To the Graduate Faculty:

I am submitting herewith a project written by David Lindsey Pittman entitled "Practical Development of Goal-Oriented Action Planning AI." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Software Development.

_____
Jeff Wofford, Faculty Supervisor

We have read this Project
and recommend its acceptance:

_____

_____

_____

Accepted for the Faculty:

_____
Executive Director, The Guildhall at SMU

PRACTICAL DEVELOPMENT OF GOAL-ORIENTED ACTION PLANNING AI

A Project Presented to the Graduate Faculty of
The Guildhall at Southern Methodist University
in
Partial Fulfillment of the Requirements
for the degree of
Master of Interactive Technology
in Digital Game Development
with
Specialization in Software Development
by

David L. Pittman

(BS, University of Nebraska-Lincoln, 2005)

March 21, 2007

Pittman, David L. BS, University of Nebraska-Lincoln, 2005

Practical Development of Goal-Oriented Action Planning AI

Supervisor: Professor Jeff Wofford

Master of Interactive Technology degree conferred March 23, 2007

Project completed March 21, 2007

As real-time graphics continue to approach realism and gameplay systems become increasingly complex, game AI must progress in parallel to maintain the illusion of believable characters. The decision-making architecture known as Goal-Oriented Action Planning (GOAP) offers dynamic problem solving, which is a key ingredient in crafting AIs which appear to understand their environment and react in a logical manner. This project analyzes the development of a GOAP-based AI and its application to various game genres, with consideration to the practicality of the technology. In this context, practical is defined as being able to be implemented in an interpreted scripting language at production quality in a reasonable amount of time, performing efficiently, and producing equal or better results compared to traditional AI techniques. In the context of certain game genres such as tactical shooters and strategy games, coordinated character behavior is as important as individual behavior to the success of an AI system. Prior implementations of GOAP have used informal solutions to the problem of coordinated decision-making. This project proposes a scheme for implementing command and control hierarchies which use GOAP at every level to propagate orders from a commanding unit to atomic AI agents.

ACKNOWLEDGEMENTS

In the completion of this project, I owe a debt of gratitude to the faculty at The Guildhall at SMU, including Gary Brubaker, Jeff Wofford, and Wouter van Oortmerrsen. The success of the project owes a great deal to their knowledge and guidance. My thanks also go to Jeff Orkin, who freely gave his time and wisdom to assist this project.

For their support and camaraderie, I would like to thank my literal and figurative brothers, the programmers of Cohort 5. My greatest thanks go to Kyle for always pushing me to greater heights, Kim for her love and understanding, and to my parents for their unending love and support (both emotionally and monetarily).

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Goal-Oriented Action Planning

As game graphics become increasingly realistic and believable, character artificial intelligence must continue to progress in parallel. Using static AI techniques in a modern game with lifelike models can create a jarring effect for the user (similar to the "uncanny valley" phenomenon described in the field of robotics [Mori70]). The player's suspension of disbelief is broken when a character behaves in a highly unnatural manner. Good game AI is not just about producing systems which can solve problems, it is about the compelling simulation of a living being. The AI should behave realistically and react dynamically to the player and to situations which may or may not have been specifically anticipated by a designer. Because game AI is a feature in which programmers, designers, and artists are all invested, it is desirable to develop an AI system that allows all disciplines to configure character behaviors in an intuitive and friendly way.

The purpose of this project is to evaluate the usefulness of the game AI technique known as Goal-Oriented Action Planning (GOAP) in terms of its cost to implement and runtime performance on an interpreted scripting language and its value for various game genres. Furthermore, the project proposes the use of GOAP for military-style "command and control" hierarchies, as might be found in a squad-based shooter or real-time strategy game. GOAP is a relatively new addition to the game AI vocabulary, having been introduced by the AIISC in 2003 [AIISC]. Fundamentally, the technique involves formulating a list of actions whose outcome will satisfy a goal.

GOAP is a technique for AI decision-making that produces a sequence of actions (called a plan) to achieve a desired goal state. The primary objects in GOAP are Goals and Actions. A Goal is described by relevant world state properties. An Action is described in terms of its effects on the world state and its world state preconditions. A plan of action (literally, an ordered array of Actions) is formulated by a component called a planner.

