requires a consistent A* node structure regardless of what system was used and this node structure is used to map from a generic A* node to either a GOAP action or path node.

3.9.1 A* Node

The basis of the A* search is the A* node. Each node has the same characteristics; each has an ID, an overall (f), actual (g) and heuristic (h) cost with links to its parent node, next and previous nodes which are used for maintaining the linked lists. The node ID is the most important piece of data for the A* node as it is what the A* map uses to map from the

A* node to a path node or GOAP action. There are two other types of nodes used in the A* search which extend the generic A* node, the AStarPlannerNode and AStarNavNode. The AStarPlannerNode contains two extra pieces of data over the regular A* nodes, the current and goal world states. The GOAP planner must keep track of current and goal world states upon applying actions when planning (see figure 7) and so these are stored alongside the A* nodes so that they can be revisited at a later stage if necessary. The AStarNavNode contains a reference to the node ID of a path node. The two types of A* map are called the AStarMapNavigation and AStarMapPlanner. Whenever the A* algorithm requests for a node to be created it sends in an ID into the A* map which creates a new node. The AStarMapPlanner creates an AStarPlannerNode while the AStarMapNavigation creates an AStarNavNode.

3.9.2 A* Storage

The A* storage maintains two linked lists of A* nodes, the open and closed list. The open list is sorted in ascending order according to the A* nodes actual (or f) values. Searching of the open and closed list for nodes with specific IDs was included as it was needed during the A* search.

3.9.3 A* Map

The A* map is used to determine how the generic A* nodes map to the GOAP actions and the path nodes. Another function of the A* map is to calculate and return what the neighbours of a specific node are.

When an action is created by the action container, the action is given a unique ID. During a planning A* search, each A* node's ID relates to an action ID. When the A* map is requested to obtain an action, it just requests the action from the actions container with the same ID as the A* node ID.

The A* planner map also maintains an action effects table. This idea is described by Orkin (Orkin, 2004) where he recommends the hashing of actions according to their effects to allow the planner to quickly find neighbouring actions. An array of vectors was setup where each entry in the array corresponds to one of the world state symbols (e.g. the first array entry is for the weaponLoaded symbol, the second targetIsDead symbol etc.). Each world state symbol of an action's effect world state that is set is examined and the action is added to appropriate vector for that world state symbol in the action-effect table. See figure 8 below for an illustration of the action-effect table being created.

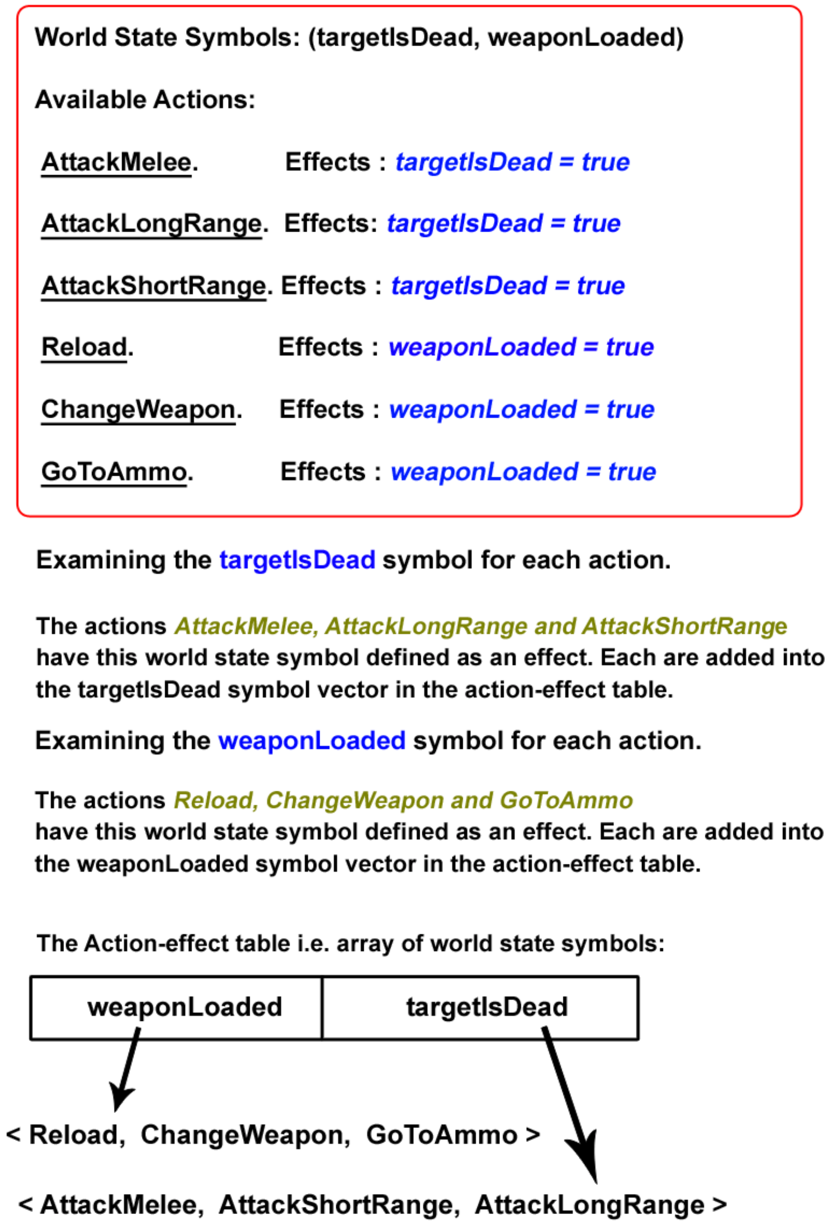

World State Symbols: (targetIsDead, weaponLoaded)
Available Actions:
AttackMelee. Effects : targetIsDead = true
AttackLongRange. Effects: targetIsDead = true
AttackShortRange. Effects : targetIsDead = true
Reload. Effects : weaponLoaded = true
ChangeWeapon. Effects : weaponLoaded = true
GoToAmmo. Effects : weaponLoaded = true
Examining the targetIsDead symbol for each action.
The actions AttackMelee, AttackLongRange and AttackShortRange have this world state symbol defined as an effect. Each are added into the targetIsDead symbol vector in the action-effect table.
Examining the weaponLoaded symbol for each action.
The actions Reload, ChangeWeapon and GoToAmmo have this world state symbol defined as an effect. Each are added into the weaponLoaded symbol vector in the action-effect table.
The Action-effect table i.e. array of world state symbols:
weaponLoaded
targetIsDead
< Reload, ChangeWeapon, GoToAmmo >
< AttackMelee, AttackShortRange, AttackLongRange >

Figure 8 Creation of the action-effect table

Now when the A* map searches for neighbouring actions of a node during the A* search, the input A* node is converted to its corresponding AStarPlannerNode. A check is then performed to see which symbols are unsatisfied between the node's current and goal world states. If an unsatisfied symbol exists, the effects table is examined for that world state symbol. Each action associated with the world state symbol is examined and if the action solves the unsatisfied symbol then it is considered a possible neighbour. This process continues until all possible actions have been examined for the symbol in the action effects table. Figure 9 shows the neighbours of a node being calculated during an A* search.

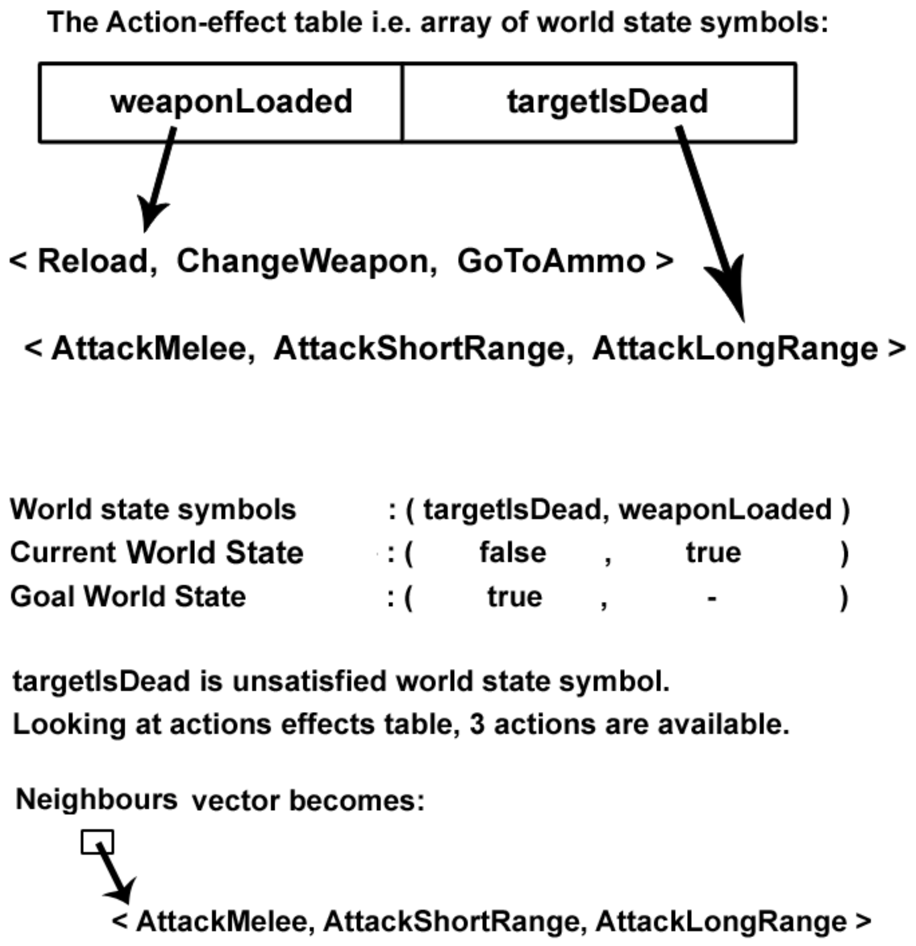

The Action-effect table i.e. array of world state symbols:
weaponLoaded
targetIsDead
< Reload, ChangeWeapon, GoToAmmo >
< AttackMelee, AttackShortRange, AttackLongRange >
World state symbols : (targetIsDead, weaponLoaded)
Current World State : (false , true)
Goal World State : (true , -)
targetIsDead is unsatisfied world state symbol.
Looking at actions effects table, 3 actions are available.
Neighbours vector becomes:
< AttackMelee, AttackShortRange, AttackLongRange >

Figure 9 Calculation of neighbours during A* search

During the pathfinding A* search, the A* map performs the same functions as the planning A* map. When the path nodes are being created by the path node container class, each path node is given a unique ID just like the actions. When the navigation A* map requires a path node it requests the path node container class to find the node with the same ID as the A* node. The navigation A* map must also find the neighbours of an A* node. For a given A* node, the A* map just obtains the path node that matches the A* node's ID and builds up a list of neighbours by looking around this path node and eliminating unwalkable nodes.

3.9.4 A* Goal

The A* goal was designed to determine what the heuristic and actual costs are between A* nodes, to test if the A* search has finished and to check if the current plan is valid. If forms an integral part of the planning process for the GOAP system as all the world state changes take place within the A* goal.

The navigation A* goal design was quite simple. When determining the A* node cost (i.e. the g value) the A* goal just returns the cost to go from one path node to another. The heuristic cost between nodes is the distance between them 'as the crow flies'. The A* goal can recognise if the A* search has finished by checking if the current A* node being analysed is the same node as the goal node.

The planning A* goal is a little more complex. As can be seen from the pseudo-code for the A* search in Appendix A, when examining a given neighbour of the current node, the value for g or the actual cost is first calculated followed by the calculation of the heuristic cost, h. These values are then combined to obtain the overall cost, f, for the neighbour node which determines where on the open list the neighbour is placed.

The actual method of calculating the heuristic and actual cost of the GOAP actions was designed from Orkin's explanation of the F.E.A.R. A* machine (Orkin, 2005). When the planner A* search begins, the current and goal world states of the AStarPlannerNode are initially setup. The planner's A* goal maintains a pointer to the actual GOAP goal being searched towards. The GOAP goal's satisfaction conditions are applied to the AStarPlannerNode's goal state (step 2 in figure 7). The current state is subsequently updated by adding each new symbol that has been added to the goal world state and setting its value from the agent's current state (step 3 in figure 7). The heuristic cost returned is the number of symbols different between the current and goal states.