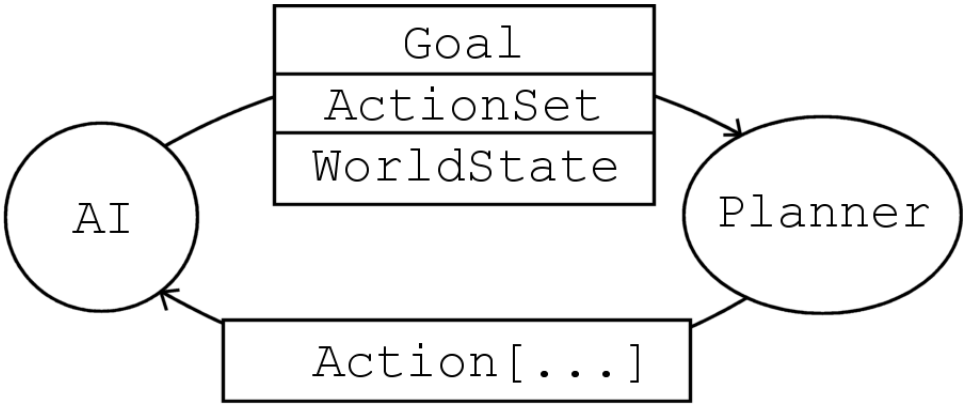

Figure 1.1 Relationship between primary GOAP classes

The input to the planner includes a current world state, a previously selected Goal, and a set of Actions. The Goal's properties are modulated with the current world state to produce the goal world state. The plan is then formulated by searching the state space graph for a minimal ordered set of Actions which produce the goal state from the current state. For our purposes, we can generally ignore world state side effects caused by Actions. This means that we actually only need to consider those properties of the world state which the Goal or an Action precondition specifies; all the other elements of the world state are ignored in the search.

Figure 1.2 An example of a plan formulated by GOAP

1.2 Command and Control Hierarchies

This project proposes a GOAP-based implementation of coordinated AI behavior using a command and control hierarchy. A command and control hierarchy is a model (typically associated with military forces) for distributing orders from a top-level commander down to atomic units. This kind of structure can be used in game AI to coordinate behaviors of agents in squads or teams. For example, a team of bots in a capture-the-flag game could be grouped into small, unified squads which operate under the command of a team-level AI

to accomplish the various goals in that game (defending the team's flag, capturing the opposition's flag, and killing any opponents along the way). An important consideration in modeling this hierarchy is what degree of autonomy any unit is allowed. The commanding unit may not have complete or accurate information, and it may be preferable for the subordinate unit to react to new information without having to propagate that information back up the chain of command and wait for new orders to come back down. Consider a squad of bots ambushed by an enemy squad while en route to the enemies' flag. It is clearly desirable that the squad (or the individual units) immediately respond by taking cover and firing back at its attackers. In this project, a method is proposed which supports both autonomous and non-autonomous AI classes and gives commanding units a strong degree of control over which orders should be followed rigidly and which are more loose suggestions to the subordinate units. In a multi-level hierarchy, the top- and bottom-level AI classes would typically be autonomous (i.e., choosing their own Goals). The top-level must be autonomous by definition, and it is generally desirable for atomic agents to be autonomous so that they can react quickly to new information. The intermediate classes, however, could be non-autonomous and simply exist to propagate the top-level orders down through the chain of command appropriately. In either case, this method works by suggesting goals to the subordinate unit. It does not actually create a plan of action for the subordinate unit. In this way, the model is analogous to a military commander handing down an order to a squad leader but letting the squad leader actually devise the plan to execute the order. This means that "non-autonomous" units in this scheme do employ some planning of their own, but they do not make decisions about their goals. In short, every unit decides how, but only autonomous units decide what.

The following diagram indicates the relationship between classes in a command hierarchy. There is a one-to-many (1:*) association between any commanding unit and its subordinates, and the actions of the commanding unit recommend Goals to the subordinates. Any given class communicates only with its direct superior and subordinate (if any).