When determining the value of g for a neighbour node, it is first necessary to map from the input A* node and obtain the matching action using the A* map. The action's cost is then returned as the value for g. The world states of the current AStarPlannerNode are transferred to the neighbour AStarPlannerNode so that the neighbour has the same current and goal world states as the current node before moving on to calculating the heuristic cost (step 9 in figure 7).

When determining the heuristic cost, the A* goal obtains the GOAP action from the A* map by passing in the neighbours node ID. This action is then executed which involves solving whatever unsatisfied world state symbols the action can solve, applying the action's world state effects to the current state and finally merging the agent's current world state with any new symbols added to the goal state (steps 5, 6 and 7 in figure 7). Again the heuristic cost returned is the number of symbols different between the current and goal states of the AStarPlannerNode after applying the GOAP action.

The planner A* goal must also determine if an A* search has finished. This is performed by checking if the current node's goal and current states are the same. If so then a copy of the agent's *current* world state is obtained. Each of the actions that are part of the plan so far are applied to this world state one by one. If the world state left after applying the actions satisfies the goal conditions (i.e. there are no unsatisfied world state symbols) then the planning process has completed and the A* search is deemed to have finished.

3.10 The planners

Just like all the other parts in the A* system, there is a planner for both the navigation and the GOAP system. Each of the planners is a global singleton class that is shared across all the agents in the game. The role of the planner is to take in a goal (whether it be a GOAP goal or navigation target) and request the A* machine to run an A* search to satisfy that goal. Once the A* algorithm has completed successfully the planners obtain the current node that the A* machine breaks out at and this is added as the first step in the plan. The planner then proceeds to add every parent of the node until there are no more parents left to add which then completes the plan.

Each GOAP goal has a reference to the global GOAP planner and requests it to build a plan if asked to do so from the goal manager. The navigation manager has a reference to the navigation planner and it builds plans if a new navigation target is placed on the blackboard.

3.11GOAP Controller

The GOAP controller was designed to extend the base AI module. It is charged with updating the various subsystem managers, the agent's world state and the GOAP system. Updating the agent's world state is a case of setting world state symbols to be true or false depending on the contents of the blackboard. Updating the GOAP system simply updates to the goal manager. The updating of the subsystems involves calls to the sensory, target, weapons and navigation managers along with updating of the working memory. The GOAP controller is also charged with cleaning up and resetting all the systems if an agent dies in the game.

See figure 10 for the UML of the GOAP system.

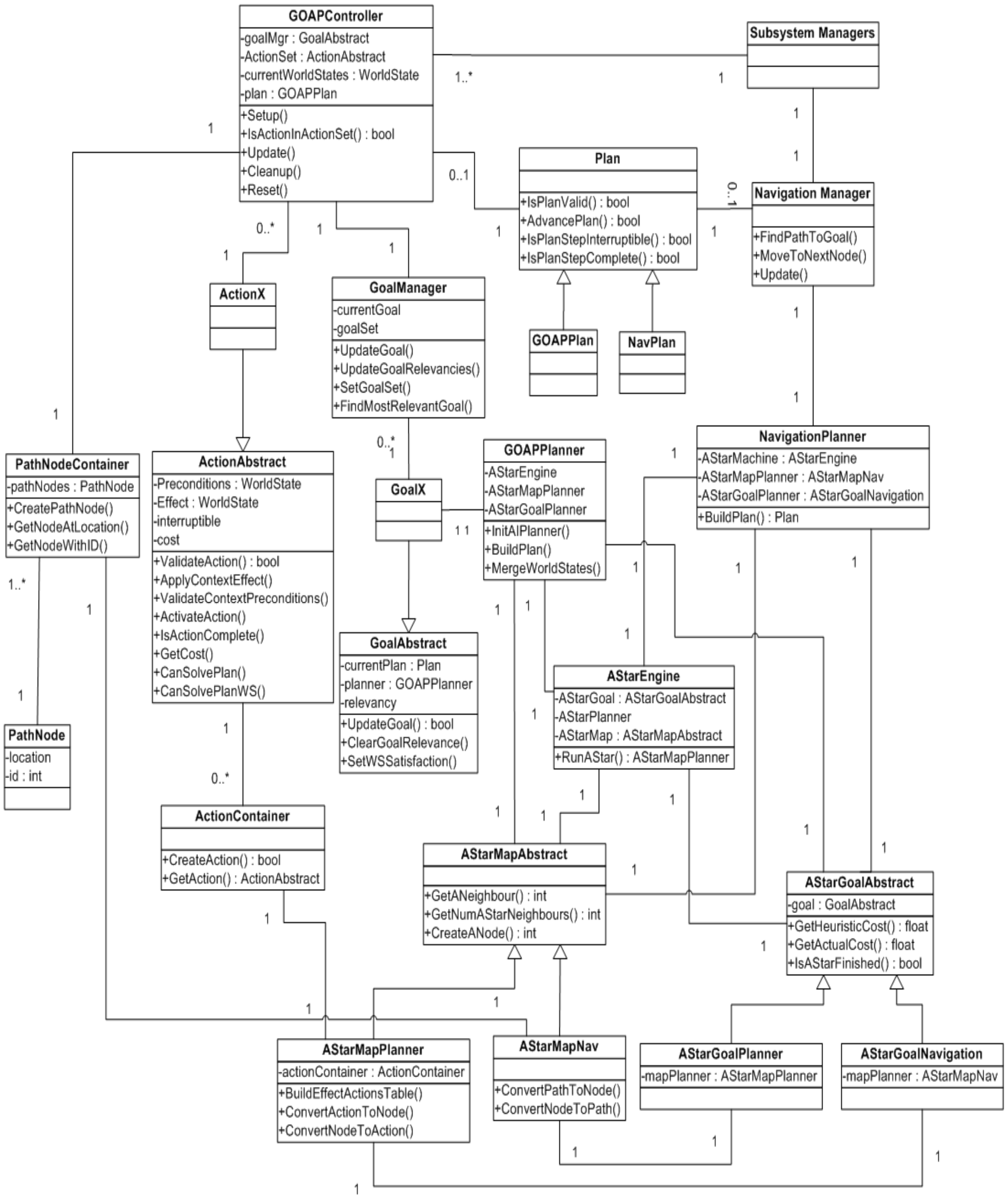


Figure 10 GOAP System UML

3.11 FSM Design

3.11.1 The Finite State Machine architecture

It was decided to go with Buckland's design for FSMs (Buckland, 2005) due to the available source code, documentation and relatively straight-forward implementation. Buckland encodes FSM states as C++ templates and encapsulates the actual state machine and state implementation in two single header files. The programmer then just has to implement the designed FSM states including the state transition logic, set an initial state for the state machine and then update it every frame. Each state has three primary functions; Enter, Execute and Exit. When a state is first initialised, the Enter function is called. Every time the state machine is updated, the current state's Execute function is called. When the current state of the FSM is changed, the current state's Exit function is called before the change is finally made. State transition logic is placed in both the Enter and Execute functions. States can transition if a state completes in the virtual world or if something occurs in the world that invalidates the state. The rest of the FSM design dealt with determining all the possible states and coming up with what the state transitions are between them.

The states designed as part of this project are outlined in Appendix B.

3.11.2 The FSM Controller

The FSM controller was designed to extend the base AI module. It has much of the same functionality as the GOAP controller, in that it updates the various subsystem managers but instead of updating the goal manager it just updates a reference to the state machine which runs the current states' Execute function.

Chapter 4. IMPLEMENTATION

4.1 Standalone GOAP system

The GOAP system was developed separately from the game application to begin with. Getting the GOAP system setup initially with the abstract action, goal, planner, A* machine and other components was completed quite quickly due to the design in place, however it was impossible to test the system without any goals or actions. Some sample goals and actions that were to be included in the game were created and testing was performed by manually setting a goal's relevancy e.g. the KillEnemy goal to a high value so that it was always selected by the GoalManager as the most relevant goal. The plans that were built for each goal were inspected and validated before moving on to implement new goals and actions.

Originally it was hoped that world state symbols could include variable support for the GOAP system. The design included variable support from the off but several problems were encountered during the implementation that forced the world state symbols to be reduced back to Boolean values. It was intended to have support for integers, floats, path nodes and other objects such as AIModules and pickups. To include these, functions in the world state class that get and set the values for each different variable type were required. Not only that but a means of testing whether a variable is equal to another variable or not was also required. This was straight-forward for integers and floats but not so for objects. To test objects against one another the only way to match them was to perform a custom test for each different type of object. If the world state symbol was discovered to be a path node and was being compared to another node, the test would involve checking the location to see if it was the same. If an AI module was being tested against another AI module symbol then a different test would be needed. Instead of having many different variable types stored alongside the world state, it made sense to use a single object (called an OBJECT) as the base class for all objects in the game and it would be used to store objects in a world state. Now to test one object against another, a mechanism was required to convert from the base object to the real object (i.e. OBJECT to path node, AIModule etc.) so that custom tests could be run. While this conversion may have been possible, there existed a bigger problem and that was deciding how the variables were represented when used in actions and goals. Basically, how could a world state symbol that is a variable be represented for preconditions or effects if its value can't be pre-determined in the C++ code? No solution to this question arose throughout implementation and so it was decided to instead just go with the Boolean approach which had a clear cut solution available.

Using just Boolean symbols brought about their own challenges. The limitation of Boolean symbols in the planner was something that caused problems when implementing the actions and goals initially. The major issue surrounded movement of the agent. The original design was to have a single world state Boolean symbol to handle movement called atTargetNode. The atTargetNode symbol was supposed to be updated by the navigation manager. If the agent is at its final destination then the atTargetNode was set to true, otherwise it was set to false. It was planned to have a GoTo goal that was satisfied if the atTargetNode was set to true. Alongside the GoTo goal there was going to be a GoTo action which has an effect of setting the atTargetNode symbol value to true. When the GoTo action is activated, it would inform the blackboard of the new navigation target which in turn moves the agent. So, suppose the agent is at its final node and has no new destination, then the atTargetNode world state symbol is set to true. Now if the GoTo goal becomes relevant (e.g. an order is received from the squad manager to go somewhere) then for the goal to be satisfied, the atTargetNode symbol must be true. The planner will attempt to find actions to satisfy the GoTo goal but realises the agent's current world state already satisfies the goal state, quits out and doesn't create a plan. The solution to this issue was to never actually set the value of the atTargetNode world state symbol for the agent's current world state. The agent starts off with the atTargetNode symbol being false. The symbol's value is only changed when planning is undertaken but this only affects *copies* of the agent's world state, it never affects the *actual* agent's world state.

So now every time the GoTo goal is selected, the atTargetNode symbol will always be false and thus the GoTo action gets picked by the planner to solve the goal. When the GoTo action is executed the effects of the action aren't actually applied to the agent's current state and so the agent's atTargetNode symbol remains at false for the entirety of the game. This ploy was also used for the targetIsDead symbol but of course there are still world state symbols such as inMeleeRange, weaponLoaded etc. that are updated every frame as they are needed by the planner to discover when actions are complete or when goals are relevant.

Once the GOAP system was debugged and working with the sample actions and goals, it was imported into the Domination game and work on the agent architecture begun. However some unforeseen problems occurred during the implementation phase of the agent architecture that needed some attention.

4.2 Agent Architecture

The biggest obstacle encountered during the implementation of the agent's architecture was during the development of the working memory. The working memory is a collection of facts whereby each fact is a piece of information that relates to the game world. It was originally designed to contain several different types of working memory fact: valid attack positions, cover positions, pickup data, enemy and friendly agent data, events and orders. The sensory manager populates the working memory with information it senses about the game world. It soon became apparent that the memory usage and cleanup of the working memory would become a problem. When attempting to record valid attacking or covering positions, every time a new valid cover or attack position was discovered a new working memory fact was created and was given a belief value. When an attack action was looking to find an attack position, it would query the working memory which would search through all the facts and return the position with the highest belief. However targets change often and agents move around the world quickly. Nodes that had high belief values when inserted weren't so good only a couple of seconds later. So agents were being sent to locations that were good a short time previously (when the sensor was last updated) but weren't ideal when being queried.