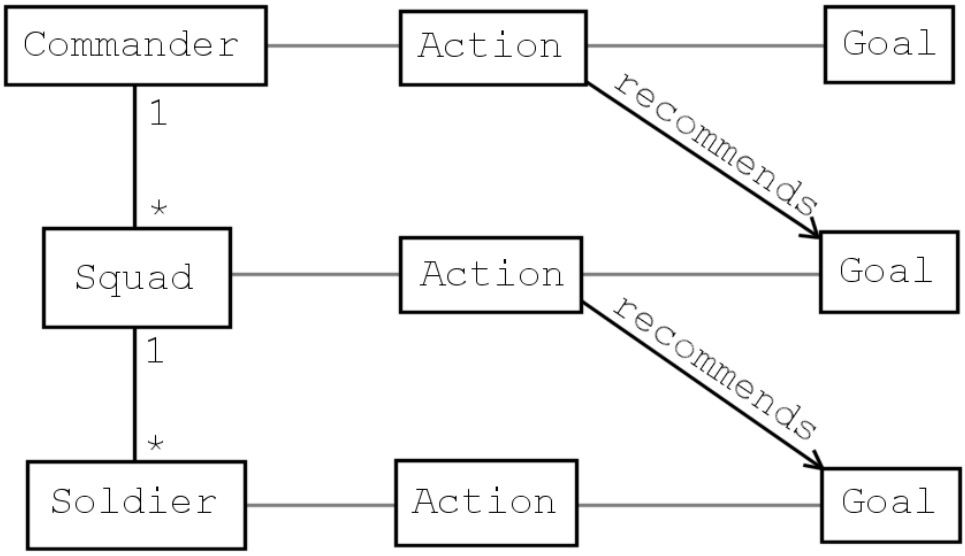

Figure 1.3 A command and control hierarchy implemented with GOAP

1.3 Motivation

GOAP has some interesting benefits for gameplay and development. Compared to more traditional, static decision-making techniques such as finite state machines and rule-based systems, GOAP AI appears to react intelligently to unexpected scenarios [GameSpot05]. This enhances the believability of the character AI and can improve the player experience by allowing more variety in how AIs achieve their goals (thereby reducing the chance that the player will "learn" and exploit the AI). The benefit to developers is that simple actions can be defined and reused in a modular way (instead of programmers or designers needing to define complex behaviors explicitly). Even without the element of planning, the goal/action paradigm is an increasingly popular choice for decision-making AI systems, having been used in numerous shipped titles such as *No One Lives Forever 2* [Monolith02], *SWAT 4* [Irrational05], and *Tribes: Vengeance* [Irrational04]. Planning a sequence of actions (instead of having a 1:1 goal/action pairing) is more computationally expensive but allows for more complex behaviors. The GOAP method has obvious potential for use in character-based games such as shooters and RPGs (because the planning process mimics the human problem-solving process to a degree), but it also has application to planning for non-character AI, such as abstract groupings (squads, platoons, etc.) of units in an RTS.

1.4 Objectives

This project is intended as a practical study of implementing GOAP in the Unreal Engine as well as a more theoretical analysis of building a command and control hierarchy using GOAP. This report is presented as one half of the project, while the other half is a production-quality implementation of GOAP AI in an *Unreal Tournament 2004* [Epic04] mod. This report serves as documentation of the development of the mod as well as a more in-depth review of the proposed scheme for hierarchical AI with planning.

The practical portion of the project is significant, as it suggests that the core algorithm may be developed cleanly by an individual with no prior GOAP experience in a relatively short amount of time. The project is also intended to demonstrate the applicability of the GOAP method to various uses within different game genres. In conjunction, these two propositions suggest that GOAP is a good choice for use in future commercial game development. This report also highlights issues that arose during development, justifies the chosen solutions, and theorizes about further issues which might face a full-scale project.

1.5 Original Proposal

For reference, a portion of the original project proposal is presented here.

Master's Project Proposal

Introduction to Goal-Oriented Action Planning

A relatively recent development in controlling AI behavior is the incorporation of automated planners to create a set of associated actions an agent can take to complete a high-level goal. In the traditional AI field, an automated planner (such as STRIPS, the Stanford Research Institute Problem Solver) is a system which typically takes an initial state, a goal state, and a set of all potential actions. These actions are defined by their preconditions (conditions which must be true for the action to be valid) and their effects or postconditions (the changes to the state as a result of the action). The planner then constructs an ordered plan of actions to achieve the goal state from the initial state. Despite the apparent suitability of this approach to game AI, automated planning and scheduling has only recently begun to appear in games. The goal/action architecture is quickly proving an increasingly popular alternative to rule-based systems for game AI, however, having been used successfully in Irrational Games' *SWAT 4* (without automated planning) and Monolith's *F.E.A.R.* (with automated planning).