One solution explored was to update the sensors more regularly. Because there are so many path nodes in the game (256x256) updating too regularly caused excessive amounts of facts to be created and the cleanup effort alone was harming frame-rate. The other choice was to only allow the addition of a single memory fact every time the sensor was updated. However even with this too many facts were being created and the performance suffered. Hence, instead of using the working memory structure to record certain data, actions were allowed to directly query the navigation manager and other systems to find the best positions to attack, the best cover positions, closest pickup positions etc. Other information that didn't need to be as up to date is still stored in the working memory like orders, events and enemy details.

4.3 Game-specific GOAP issues

One of the biggest hurdles experienced throughout the implementation was detecting when actions have completed and getting actions to execute alongside the various subsystems, where different subsystems needed separate activation. Initially, quite low-level actions were designed for the game. There was to be GoTo actions to handle movement, separate attack actions to handle attacking and other ancillary actions to deal with actions like reloading, changing weapons and so on. Only one action can be active at any time but it was often the case that an agent needed to move and attack at the same time. Suppose the agent was moving to some location in the world using a GoTo action and is attacked en-route. The agent is in the process of a GoTo action which doesn't actually attack. If the agent just invalidated its current plan and replanned to activate an attack action then the agent would just stand there attacking as movement is handled by GoTo actions only. So it was decided to make the actions a little more high level and control different subsystems simultaneously if required. For example, the AttackShortRange action states that if the target isn't visible then the agent should move within range of it or if it is patrolling then the agent should move back to protect the patrol position. At the same time, the action must activate an attack so it is essentially controlling two subsystems, the weapons manager and the navigation manager.

Determining when actions complete was quite challenging and this project employed two very simple finite state machines to determine if subsystems have finished their tasks. The first state machine records the status of navigation, which can be unset, incomplete or complete. The second state machine records the status of an attack, which again can be unset, incomplete or complete. Orkin also used simple finite state machines in the F.E.A.R. system, which had GoTo, Animate or UseSmartObject FSMs (Orkin, 2006). The status of each of the state machines is placed on the blackboard so that actions can access them and decide when they are finished. When an action such as the AttackShortRange as described earlier activates, two of the subsystems are requested to perform work. Does the action complete when navigation is completed or when attacking is completed? As part of the plan validation, the plan's current action is validated every frame. Getting these validation checks right caused a good deal of problems as actions could regularly activate and become invalidated quickly. Finding the correct completion and validation criteria required substantial tuning and extensive debugging.

Determining the relevancy values of goals is a tricky task and entailed a degree of guesswork along with trial and error. When agent's goal relevancies are updated, there are quite often two or more goals that have relevancy values higher than zero, say for example the AttackDomPointTwo goal and the Patrol goal. If the Patrol goal's relevancy is set too low then the agent will always go and plan for the AttackDomPointTwo goal and the Patrol goal may rarely or never be selected. On the other hand if the Patrol goal's relevancy is set too high agent's may all go and prioritise this goal, end up patrolling a single Domination point and never trigger the AttackDomPointTwo goal. If goal's relevancy values are constantly set too low then they will rarely get chosen and if they are too high they can be chosen too often and cause repetitive behaviour. Hence choosing the right values for goal relevancies was an issue that required a lot of attention and testing during the implementation.

Action costs suffer from a similar problem as they need to be hand-tuned so that certain actions get chosen over others when performing A*. As the action cost affects where an action gets placed on the open list (i.e. it is the g value from the $f = g + h$ formula), if the action cost is too high it may be the case that there is always a cheaper action available to the planner and the action never gets selected even if it is more appropriate. However this problem was overcome to some extent by tuning the cost values and by creating comprehensive context preconditions which can limit the number of other actions selected by the planner that can clash.

The biggest challenge faced during development of the GOAP system was keeping the planner at an acceptable level of efficiency. Planning using A* is a CPU intensive task so it was desirable to keep the number of searches to the very minimum. Recalculating the relevancies of all the goals and finding the highest goal to plan towards can also use up resources as goals query the working memory. If the working memory contains many working memory facts then this search can become time-consuming. Instead of recalculating the relevancies of all goals, it is sometimes possible to just re-build a plan towards the same goal if a previous plan has become invalidated. Suppose the current goal is KillEnemy and upon the execution of an Attack action it is discovered the agent has no ammunition. The plan becomes invalidated but instead of recalculating all goal relevancies again, a replan is triggered which requests the planner just to build a plan towards the KillEnemy goal and not search for a new goal.

Every time the relevancy of a goal is checked, a timer is set for that goal that stops it being checked again for a certain period of time. This reduces the number of goals that are checked when looking for a valid goal and thus can improve performance.

One way of limiting the amount of replanning was to ensure that actions didn't complete or weren't invalidated too often. Obviously this can depend on the action in the game itself but the best way of ensuring that actions weren't invalidated too soon was to make sure that the context preconditions and world state preconditions were comprehensive enough before the action was activated. An action is only invalidated if the action can't possibly function any more.

However, due to the fast-paced nature of the game developed and even with the optimisations listed, there was quite a lot of planning and replanning involved during the game. If actions lasted longer than they do in the Domination game, there could possibly be minutes between having to plan or replan which of course would improve performance substantially. Also, because there are relatively few actions and goals in this project there generally isn't more than two or three actions involved in solving any given goal at a time. Hence plans are regularly finishing as only two or three actions are included in each plan and replanning occurs. In a commercial game there would be far more actions and goals (F.E.A.R has 71 goals and 121 actions) and plans would have many more actions contained within them. More actions in a plan mean that it could take longer to execute before the plan finishes and so less planning would occur as a consequence and efficiency would improve.

See the Appendix B for the goals and actions created as part of the project and for brief descriptions of their functionality.

One of the benefits of GOAP that also became problematic during development was the unpredictability of the system. As agents can pick any goals or actions available to them, predicting behaviour in the GOAP system is extremely difficult. Debugging the planning process was quite a painstaking operation involving many manual walkthroughs of each stage of the planning (goal selection, correct neighbours being selected during A*, plan formulation, plan activation, action activation, action validation, action completion etc.) in Visual Studio to determine where a problem occurred with the GOAP system. The logging system along with the outputting of data to screen helped somewhat but debugging the GOAP system definitely ranked as being far more difficult than debugging the FSM system.

4.4 Squad manager implementation

The squad manager implementation experienced several setbacks throughout development due to the fact that the game moves so quickly and agents were often killed before they could carry out squad commands. The squad manager is in charge of allocating agents to squads, assigning tasks to these squads and updating them. The Domination squad manager maintains two squads, one for each Domination point. When there is an agent in the pool of agents, it can be allocated to either the closest squad to the agent or allocated to the other squad if the closest squad is full. When updated, the individual squad handles whatever task assigned to it and issues orders to the squad members by creating Order working memory facts. When the agent is calculating the relevancy of a goal, it checks its working memory for orders and this can affect the relevancy value for a goal.

Originally there was going to be several different types of tasks for the squads, Advance, PatrolLocation, AttackLocation, AttackLocationFlank, AttackLocationFormation, GoTo, Rendezvous, Advance and Follow. All these tasks were actually implemented and tested but performed poorly in practice. The concept behind the AttackLocationFormation was to first get the squad members to meet up and then advance in a flocking formation to attack a Domination point. However to meet up the agents had to go to a location in the map and then wait for the others. It was found that agents would arrive at a rendezvous location and either the agents moving to that location would be killed or the agent itself would die thus invalidating the task. Also while agents were waiting around, the opposition teams were taking over Domination points or adding more squad members to patrol and defend them. Due to similar problems experienced with other squad tasks it was decided to stick to four primary tasks which work quite well for the Domination game, PatrolLocation, AttackLocation, AttackLocationFlank and GoTo.

4.5 FSM Implementation

As FSMs were the final part of the project to be implemented, the GOAP system and all the subsystems were already in place. Hence the development of the system was relatively quick as all of the bugs in the subsystems (navigation manager, sensory manager, target manager, working memory etc) had been eliminated. As the goals and actions of the GOAP system had already been fleshed out and tuned, they directed what states would be needed by the FSM system and helped refine the previous FSM state design. The preconditions of individual GOAP actions were also in place so this helped flesh out the state transitions between the various states. As Buckland provided the state machine architecture, the main task involved with implementing the FSM system was to create all the designed states. This proved quite tricky as several states could transition to other states, if not all of them. Even with the design in place, capturing all of the possible transitions was an arduous and error prone task and took a good deal of debugging as states often 'flipped' rapidly back and forth between one another or wrong states were selected.

Even though there were only twelve states in the Domination game finite state machine, figure 11 outlines the transitions between the states. The large amount of state transitions is clearly visible even though some of the transitions were left out of this diagram for the sake of clarity (notably the Idle state transitions to any other state, it only has states transitioning to it in this diagram).

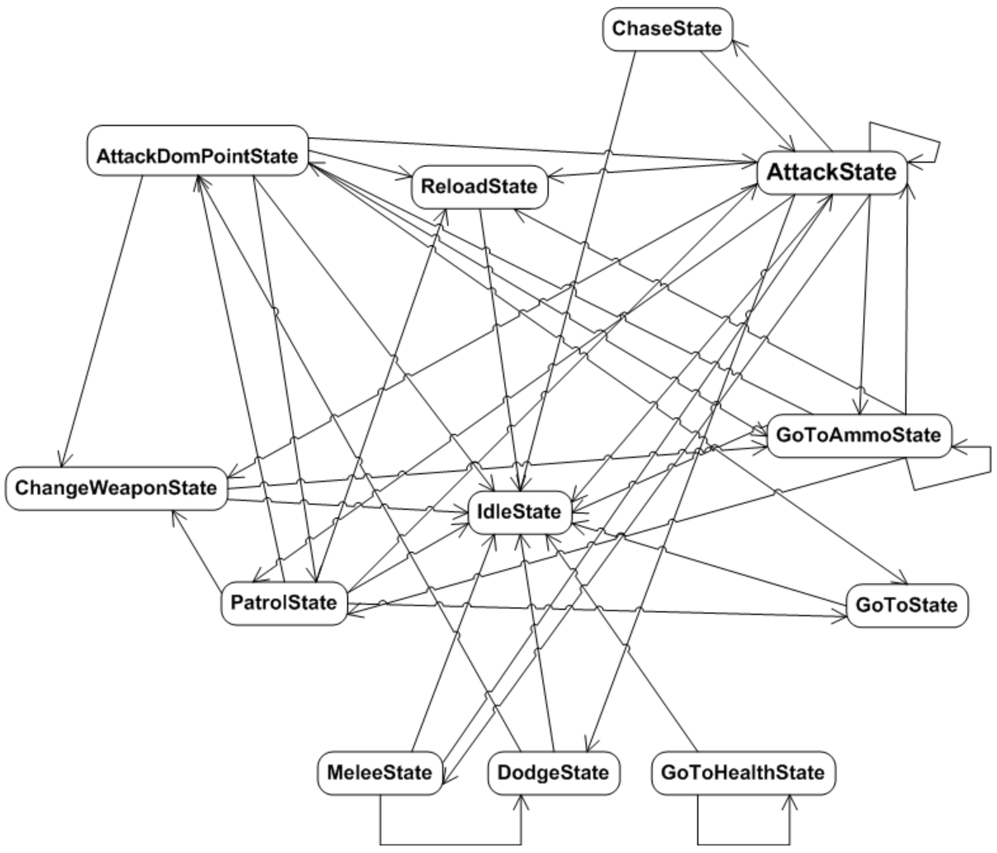


Figure 11 Transitions between states

Chapter 5. RESULTS AND DISCUSSION

5.1 Extra game types

Even though the main goal of this project was to test out the GOAP system against the FSM system in a Domination game, other game modes used in UT2005 were developed near the end of the project. Before testing begun, three extra game modes were coded up as an exercise in determining how difficult it would be to extend both of the AI systems and also to obtain more data for the experiments. The first game mode, called Deathmatch, simply pitches every agent against one another and there are no teams or squads involved. The amount of changes required for the GOAP and FSM systems to function in this game mode were negligible. The FSM needed a few extra if statements inserted to check for a specific game mode for certain state transitions. The GOAP system simply required a new XML file that disabled certain goals and actions that related to the Domination game.

The Last Man Standing game mode pitches every agent against one another. However once an agent is killed it doesn't get re-started until a new round is triggered. The last agent alive in the game is the winner of the round and then a new round is started. Again the implementation of this involved creating an extra XML file for the GOAP system while the FSM system required extra if statements scattered amongst the various states to inform the FSM system when to change state.

The final extra game mode implemented was UT2005's popular Capture the Flag team game. In this game mode, each team has a base and a flag. The flag initially starts off in a team's base and the goal of the game is to bring the opposing team's flag back to the team's base. A team scores a point if it returns the opposition flag to its own base. Like the Domination game mode, a good deal of squad co-ordination is required as each team must protect its own flag and bring it back to its own base. It must also try and attack the opposition flag and return it to base. This game mode really highlighted the GOAP system's flexibility. Several of the FSM states needed to be entirely re-written as the state transition logic was completely different. Only two new goals and four new actions were created for the GOAP system and it then functioned just fine. An extra XML file was again created to indicate which goals and actions the agent was allowed. A new squad manager and squad system was devised for this game mode and provides the same orders to both the GOAP and FSM systems.

See Appendix C for a description of the game application developed for this project.

5.2 Experimental Plan

The testing for this project involved recording data from pitching the different types of AI against one another in the different game modes. Each test involved ten AI agents and every agent was controlled by either GOAP or FSM. If the game mode was a team based game (i.e. Domination or Capture the Flag) then five agents of the same AI type were placed on the same team and an option was provided to enable squad tactics for that team. The number of kills racked up by each agent and team was noted along with how many times each agent dies. This was the primary type of test data that was collected from the Deathmatch and the Last Man Standing game modes. For the Domination and Capture the Flag game modes, the number of kills was stored but also recorded was how many points each AI team scores. As the AI can be coordinated using the squad manager into squads, the average distance between team members was also recorded.

Technical information about the two systems was also recorded for comparison purposes. Intel's VTune tool is a software profiler which plugs into Visual Studio 2005 and can measure the CPU usage, memory allocation and bottlenecks in an application and was used to gather technical data. Two teams of the same AI type were run against one another in a Deathmatch game for twenty-five minutes and the VTune data along with average frames per second (FPS) was stored. The number of times a GOAP system plans on average alongside the average number of state changes was obtained from ten twenty minute tests between the two types of AI (GOAP without squads and FSMs without squads).

Experimental plan

For each map:

Last Man Standing and Deathmatch games

GOAP vs. FSM – 1x60minute tests

Domination and Capture the Flag

GOAP with Squad vs. FSM with Squad – 5 x 20 minute tests

GOAP with Squad vs. FSM without squad – 5 x 20 minute tests

GOAP without squad vs. FSM without squad – 5 x 20 minute tests

FSM Squad vs. GOAP without squad – 5 x 20 minute tests

GOAP Squad vs. GOAP without squad – 5 x 20 minute tests

FSM squad vs. FSM without squad – 5 x 20 minute tests

Technical Experiments

GOAP No Squad vs. GOAP No Squad – 1x25 minute test

FSM No Squad vs. FSM No Squad –1x 25 minute test

These tests were performed in the Deathmatch game mode whilst being monitored by VTune.

Total time for experiments came to 1490 minutes which approximates to twenty-five hours of testing.

5.3 Experimental Results and discussion

The Deathmatch and Last Man Standing results are the obvious results to examine first as they indicate how the GOAP and FSM systems performed when pitched against one another without any squad commands or any overall goal to achieve in the game mode. In the Last Man Standing game, the GOAP system out-performed the FSM system in both maps, see figure 12 for the experiment results. The GOAP system survives longer (i.e. fewer deaths) and kills more agents than the FSM and as a consequence also wins the Last Man Standing game more often. With scores of 193 for the GOAP system and 81 for the FSM, the GOAP system scores more than double the FSM system, clearly illustrating the difference between the two systems without any squads attached. During testing, it was possible to clearly see and almost predict the transitions between states for the FSM agents while this wasn't the case when inspecting the GOAP system. This is a common major criticism of FSMs.

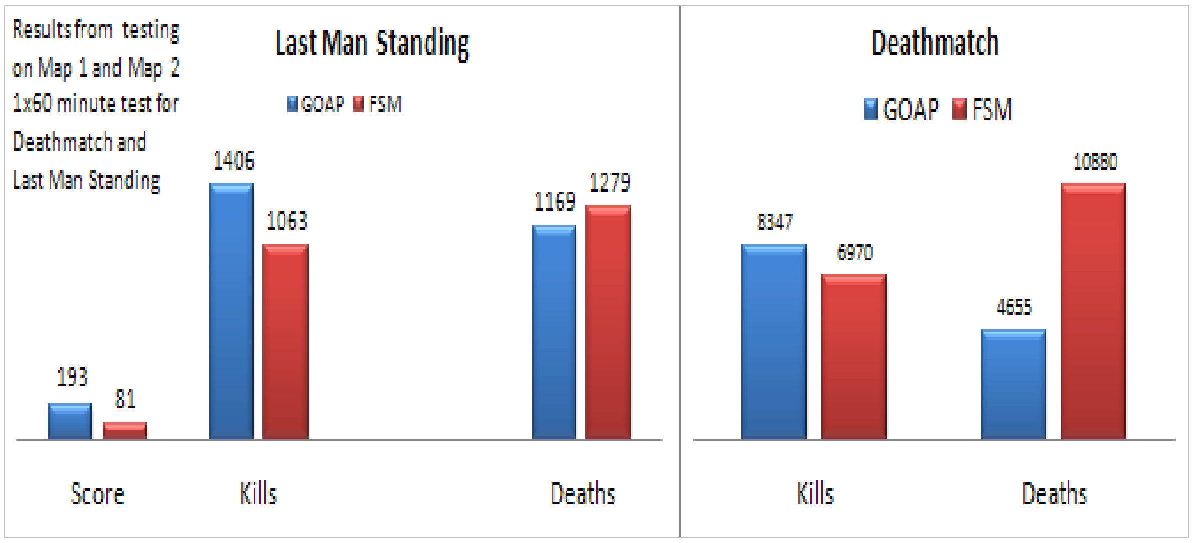


Figure 12 Last Man Standing & Deathmatch results

The Capture the Flag results in figure 13 also highlight that even without any squad commands, the GOAP system scores more points than the FSM. The GOAP system with squads enabled defeated the FSM team with squads enabled by some distance. Interestingly, GOAP without squad commands also defeated the FSM system with squads enabled over several tests highlighting GOAPs superiority in working towards a goal. The Capture the Flag results show that GOAP carried out its squad tasks far more effectively than the FSMs. While the number of kills were close for several tests (e.g. the FSM with squads vs. the FSM without squads), the team with the squads enabled still scored higher thus showing the value of the squad coordination and tactics.

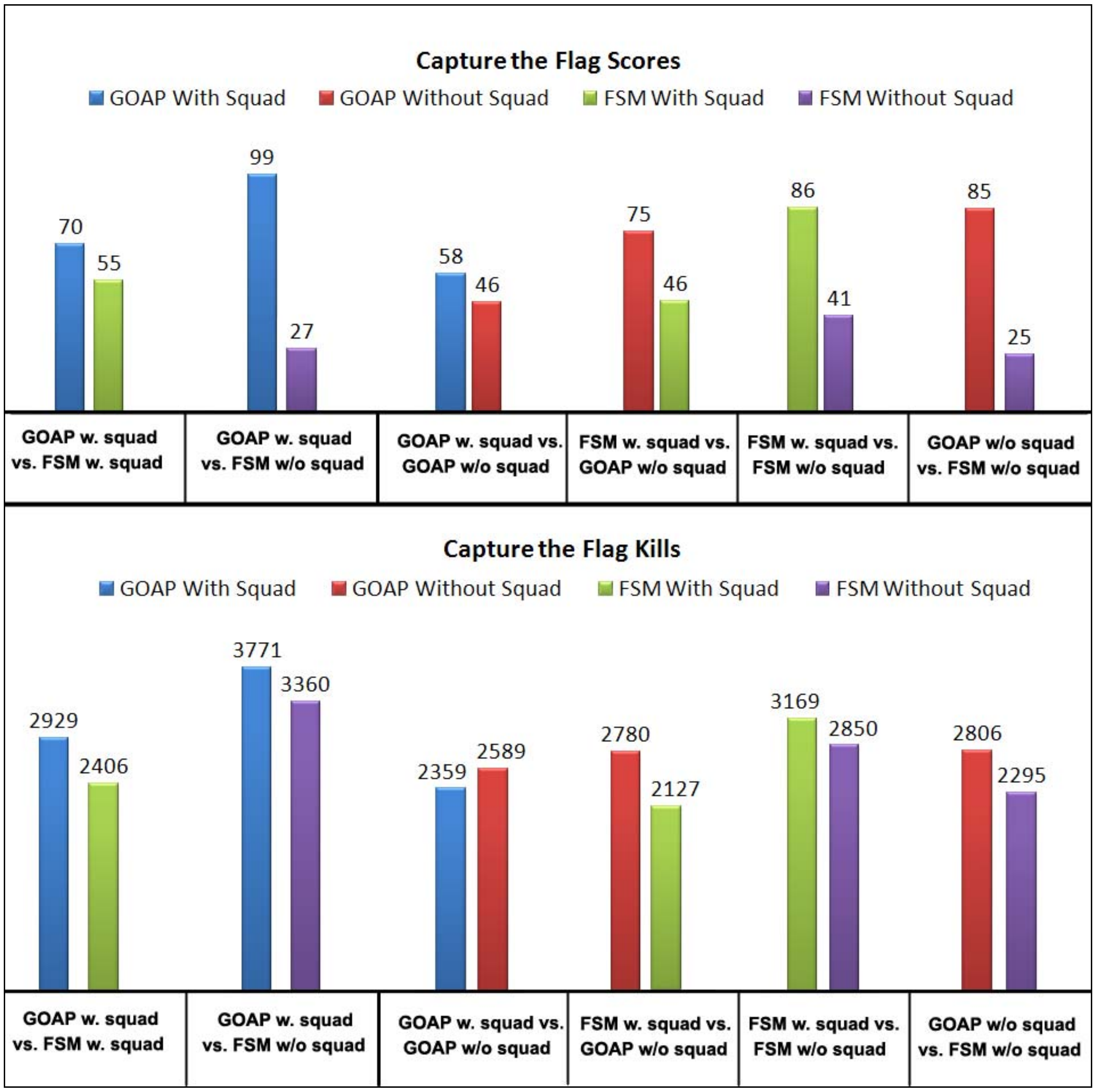


Figure 13 Capture the Flag Game Results

The Domination game results are displayed in figure 14. Much like Munoz-Avila's results from a similar project (Munóz-Avila and Lee-Urban, 2005), the AI with squad behaviour enabled defeats every other type of AI which doesn't have squad behaviour enabled and GOAP beats the FSMs overall. In every test between GOAP and FSMs, the GOAP system manages to get more kills than the FSM system, this again highlights that the GOAP system was better than the FSM at the individual level. As both the FSMs and the GOAP system have exactly the same squad commands to work with, the results indicate that the GOAP system appears to integrate better with a squad system, for this project in any case.

The results between the two AI systems without squad commands were quite close, there are only four between the scores after two hundred minutes of testing and three between the numbers of kills. The reason for this is that several Domination tests ended in stalemate with all the team members just guarding a Domination point and not attacking the other teams due to the lack of coordination or knowledge as to what other team members were doing.