An introductory example of goal-oriented action planning (GOAP) in games is described by Jeff Orkin in his GDC presentation *Three States and a Plan: The A.I. of F.E.A.R.* An agent may be given a set of potential high-level goals such as `Patrol` or `KillEnemy`. At runtime, these goals are evaluated for relevance, and the single best goal is activated. In order to complete these goals, the agent needs a plan of action, so he is given a set of potential low-level actions such as `DrawWeapon` and `Attack`. `Attack` has the precondition that the weapon is drawn and the effect that the enemy is killed. `DrawWeapon` has the precondition that the weapon is holstered and the effect that the weapon is drawn. The planner generates a sequence of these actions depending on the initial state. If the agent has his weapon holstered, his plan will be "`DrawWeapon`, then `Attack`." These actions are then executed latently (waiting for each animation to finish before proceeding to the next action).

Complexity emerges when multiple goals compete for precedence. In *F.E.A.R.*, an agent's `Cover` goal could activate when he was threatened by the player, and its high priority would likely cause it to supersede a `KillEnemy` goal. This would result in the agent moving to a safe vantage point (as defined by a level designer), at which point the `KillEnemy` goal would reactivate, causing the agent to fire from safety. The apparent result (to the player) is an enemy who is intelligent enough to protect his own life in a dynamic scenario while still fighting the player. This kind of behavior is exciting for the player and simple to choreograph for a designer (as well as requiring a less technical understanding of the AI architecture than comparable data-driven state machines and rules-based systems). A wide variety of high-level behaviors can emerge almost for free just by implementing a handful of low-level actions. The modularity of this design also provides designers with many of the same benefits as object-oriented programming, especially the ability to make large changes at a particular level without breaking the system, because goals and actions are decoupled. In fact, the developers of *SWAT 4* designed the AI architecture to specifically incorporate certain fundamental principles of OOP. In their GDC lecture *Using a Goal/Action Architecture to Integrate Modularity and Long-Term Memory into AI Behaviors*, Marc Atkin and John Abercrombie emphasized the hierarchical nature of their system (using a separate action planner each for squad-level, agent-level, and motor control-level systems), the use of actions to encapsulate common behaviors, and the ability to inherit from base goals.

Implementation Considerations

Two important challenges that arise when implementing GOAP are how to represent the state of the world and how to search the potential action space. These problems may be simplified with regard to a game (versus an academic scenario) by using a familiar searching algorithm and minimizing the set of relevant data to the planner. Other considerations include

defining how each individual agent perceives the world state and how this memory is stored and updated.

Searching for a plan in the action space is a problem with many similarities to pathfinding in games. Incidentally, the A* algorithm may be employed by the planner, with nodes representing world states and edges as the actions to change between states. The cost of a node is then the sum of the costs of the actions to reach that state (and the AI's behavior can be tweaked by assigning preferable actions smaller costs). The heuristic cost from a node can be calculated as the sum of the unsatisfied goal state properties in that node. In his book *Artificial Intelligence for Games*, Ian Millington recommends using IDA* (iterative deepening A*) for GOAP because it handles large action spaces with a relatively small memory cost and easily allows depth-limiting to prevent expensive searches for an unreachable goal state. Intuitively, a regressive search (starting from the goal node and searching backwards to the initial state) will be more efficient than a forward search. Orkin provides an example in which an agent intends to use a disabled gun turret. A valid sequence requires first turning the weapon on (via a remote generator, in this case) and then firing it. A forward search would require brute-force searching the action space to solve this problem, while a regressive search would first solve the problem of firing the gun, then tackle the sub-problem of enabling the turret. Searching for solutions in this order will be far more efficient.

Representing the world state to the planner is an especially easy task if a regressive search is implemented, as it minimizes the amount of relevant world data that must be considered. A world state property (as used by the nodes in a planner search) can be minimally stored as an enumerated key, a value, and sometimes a handle to a subject. Each node references only the properties that are relevant to the goal state. When starting from the goal node, this means that the only properties that must be considered are those which ultimately satisfy the goal (for example, a `TargetIsDead` key). As the search progresses, the goal state grows larger with each additional property that must be satisfied for each successive action (e.g., `WeaponIsLoaded`). When the current state for any given node matches the goal state, the search is completed and the edges (actions) of the search form the plan. The format for storing world properties can be used by the action definitions as well to indicate their preconditions and effects. Actions may additionally require more complex preconditions known as context preconditions. These are criteria which must be met for the action to occur but which cannot be easily captured by the planner's world state representation.