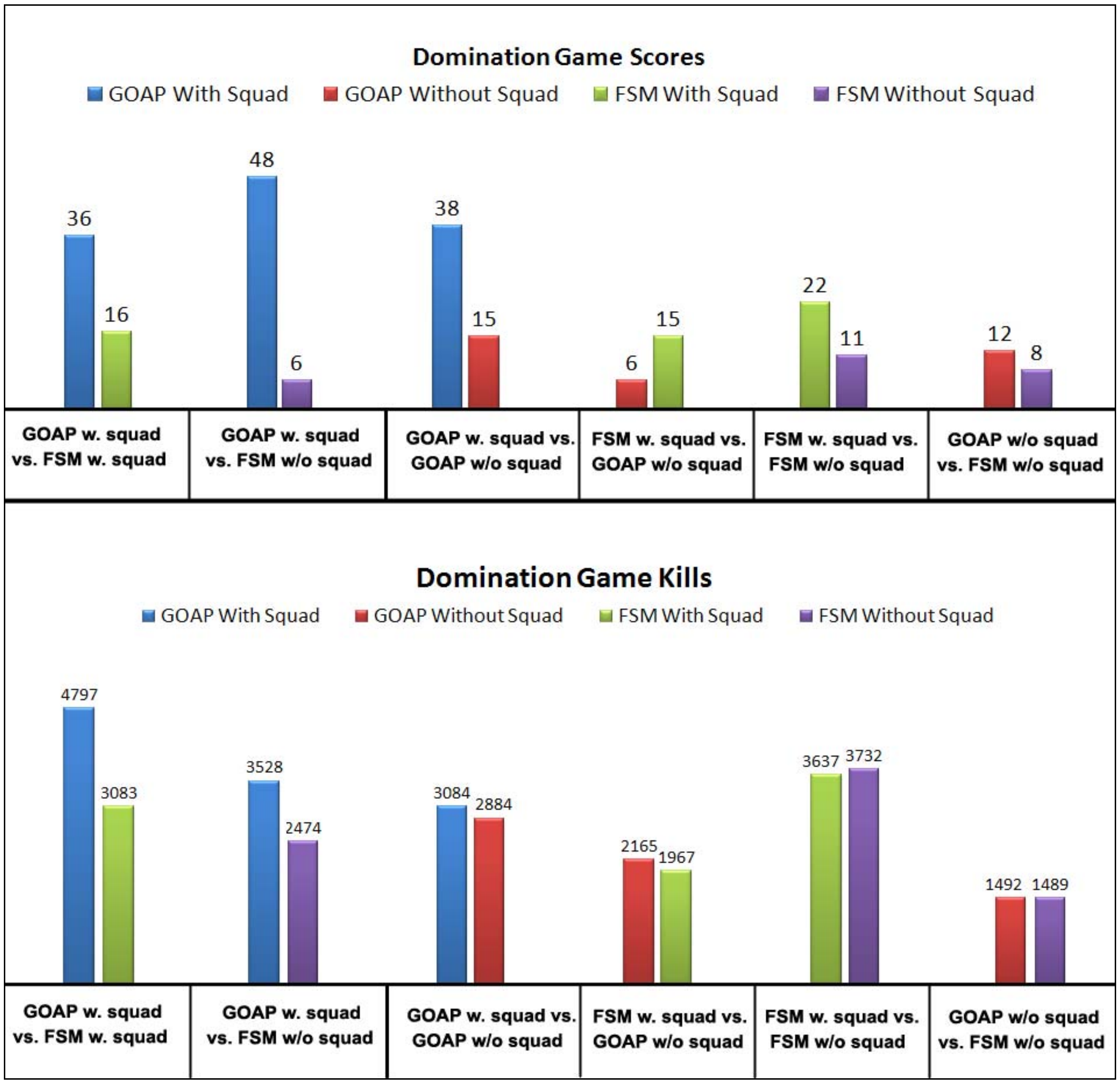


Figure 14 Domination Game Test Results

The distance between the team members was also recorded during the Domination and Capture the Flag experiments to determine the effect of the squad behaviour. Figure 15 highlights the fact that when squad behaviour was enabled the team members stayed much closer to one another on average. Given that the actual game map was 256x256, the difference between the distances with and without squad behaviour is quite large; agents can be brought approximately half the map width closer to one another with squads enabled. This goes some way to explaining as to why teams with squad behaviour enabled performed so well against teams without. Agents that stay so close to other squad members would back each other up and move to achieve goals in a coordinated fashion thus being more successful than individual agent's attempting to accomplish goals all on their own.

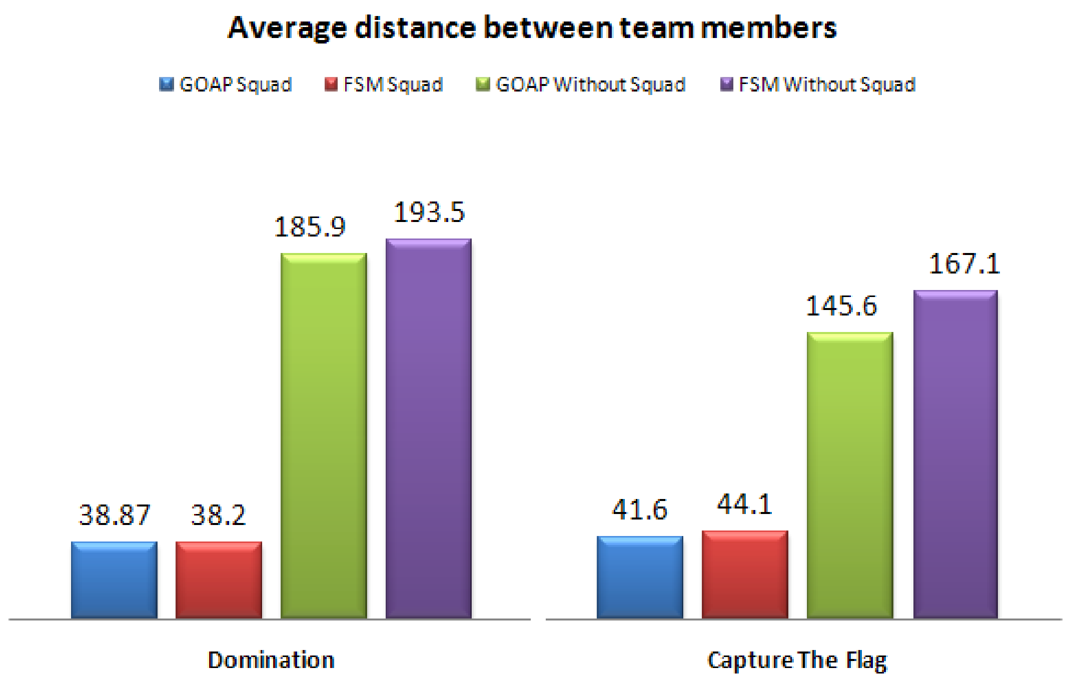


Figure 15 Average Distances between team members

Both the systems had the exact same architecture and functions in the subsystems available to them, there wasn't a single function that wasn't available to both the AI systems from the subsystems. Each of the FSM states directly corresponds to an action in the GOAP system so the GOAP system didn't have any extra functionality. Even though figure 15 shows that the distance between team members marginally favours GOAP, the FSMs were substantially out-performed by GOAP in both the Capture the Flag and Domination games with squad behaviour enabled. GOAP without squads even scored higher against the FSM with squads in the Capture the Flag game. So what explanation is there for the GOAP superiority as shown in the results? The first possible reason goes back to the results from the Deathmatch and Last Man Standing tests. The GOAP system just acting without squad commands was able to stay alive longer and kill better than the FSM. So when attempting to satisfy a game mode's goal using squad commands obviously an agent that can stay alive longer and kill better has a higher chance of successfully completing the order or goal.

Another reason that the GOAP system out-performed the FSM system is due to the fact that the FSM is reactive and if it receives an order, more often than not it changes state immediately to follow that order. However, the GOAP system weighs up possible options by evaluating the relevancy of *all* its goals, the order may affect the relevancy of a goal but the GOAP system may decide to take some other course of action instead and come back to the goal that satisfies the order at a later stage. This prioritisation of goals can bring about unexpected behaviour in the GOAP system and is one explanation as to why the GOAP system performs so well in the experiments.

Human error could be another factor as to why the results favour the GOAP system. Devising all the state transitions for the FSM system was tricky even for the few states that were in this game and it is a possibility that the state transitions were maybe incomplete or even incorrect. This is one of the most common complaints encountered by AI developers when using FSMs and becomes especially difficult to manage when the number of states increases (Isla, 2007). The state transitions didn't need to be created for the GOAP system so it wasn't an issue.

Orkin hints at one reason why the GOAP system could out-perform a FSM explaining that GOAP is declarative and FSMs are procedural (Orkin, 2005). The GOAP system forms a chain of actions to solve a goal while FSMs are reactive and can only flip from state to state, they cannot plan out what states they will go to next in advance. The GOAP planning combined with the dynamic replanning could have helped in bringing about the GOAP dominance in the results.

Finally, the GOAP system threw up several surprises during testing. For example, when an agent is patrolling, it can only break out of its control pattern if it senses or sees an opposing agent. The agent would then proceed to attack the target and once finished, it should immediately go back and into its original patrol pattern. However, occasionally the agent would unexpectedly go and search for health or search for ammo or even a Domination point instead of going back patrolling as was expected of it. This along with other unexpected behaviour wasn't directly coded into the GOAP system but occurred entirely through the planning process and provides another possible reason as to why the GOAP system out-performs FSMs.

So overall, the results of the experiments indicate that GOAP is the superior AI system and integrates better with the squad behaviour than the FSM system for the game scenario.

The next step when comparing the FSM and GOAP systems was based on the results from the technical experiments. Memory, CPU usage and average frames per second were recorded during the tests using VTune. The average percentage processor time used for the GOAP system was 33.62% while the average for the FSM was 35.44%, 1.82% of a difference. However the GOAP system had a minimum processor usage of 2.34% and a maximum of 98.44% while the FSM had a minimum of 0.05% and a maximum of 71.09%. The GOAP system approached 100% processor usage more often while the FSM was comfortably short of this limit. On occasion during the experiments the GOAP system would actually peak when several GOAP agents happened to plan at the same time (i.e. if a game suddenly became active) and the application would be visibly affected by this whereby frame rate would drop significantly. This problem never occurred with the FSM system.

The VTune results also indicate the average bytes available to the system throughout program execution. The FSM again out-performs the GOAP system in this regard, the FSM had on average 779,716,480 bytes available while the GOAP system had the lower value of 777,914,432 on average, 1,802,048 bytes or 1.8MB of a difference. This makes sense as the GOAP system allocates memory for the goal manager, the action container, the GOAP planner, the agent GOAP goals, the GOAP actions and the GOAP plans while the FSM system only allocates memory for the FSM states and the FSM machine.

The average number of plans per agent is plotted against the average number of times states changed per agent in figure 16. These were measured during the Domination and Capture the Flag experiments between the two AI systems with squads enabled. The actual checks performed when searching for state transitions are almost the same as the checks performed when determining the goal's relevancies and checking action's context preconditions. In general, the same querying of the working memory and blackboard was carried out for state transitions and when planning occurs. As the GOAP system forms far more plans, then this means that similar checks were being run far more often in the GOAP system than the FSM system (especially for the Capture the Flag game). So obviously the GOAP system is going to be less efficient.

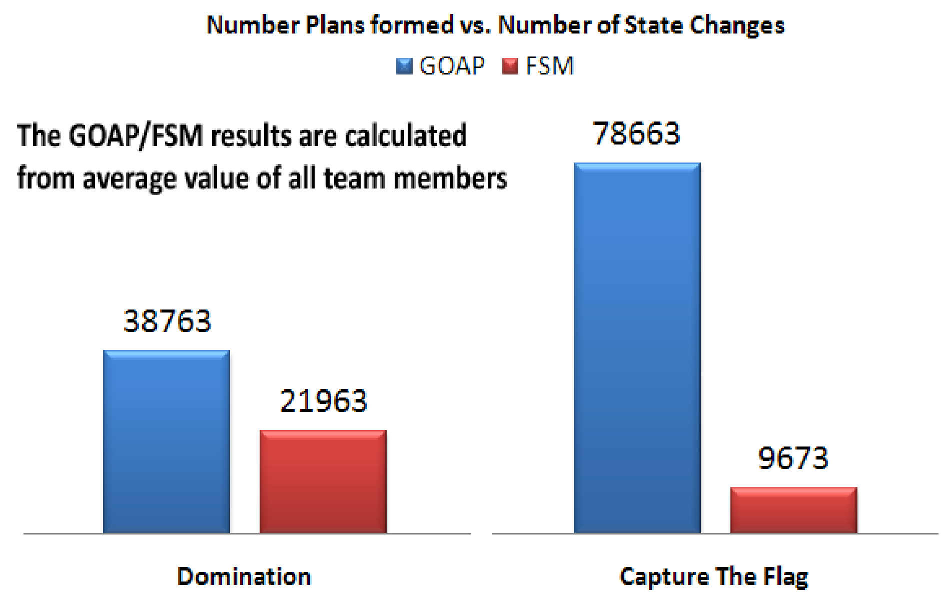


Figure 16 Number of plans formed vs. Number of state transitions

The average FPS was also monitored during the experimental phase. The average frame rate from the GOAP system (without squads enabled) was 439 frames per second while the average from the FSM system (also without squads enabled) was 464 frames per second. This difference in frame rate emphasises the difference in performance between the two AI techniques.