An agent's perception of the world state is determined by sensors which may be updated on events or through polling. The knowledge returned by a sensor is stored in a persistent location (e.g., using a blackboard system) so that it may continue to be accessed when a goal or action ends. A confidence value may be associated with the knowledge store

to represent the agent's certainty of the truthfulness of the information. Sensors and knowledge stores also have the potential to be used to distribute the computationally expensive cost of evaluating context preconditions across many frames. The results are cached and accessible to the planner so that it can efficiently check many context preconditions during a search (in a single frame).

Proposal

My proposed project is an implementation of goal-oriented action planning for squad-based AI in *Unreal Tournament 2004*, with a focus on providing useful debugging and designer tools for easy understanding and tweaking of the AI. My primary goal in this project is to implement the goal/action system and automated planner and learn more about the nature of this field of AI in the process. This will manifest in a game demo in which the player can give orders (via a simple point-and-click interface) to a friendly squad to direct their movement and cause them to engage in combat with one or more enemy squads. Beyond the basic implementation, I intend to develop the AI with a rigorous focus on the toolsets. During my internship at Gearbox Software, I spent many weeks working on their AI debugging tools, and I recognize the specific need for such a feature in the development cycle. Gearbox also firmly advocates a data-driven approach to game design, enabling the designers to tweak weapon and enemy behaviors without requiring programmers to recompile anything. I believe there is some value to this methodology, and I intend to support both of these feature sets as a secondary goal of my project. A tertiary objective is to produce a highly playable, fun demonstration of a squad-based shooter game utilizing the AI to its fullest for gameplay purposes. (This will involve at least a small amount of level design and some iterative gameplay testing, and is the least relevant to my actual study.)

My justification for using *Unreal Tournament 2004* is twofold. By utilizing the existing low-level AI functionality in the Unreal engine (pathfinding, movement, animation), I can spend my time more appropriately on the high-level GOAP system. I anticipate being able to reuse most of Unreal's pathfinding code by extending the PathNode object, but I will be writing an implementation of A* for the action planner anyway and could theoretically make it generic enough to do my pathfinding as well. My second reason for using Unreal is for its latent functionality. Function calls can wait for an animation to finish before continuing execution, and UnrealScript includes much of the common functionality of threaded programming. I have not previously worked with animation at any significant level, and this represents an opportunity to do so. My primary reason for not using my own engine developed at the Guildhall is the limited timeframe I would have to develop a low-level AI framework on which to build the high-level AI. Additionally, my engine lacks the robust development tools that Unreal can provide, and I intend to take full advantage of the editor

(and perhaps a fellow level designer or two) to produce an impressive demo piece. The obvious disadvantage to using Unreal technology is that I will be programming in a scripting language (without a reliable debugger) instead of C++, but I can accept that if the committee is willing to allow it. It might also be argued that the use of Unreal could render my project underscoped; however, I have analyzed the estimated time I will spend on each aspect of the project, and I believe my plan is on level with the expected timeline for the Master's project.

In order to demonstrate the benefits of GOAP for gameplay, I will evaluate my project by having external users test the game and provide feedback regarding the AI and squad behavior. If time allows, I will also develop a simple FSM-based AI system for use as a baseline with which to compare the GOAP AI. Likewise, I intend to gather opinions from fellow programmers and level designers regarding the success of the AI debugging and design tools (and perhaps again compare the debug tools to a baseline—in this case, having no tools to provide AI information). Internally, I will evaluate the tools as I use them to test and debug the AI system, and I will report on their effectiveness at the completion of the project.

CHAPTER 2

FIELD REVIEW

1.1 Traditional Decision-Making

Decision-making in game AI has traditionally been a mostly static process. By far the most common architecture is the finite state machine (FSM), in which an AI's behavior is wholly determined by its state and the rules for changing between states are fixed. This has sufficed well for years, but it is not without faults. Given the highly deterministic nature of the architecture, it is easy for a player to "learn" the AI and develop strategies which exploit the fixed behavior. Various methods of introducing non-determinism to AI decision-making have been proposed, including neural networks and genetic algorithms. These systems are more difficult to tune and debug because of their inherent non-determinism, and the few shipped titles which have utilized these algorithms have been criticized for apparently random AI behavior [GameSpot96].