The technical results clearly show that the FSM system provides better performance than the GOAP system with better frame rates, memory management and CPU usage.

Finally, the ease of management, flexibility and reusability of GOAP and FSMs was measured by extending the two systems to incorporate new game modes, new behaviour and different agent types. The first test was to increase the number of game modes. As discussed in the Implementation section, the GOAP system required far fewer changes to allow it to function with the new game modes than the FSM system. The Deathmatch and Last Man Standing game modes required changes to be made in several FSM states while the GOAP system simply required a new XML file for each game mode. The Capture the Flag game mode required several of the FSM states to be duplicated and rewritten along with new states added. The GOAP just required two new goals, four new actions and a new XML file.

An extra action and state were added to the game late on. The BlindFire action is triggered when an agent is being damaged and doesn't know where it is being hit from. The agent fires randomly in any direction once activated. Including this into the GOAP system required just defining an extra action and placing the action into the agent's action set. The FSM system required a new state to be created along with extra transition logic placed in the Attack, Idle, GoToAmmo. Patrol, GoToHealth and Melee states to inform the FSM when to change to the BlindFire state. Again this illustrated the flexibility along with the ease of management of the GOAP system. Actions and goals can be easily added or removed from GOAP system whilst the high amount of coupling intrinsic in the FSM system makes this task far more difficult. The FSM needed the whole system to be recompiled again due to the coupling between all the states. This highlights another major benefit provided by GOAP as illustrated by figure 17. The actions and goals just need to be defined and there is no explicit link between the two, the GOAP system establishes the links at runtime. Links exist between FSM states which are hard-coded into the C++ code pre-runtime and this doesn't facilitate new states or changes to existing ones.

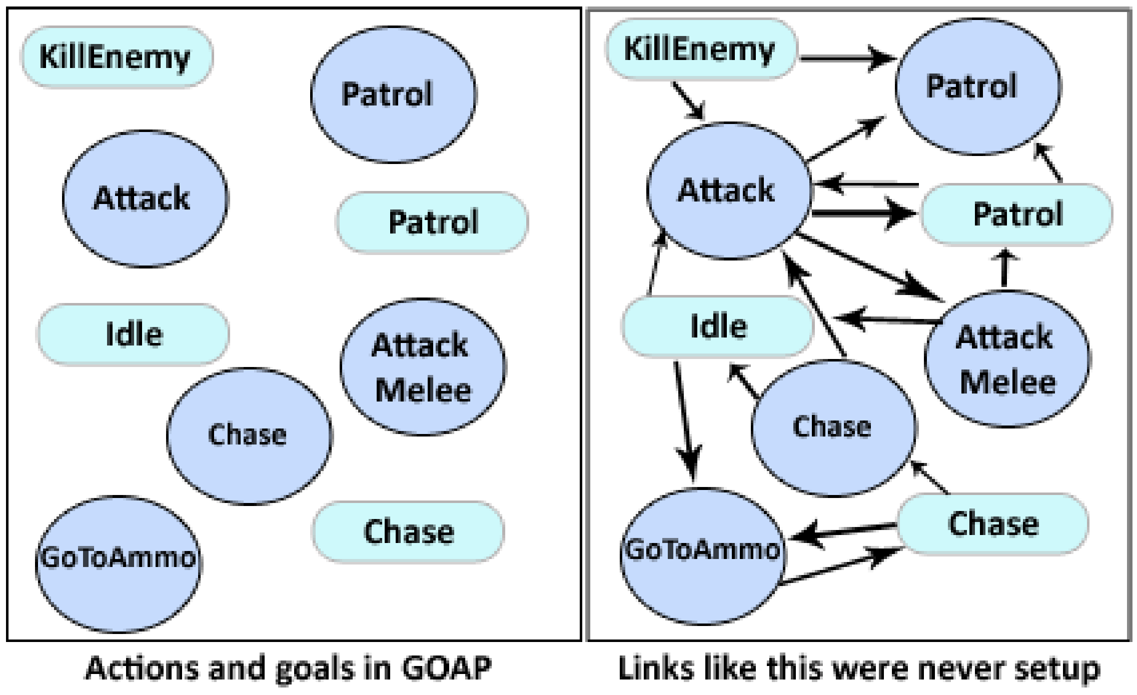


Figure 17 Links between actions formed at run-time for GOAP

GOAP actions and goals are decoupled and modular so it is only the new actions or goals that need to be compiled to incorporate any changes. As a modern game can have hundreds of classes and compilation time can run into hours, reducing compile times and extensive recompilations is highly desirable.

It was possible to highlight the reusability of the GOAP system by creating three different types of agent. The first type of agent had all the GOAP actions and FSM states but couldn’t melee attack, the second agent had all the GOAP actions and FSM states but couldn’t fire, only perform melee attacks. The final agent had all the GOAP actions and FSM states available to it. When implementing this with the GOAP system, as all the same actions were used across all the agents, each of the three agent types just required an XML file with a different action and goal set defined. When the agent is being loaded, the XML indicates to the GOAP system what the agent's capabilities are. However performing the same task with the FSM system requires different state machines and states for each agent, increasing the complexity and code repetition quite a bit. Perhaps if a data-driven FSM implementation were used that converts an external file into code, then FSMs could be as flexible as GOAP at creating different agents. The extra difficulty involved in getting an external FSM language setup alongside a compiler/interpreter makes this task a little more complex than simply defining an XML file.

The ease of creating different types of AI opponents is obviously a major benefit of the GOAP system to developers. Most modern games start with a gentle learning curve where players play against easy opponents at the beginning of a game and the AI opponents get more difficult as the player progresses. The steps above illustrate how this could be done for the GOAP system and this is much cleaner than with the FSM. This reduces repetition of code and development time for the game studio.

The GOAP system is also highly re-usable. Once created for the first time, the system can be brought into another project easily. The project would need to have new goals and actions created but the system would function fine as long as the A* machine was also brought across with it. A GOAP system can be easily transferred across to be used for a completely different genre e.g. an RTS game without any changes whatsoever to the actual system, the actions and goals bring about the different functionality. The FSM states however cannot be brought across to another project.

Orkin explained that the GOAP system can improve the workflow between game designers and programmers (Orkin, 2004). The designers just need to concentrate on the external data files to define the goal and action sets, action costs and goal relevancies while the programmers can concentrate on implementing the AI system. This is a change from the traditional implementation of FSM systems where the designers and programmers often need to work very closely with one another determining the actual states and the transitions between them (Orkin, 2004).

The layering of behaviour is something that Orkin highlights as a benefit of GOAP (Orkin, 2005). This project layered attack actions on top of a base attack action and while it worked quite well, the same approach could be taken by the FSMs. Houlette explains 'Polymorphic FSMs' (Houlette, 2003) which layer FSM states on top of one another in a similar way to what was done for the GOAP system.

Aside from the technical results, the rest of the comparisons have been wholly in favour of the GOAP system. However there are drawbacks involved with development of a GOAP system that the results didn't highlight.

Debugging the GOAP system proved to be far more challenging than debugging the FSM system during development. There is much more to keep track of and it is very unpredictable. There were times during testing when agents would just stay in the same position as they wouldn't have any goal active or be executing any action. To debug this, the first task was to identity each possible goal that could be active and see why none were forming plans. Even if a goal is relevant and selected for planning, it may be the case that actions which match the goal fail the checks required for the planner to select them and thus no plan was built and the goal is consequently skipped. This can continue until there were no goals left to check, a plan wasn't formed and the agent had no action to execute. The next debugging step was to check the context preconditions of all available actions that map to the active goal and find out what may be stopping the planner from selecting the action. Even if a plan was built successfully, something may be occurring once the action was activated that caused the whole plan to be invalidated immediately or very soon after activation. So further debugging checks were required to verify the actions activation and validation functions. All of these checks along with other issues faced while debugging proved troublesome for this project and given that F.E.A.R. has 71 goals and 121 actions, these issues could rapidly become time-consuming and harm productivity if not kept in check. Debugging the FSMs involved monitoring the active state and checking why certain state transitioning logic wasn't triggering if an agent was stuck in a state. The problem of not having any goal/action active couldn't happen for the FSM as it has to be in a state at all times.

Another drawback with GOAP lies in the difficulty implementing it. Buckland's FSM was implemented in two header files and the programmer just has to implement the states. The GOAP system requires a goal manager, planners, A* goal, A* map, action container before a start can be made on the goals and actions. The FSM states were created far quicker than the GOAP systems. This was partly due to that fact that they were created after GOAP when all the subsystems were working properly but also because there was less potential to go wrong during implementation. Problems with any of the GOAP systems listed above can derail the entire system while the simplicity of the FSM implementation means only the states can cause problems. Issues within the FSMs can quickly be located while there needed to be a good deal of searching around to pinpoint the relevant buggy module for the GOAP system.

Chapter 6. CONCLUSIONS

The results of the experiments clearly highlight that GOAP is a superior AI system than FSMs aside from the technical comparison results. Modern games require that every component be as efficient and optimised as much as possible, especially for games consoles. This reason, combined with the other drawbacks listed at the end of the discussion section may begin to explain why GOAP has not yet become a more prevalent AI choice especially over FSMs. FSMs are a tried and tested AI technique and combined with their simplicity, less processing and memory overhead and relatively quicker development time mean that there may be no desire from companies to change to GOAP.

However if the GOAP could be optimised to operate at acceptable levels (F.E.A.R. was obviously able to do this) the experimental results highlight that the benefits of GOAP make it a more attractive system than the FSMs, both from a developers and player perspective. The unpredictable and superior (as shown in the results) behaviour can provide a challenging AI opponent to a player whilst the difficulty of the AI system can easily be tweaked to suit less talented players. Further tests are needed to truly gauge the system's value to players. From a developer's perspective, the GOAP system integrated better with the squad system, scored higher in the tests and proved to be far more flexible, maintainable, reusable and easier to manage than FSMs. The implementation and testing phases also highlighted some of the general benefits that Orkin has emphasised about the GOAP system (Orkin, 2005) i.e. dynamic re-planning and decoupling of goals and actions.

In summation, GOAP's benefits substantially out-weigh its short-comings, it is a far superior system to FSMs and surely it is only a matter of time before more developers begin to adopt it as a primary means for controlling NPC behaviour. This dissertation has added its results and findings to the limited pool of public knowledge relating to the domain of AI planning in games. No other known project has previously taken a GOAP system and compared it with any other AI system and so the findings of this project are a first in the field of GOAP in games. There are presently no open-source AI planners available for games and as a result of this project, a standalone GOAP planner was developed which in the future could be released to the general public which would allow it to be extended and improved upon.

Though the research and implementation of this project answered the research question that was posed, there is a great deal more work that can be done in the future not just continuing the work from this project, but also in the fields of GOAP and AI planning in general.

6.1 Future work

Although this project researched how GOAP can improve the AI of an NPC by testing against FSMs, another way to test if it is superior to FSMs is to perform extensive play testing. Test subjects could play against GOAP and FSM opponents and surveyed on which they felt was better. Due to the subjective nature of this test, large numbers of test subjects would be required to obtain adequate results. Ultimately, games are created to satisfy customers so this is the next logical step in determining if GOAP really is an improvement over FSMs from a player's perspective.

Another extension of the project would be to actually modify Unreal Tournament 2005 and place the different AI agents into the real Domination, Deathmatch, Capture the Flag and Last Man Standing games modes in UT2005 rather than the 2D simulation that was created for this dissertation. Combining the systems alongside complex 3D graphics, physics and pathfinding would get far more accurate results especially from the technical experiments as they would reflect modern game requirements more closely.

One way by which the finite state machine could be improved upon is to extend the FSM system to become a fuzzy state machine (FuSM). One of the criticisms regularly levelled at FSMs is the predictability of the state transitions (Brownlee, 2002). FuSMs can make it more difficult to predict these state transitions by using fuzzy variables to influence state changes. Fuzzy state machines can consider several states to be the current state, if not all of them. Each state is given a value between zero and one and the state with the highest value is chosen to be the current state. States are assigned values by the fuzzy logic system which takes in various inputs, fuzzifies them, applies fuzzy rules and defuzzifies to obtain a value for the state. However, while FuSMs may sound good in theory, Houlette (Houlette, 2003) indicates that there is little information beyond theory and small pieces of literature detailing implementations of them available. Hierarchical and polymorphic FSMs could also be employed to further increase the functionality of the FSM.

FSMs were the only other AI technology that was compared with GOAP during this project. FSMs may not be the best AI technology available at present but they remain highly popular. A further extension to the experiments carried out for this project could be to investigate and compare other AI techniques with GOAP. Technologies such as fuzzy logic, scripted AI, neural networks, rule based systems, Bayesian techniques and genetic algorithms are amongst the techniques that could be created as comparative AI.

The GOAP system that was developed used only Boolean symbols for the world state symbols. If the system were extended to incorporate variable world state symbols then this could massively increase the power of the system. As Orkin explains in his discussion of GOAP (Orkin, 2005), variables allow for the passing of values between different goals and actions. Although the inclusion of variable support was a little out of the scope for this project, future work could design an extension to the system that would allow for their use.

Using world state symbols when combined with action's preconditions and effects can have its limitations. Suppose there is an action with preconditions weaponLoaded= true and targetInSights = true. Then the action will only trigger when both weaponLoaded AND targetInSights world state symbols are true. A nice extension to the GOAP system could be to include other Boolean operators between preconditions and effects of actions such as OR, XOR, NAND etc. So an action could trigger if weaponLoaded is true OR if targetInSights is true.

The squad tactic design for this project was principally an ad-hoc approach whereby different behaviour was created by passing orders into squad members from the squad manager. Different squad managers were then developed for different game modes. While this approach worked well in this case, commercial games may require something a little more structured. Both Orkin and Munóz-Avila recommend Hierarchical Task Networks as a way of applying a formalised planning system to squad behaviours. HTNs plan at a higher level than a GOAP system using tasks rather than goals. Each task is either a compound or primitive task. A compound task is made up of several primitive tasks and a primitive task equates to a concrete action. HTNs use methods which define how to achieve a compound task. HTN planning reasons on higher level strategies and concentrates on accomplishing methods using tasks rather than directly selecting actions to execute. HTNs are a very new field of research in games and there is little information available about how to incorporate them at present. However there is research being undertaken at the moment into their use and if included, could immensely improve the squad behaviour in the future.

The agent architecture formed a quite significant part of the GOAP system but the biggest problem encountered throughout development was the working memory cleanup. If it wasn't cleaned up regularly enough, the working memory became clogged with useless facts. If done too often, then the working memory was emptied of working memory facts as they were deleted too soon. Further work could include devising some sort of cleanup solution as if this issue was overcome, distributed processing, caching and hence efficiency of the planner would be greatly improved. The working memory only allows for certain types of facts. In the future it could be useful if the working memory was to be extended to allow for any type of fact or if the working memory wasn't just restricted a certain structure of working memory fact.

Action cost and goal relevancy values were coded directly into the application using C++ during the project. However in the F.E.A.R. system this data was contained in an external database which is read throughout the application execution. Creating a similar external database that is read in at the initialisation of the application which contains not only the cost and relevancy values but also agent's action and goal sets could be another addition to the GOAP system. It would improve compile times as changes to the cost and relevancy values would be external to the application. One criticism of the existing GOAP system is that action costs are static values; they never change throughout application execution. If a context sensitive method of altering action cost values were devised, behaviour in the system could be improved further. For example, if an agent is close to a health pickup, the cost of the GoToHealth action could be reduced while the cost of the GoToAmmo action could be increased, making the GoToHealth action more attractive to the A* machine. The database could store these values so that they could be subsequently obtained and updated at a later stage.

The first-person shooter genre is the only genre publicly known so far to have used GOAP in any shape or form. However other genres of games are actually more suited to using GOAP. For example, take a real-time strategy game like Civilisation where the AI has several plans on-going at the same time. It must plan what buildings to put on the map and where, what units to build, where to move them to and can also have economic and strategic plans. The GOAP system would be ideal for this kind of scenario whereby a single planner could be used by the different systems to plan out an AI's course of action. Other game genres whereby GOAP could be used include sports games, simulation type games like The Sims or even puzzle games.

APPENDICES

APPENDIX A

The A* Algorithm

Below is the A* algorithm used in this project.

```
BeginAStar
{       currentNode = createANode( goalNode )
        Storage.AddToOpenList( currentNode )
        h = aStarGoal.GetHeuristicValue( currentNode, aStarMap )
        currentNode.g = 0
        currentNode.h = h
        currentNode.f = h
        while( true )
        {       currentNode = storage.RemoveCheapestOpenNode()
                Storage.AddToClosedList( currentNode )
                if( aStarGoal.IsAStarFinished( currentNode )
                {       break
                }

                Neighbours = aStarMap.GetNeighbours( currentNode )
                for each neighbourID
                {       if( storage.IsInOpenList( neighbourID ))
                        {       neighbour = storage.FindInOpenList( neighbourID )
                        }
                        else if( storage.IsInClosedList( neighbourID ))
                        {       neighbour = storage.FindInClosedList( neighbourID )
                        }
                        else
                        {       neighbour = aStarMap.CreateNewNode( neighbourID )
                        }

                        g = currentNode.g
                        g += ( aStarGoal.GetActualCost( neighbourID, currentNode ))
                        h = aStarGoal.GetHeuristicValue( neighbourID )
                        f = g + h
                        if(  f > neighbour.f )
                        {       break
                        }

                        Storage.AddToOpenList( neighbourID )
                        neighbour.parent = currentNode

                }
        }
}
```

APPENDIX B

Goals and actions created

Below are the goals and actions created for the GOAP system.

ACTIONS

Action: ActionAttackDomPointOne

Preconditions: weaponLoaded = true

Effects: atDomPointOne = true

Context Preconditions: Must have a loaded weapon, no valid target and no conflicting orders.

Action Activation: Places the first Domination point as the new navigation target on the blackboard.

Action Completes: When navigation is complete or if the Domination point is overtaken.

Action: ActionAttackDomPointTwo – as above except for the second domination point.

Action: ActionAttack

Preconditions: inWeaponsRange = true, weaponLoaded = true

Effects: targetIsDead = true

Context Preconditions: Has a valid loaded weapon and a valid target.

Action Activation: Places a command on the blackboard to begin an attack.

Action Completes: When the target is dead or if the attack is complete.

Action: ActionAttackLongRange

Preconditions: inWeaponsRange = true, weaponLoaded = true

Effects: targetIsDead = true

Context Preconditions: Has a loaded long range weapon equipped and a valid target.

Action Activation: Places a command on the blackboard to begin an attack whilst manoeuvring to stay in long range.

Action Completes: When the target is dead, target is out of range or if the attack is complete.

Action: ActionAttackShortRange

Preconditions: inWeaponsRange = true, weaponLoaded = true

Effects: targetIsDead = true

Context Preconditions: Has a loaded short range weapon equipped and a valid target.

Action Activation: Places a command on the blackboard to begin an attack whilst manoeuvring to stay in short range.

Action Completes: When the target is dead, target is out of range or if the attack is complete.

Action: ActionAttackMelee

Preconditions: inMeleeRange = true

Effects: targetIsDead = true

Context Preconditions: Have no weapon loaded and has a target in melee range.

Action Activation: Activates a melee attack.

Action Completes: When the target is dead, target is out of melee range or if the attack is complete.

Action: ActionChangeWeapon

Preconditions: None

Effects: weaponLoaded = true

Context Preconditions: Current weapon isn't loaded and there exists another weapon with ammo.

Action Activation: Places a request on the blackboard to change weapons.

Action Completes: When the agent has a loaded weapon.

Action: ActionGoTo

Preconditions: None

Effects: atTargetNode = true

Context Preconditions: Agent has a GoTo order.

Action Activation: Places the order's GoTo position as the navigation target on the blackboard.

Action Completes: When the agent completes navigation.

Action: ActionGoToAmmo

Preconditions: None.

Effects: atTargetNode = true

Context Preconditions: Has no ammo.

Action Activation: Places the position of the closest ammo pickup as the new navigation target on the blackboard.

Action Completes: When the agent has ammo.

Action: ActionGoToHealth

Preconditions: None.

Effects: atTargetNode = true

Context Preconditions: Has low health or a low health event in working memory.

Action Activation: Places the position of the closest health pickup as the new navigation target on the blackboard.

Action Completes: When the agent doesn't have a low health event or low health.

Action: ActionIdle

Preconditions: None.

Effects: atTargetNode = true

Context Preconditions: Agent has no orders, events or target

Action Activation: Enables roaming on the blackboard.

Action Completes: When an event, order or target is sensed or if the game becomes active.

Action: ActionPatrol

Preconditions: None.

Effects: atTargetNode = true

Context Preconditions: If a patrol order exists or if the agent is within range of a patrol location.

Action Activation: Enables patrolling.

Action Completes: If patrolling location is overtaken or if the agent has a target.

Action: ActionChase

Preconditions: None.

Effects: inWeaponsRange = true

Context Preconditions: Agent has a target that is out of weapons range.

Action Activation: Requests the navigation manager to pursue the target.

Action Completes: When the target is lost, dead or in weapons range.

Action: ActionDodge

Preconditions: None.

Effects: atTargetNode = true

Context Preconditions: Agent is taking damage and is in melee range.

Action Activation: Requests the navigation manager to dodge the damage it is taking.

Action Completes: When the navigation manager has finished moving.

Action: ActionDodgeFire

Preconditions: None.

Effects: atTargetNode = true

Context Preconditions: Agent is taking damage and is outside melee range.

Action Activation: Requests the navigation manager to dodge the damage it is taking.

Action Completes: When the navigation manager has finished moving.

GOALS

Goal: GoalAttackDomPointOne

WS Satisfaction: atDomPointOne = true

Goal Relevancy: Goal is relevant if the agent has orders to attack the first Domination point, has an event that indicates all other Domination points are taken or if Domination point isn't controlled by agent's team.

Goal: GoalAttackDomPointTwo – as above except for the second dom point.

Goal: GoalDodge

WS Satisfaction: atTargetNode = true

Goal Relevancy: Relevant if the agent is being damaged or if it is in an enemy's sights and in melee range.

Goal: GoalPatrol

WS Satisfaction: atTargetNode = true

Goal Relevancy: Relevant if the agent has an order to patrol or if close to a Domination point that is owned by its team.

Goal: GoalFindHealth

WS Satisfaction: atTargetNode = true

Goal Relevancy: Relevant if the agent has low health and isn't being attacked.

Goal: GoalFindAmmo

WS Satisfaction: weaponLoaded= true

Goal Relevancy: Relevant if the agent's current weapon isn't loaded.

Goal: GoalGoToTask

WS Satisfaction: atTargetNode= true

Goal Relevancy: Relevant if there is a go to order.

Goal: GoalIdle

WS Satisfaction: atTargetNode= true

Goal Relevancy: Relevant if the game isn't active and the agent hasn't any orders, events or targets.

Goal: GoalKillEnemy

WS Satisfaction: targetIsDead= true

Goal Relevancy: Relevant if the agent has a target.

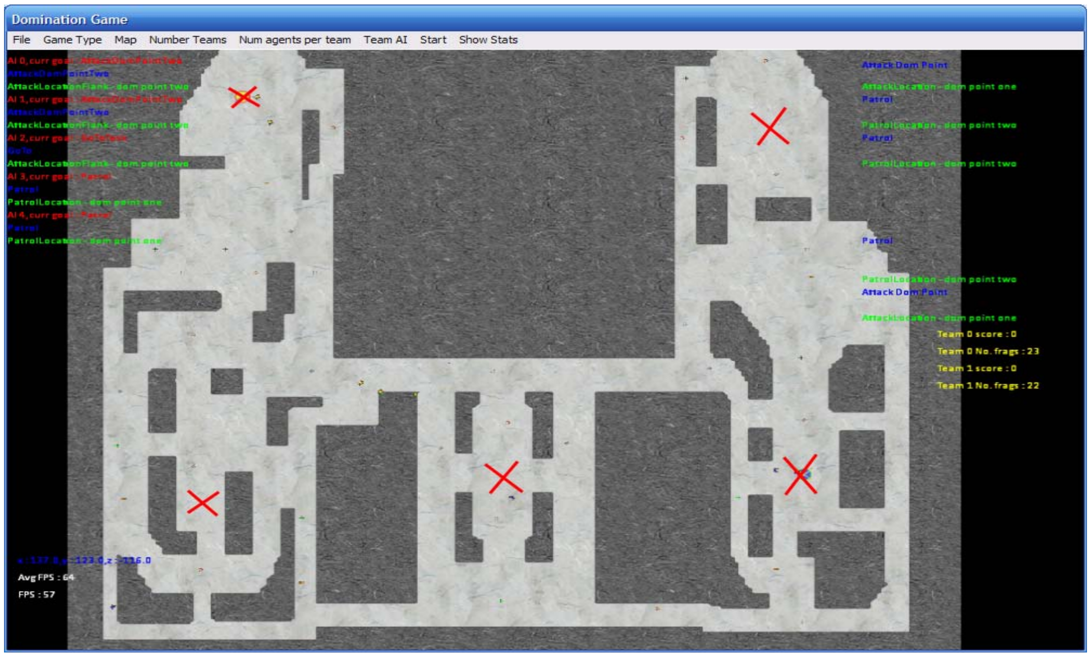
The states that were developed for the FSM were: AttackDomPointState, AttackState, ChangeWeaponState, ChaseState, DodgeState, GoToAmmoState, GoToHealthState, GoToState, IdleState, MeleeState, PatrolState, ReloadState. BlindFireState. The details of these states are much the same as the matching GOAP actions.

Extra GOAP goals and actions were created for the extra game modes. These were: ActionAttackFlagCarrier, ActionCarryFlag, ActionPickupFlag, ActionProtectFlagBearer, ActionSeekClosestEnemy, GoalCaptureFlag, GoalReturnFlagToBase, ActionBlindFire.

APPENDIX C

The game

Two maps were created for the game where the AI can be placed in to carry out the tests. Below are screenshots of the game, the first is of the Domination game in the first map and the second is a Capture the Flag game in the second map. On the first screenshot the left hand side indicates what current GOAP goal and action is active for each agent along with the agent's current squad command. The right hand side displays the FSM agent's current state along with its squad's current command. Every effort was taken to be as faithful as possible to the original UT2005 Domination and other game modes implemented as part of the project. The maps were based on actual UT2005 maps which were obtained from the Unreal Tournament editor. The red 'x' indicates where the possible Domination or Capture the Flag locations can be; these were randomised for each test. Agent's have over forty positions that they are restarted from once killed and it was entirely random which one was chosen. Pickups such as ammo and health are strewn around each level and are spread out as much as possible. Each agent has four weapons (a pistol, shotgun, rocket launcher and minigun) with no ammunition and a health of one hundred starting off.



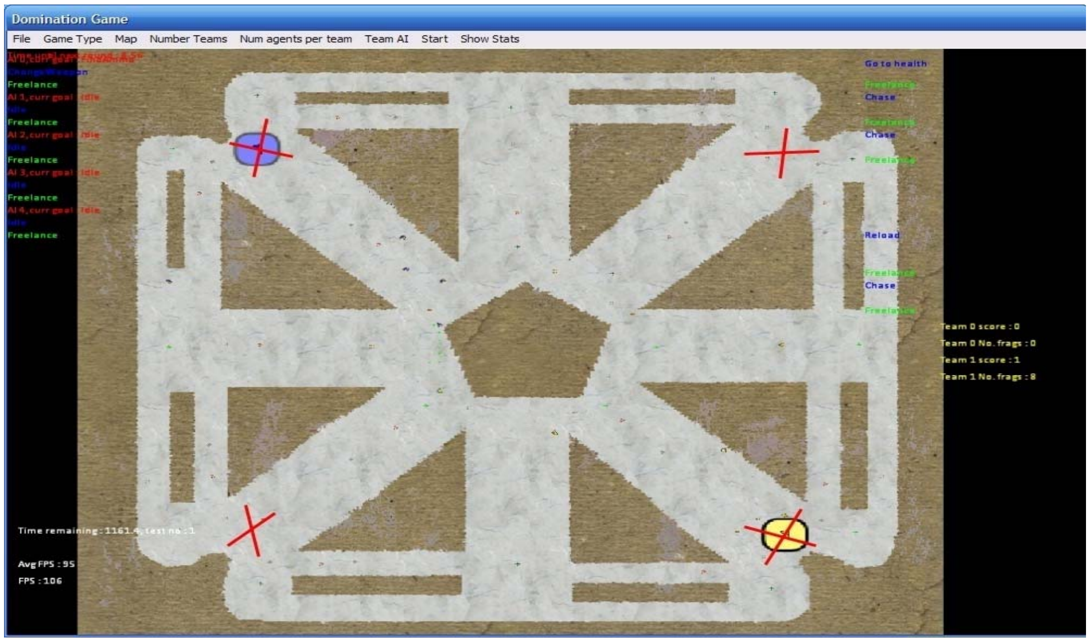
```
AI 0,curr goal : AttackDomPointTwo
AttackDomPointTwo
AttackLocationFlank - dom point two
AI 1,curr goal : AttackDomPointTwo
AttackDomPointTwo
AttackLocationFlank - dom point two
AI 2,curr goal : GoToTask
GoTo
AttackLocationFlank - dom point two
AI 3,curr goal : Patrol
Patrol
PatrolLocation - dom point one
AI 4,curr goal : Patrol
Patrol
PatrolLocation - dom point one
```

```
x : 137.0,y : 123.0,z : -116.0
Avg FPS : 64
FPS : 57
```

```
Attack Dom Point
AttackLocation - dom point one
Patrol
PatrolLocation - dom point two
Patrol
PatrolLocation - dom point two
Patrol
PatrolLocation - dom point two
Attack Dom Point
AttackLocation - dom point one
```

```
Team 0 score : 0
Team 0 No. frags : 23
Team 1 score : 0
Team 1 No. frags : 22
```

Figure 18 Map 1 developed for the game



```
[illegible]
ChangeWeapon
Freelance
AI 1,curr goal : Idle
Idle
Freelance
AI 2,curr goal : Idle
Idle
Freelance
AI 3,curr goal : Idle
Idle
Freelance
AI 4,curr goal : Idle
Idle
Freelance
```

```
Time remaining : 1161.4, test no : 1
Avg FPS : 95
FPS : 106
```

```
Go to health
Freelance
Chase
Freelance
Chase
Freelance
Reload
Freelance
Chase
Freelance
```

```
Team 0 score : 0
Team 0 No. frags : 0
Team 1 score : 1
Team 1 No. frags : 8
```

Figure 19 Map 2 developed for the game

REFERENCES

REFERENCES

Bridge, D. (2007). AI Planning. Retrieved June 11, 2007, from Computer Science UCC: http://www.cs.ucc.ie/~dgb/courses/tai/notes/handout34.pdf

Brownlee, J. (2002). A finite state machine framework. Retrieved 13 August, 2007, from AI Depot website: http://ai-depot.com/FiniteStateMachines/FSM-Background.html

Buckland, M. (2005). *Programming game AI by example pp. 43-88* .Wordware Publishing.

Burke, R et al. (2001). Creature smarts: The art and architecture of a virtual brain. In the proceedings of the Game Developers Conference.. San Jose, CA. 1(Jul): pp. 147-166.

Higgins, D. (2002). A generic A* machine. *In AI Game Programming Wisdom pp. 133-145*. Charles River Media.

Houlette, R. (2003). The ultimate guide to FSMs in games. *In AI Game Programming Wisdom 2 pp. 283-302.* Charles River Media.

Isla, D. *Handling complexity in the Halo 2 AI.* Retrieved June 11, 2007, from Gamasutra: http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml

Lester, P. A* Pathfinding for beginners. Retrieved June 11, 2007, from the GameDev website: http://www.gamedev.net/reference/programming/features/astar/

Long, D. et al. (2003). *Working group on Goal-oriented Action Planning*. AIISC.

Munóz-Avila, H., and Lee-Urban, H. (2005). Hierarchical plan representations for encoding strategic game AI. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference 2005.* 1(2005): pp. 63-69

Nareyek, A. et al. (2005). *Working group on Goal-oriented Action Planning*. AIISC.

Orkin, J. (2002). Applying goal oriented action planning in games. *In AI Game Programming Wisdom 2 pp. 217-229*. Charles River Media.

Orkin, J. (2003). *Applying blackboard systems to FPS*. Austin, TX: Digital Media Collaboratory.

Orkin, J. et al. (2004). *Working group on Goal-oriented Action Planning*. AIISC.

Orkin, J. (2004). Symbolic representation of game world state: Toward real-time planning in games. *AAAI Challenges in Game AI.* 1(2006): pp. 41 - 46.

Orkin, J. (2005). Agent architecture considerations for real-time planning in games. *Artificial Intelligence and Interactive Entertainment Conference* 2005. 1(June): pp. 105-110.

Orkin, J. (2006). Three states and a plan, the AI for FEAR. *Game Developers Conference, San Jose,CA.* CMP Media.

BIBLIOGRAPHY

BIBLIOGRAPHY

Bourg, D. (2004). *AI for game developers*. O'Reilly Publishing.

Bridge, D. (2007). Search strategies. Retrieved 24th August 2007, from the Computer Science Department at University College Cork website: http://www.cs.ucc.ie/~dgb/courses/tai/notes/handout32.pdf

Buckland, M. (2002). *AI techniques for game programming*. Course Technology PTR.

Busby, J et al. (2004). *Mastering Unreal technology, the art of level design*. Sams Publishing.

Butcher, C., and Griesemer, J. (2002). The integration of AI and level design in Halo. *Game Developers Conference, San Jose, CA.* 1(2002): pp. 147-166., CA: Gamasutra.

DeLoura, M. (2001). *Game programming gems II*. Charles River Media.

Dybsand, E. (2002). Goal directed behaviour using composite tasks. *In AI Game Programming Wisdom 2: pp. 237-247*. Charles River Media.

Granberg, C. (2006). *Programming an RTS game with Direct3D.* Charles River Media.

Helmert, H. (2007). *Introduction to PDDL*. Retrieved June 11, 2007, from Qiang Yang's website: http://www.cs.ust.hk/~qyang/221/introtopddl2.pdf

Isla, D., Blumberg, B. (2002). New challenges for character-based AI for games. *AAAI Spring Symposium on AI and Interactive Entertainment*. 2(2002): pp. 41-45.

Luna, F. (2003*). Introduction into programming using DirectX 9.0*. WordWare publishing.

Magerko, et al.. (2004). AI Characters and Directors for Interactive Computer Games. *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference,* 1(Mar): pp. 877-883.

Mareas, M., and Stern, A. (2002). A behavior language for story-based believable agents. *IEEE Intelligence Systems*, 17(July): pp. 39-47.

Munóz-Avila, H., and Fischer, T. (2004). Strategic planning for Unreal Tournament bots. *In Proceedings of AAAI Workshop 2004 on Challenges in Game AI*. 82(9): pp. 22-26

Muñoz-Avila, H. (2007). *Classical AI planning.* Retrieved June 11, 2007, from Computer Science LeHigh University: http://www.cse.lehigh.edu/~munoz/publications.html

Nilsson, N., and Fikes, R. (1970). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(1971): pp. 189-208.

Nilsson, N. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann.

Obst, O., Maas, A., and Boedecker, J. (2005). HTN planning for flexible coordination Of multiagent team behavior. *In Proceedings of RoboCup 2005*: pp. 521-528

Smith, S., Nau, D., and Throop, T. (1998). Success in spades, using AI planning techniques to win the world championship of computer bridge. *AAAI National Conference:* pp. 1079-1086.

van Lent, M., and Laird, J. (1999). Developing an artificial engine. *Game Developers Conference 1999, San Jose, CA.* 1(Mar): pp. 577-588

Wallace, N. (2002). Hierarchical planning in dynamic worlds. *In AI Game Programming Wisdom 2. pp: 229-236.* Charles River Media.