1.2 Planning

The motivation for planning arose from the cross-section of the faults of these prior approaches to AI decision-making. The ideal AI system would be deterministic to make development easier, but still offer a broad range of responses to any given scenario. Planning AI using a goal-based approach is a relatively new field for game AI. The bulk of the existing research has been written within the last five years. The broader subject of automated planning outside the scope of games has been researched within academia for decades, but only recently has it become practical to apply to games. A general purpose goal-based planning AI architecture was proposed in 2002 [O'Brien02], and the more specific STRIPS-influenced GOAP technique was widely introduced soon after [Orkin04a]. A planning-based approach to creating coordinated behaviors using hierarchical task-network planning instead of STRIPS was proposed [Muñoz-Avila06], but its implementation abandons the convenient agent-centric architecture proposed in [Orkin04a]. Planning in games is a state-of-the-art feature, having only been seen in shipped titles in the last couple of years. GOAP has been used in Monolith's *F.E.A.R.* [Monolith05a] and *Condemned* [Monolith05b]. It has been suggested that Bethesda's *Oblivion* [Bethesda06] originally used a planning system (dubbed Radiant AI), although the feature appears to have been pared

down prior to the game's release, perhaps due to difficulty tuning the behavior. Based on these disparate outcomes, the usefulness of planning for game AI is still largely undecided.

CHAPTER 2

METHODOLOGY

2.1 Original Proposal

For reference, an overview of the intended design from the original project proposal is presented here.

1.1 Overview

My Guildhall Master's project, codenamed Carnival, is a demonstration of an artificial intelligence system called Goal-Oriented Action Planning (GOAP). GOAP is a high-level decision-making technique in which a series of actions (a plan) is generated to satisfy an NPC's goals. A simpler form of this goal/action architecture (without planning) has been utilized in such games as Monolith Production's *No One Lives Forever 2* and Irrational Games' *SWAT 4*, while Monolith's *F.E.A.R.* used full-fledged action planning.

My intent is to develop a GOAP AI architecture along with debugging and design tools that would make it practical for use in a game development studio. As described in my proposal, GOAP builds a list of NPC actions to satisfy a goal. Goals are defined as some proposition about the world state. Actions are defined by their world state preconditions and effects (literally, what must be true for the action to occur, and what will change as a result of the action). Building a plan is accomplished by searching the potential action space (with the A* algorithm) for a low-cost plan that can will produce the goal's desired world state and can be completed given the current world state.

1.2 Product Description

The end result will be a playable *Unreal Tournament 2004* mod with an emphasis on squad-based combat. The player will be able to give a few simple, high-level orders to a friendly squad by means of a point-and-click command ring (a la *Brothers in Arms: Road to Hill 30*). Both friendly and enemy units will autonomously resolve their goals such that they will usually follow orders but not to the detriment of their health. Use of cover will feature prominently in the gameplay, and all units (including the player) will have the use of rifles, pistols, and/or grenades.

The final product also includes the debugging and design tools. These will be developed as I need them and cannot be documented thoroughly at the commencement of the project; however, they are likely to include in-game visualization using text and lines, as well as some kind of historical record of the path and previous states of each NPC. These will most likely be activated through console commands for the sake of development time,

although a debugging HUD would be desirable. On the design side, there will be a set of new navigation points for assisting and controlling the AI, and behaviors will be fully customizable through the UnrealEd interface by exposing goal sets, action sets, and tuning variables to the editor.

2.2 Overview

The practical development portion of this project is an implementation of the GOAP architecture as described in [Orkin04a] in an Unreal Tournament 2004 mod. The project was designed to be built to production quality standards, easily extensible, and supported by a set of design and diagnostic tools. These are the same standards to which a commercial product would be held. The GOAP platform was completed in one term (about three months), including an AI agent architecture, a state-space search for planning, and a debugging toolset. Various demos were then built on this platform to show the flexibility of the technique for different game genres.

2.3 Selecting a Goal

Before a plan can be formulated, an agent needs to select which Goal it will attempt to complete. Each Goal has a function EvaluateRelevancy which returns a value between 0.0 and 1.0 depending on the context and the weight the designer places on that Goal. (For example, a TakeCover Goal will be very relevant when an agent has low health and is faced by multiple enemies.) If an agent has a superior AI class (e.g., a Squad or Commander), it may receive Goal recommendations from that class in the form of relevancy biases. This provides a method for coordinated behaviors that allows an agent to retain its autonomy in the event that another Goal outweighs a recommended Goal. Likewise, a superior AI may add a negative bias to a Goal in the AI's set to discourage certain behaviors. In this implementation, a superior AI can not add or remove Goals from the AI's set, only modify the relevancy of the existing Goals.

2.4 Graph Search and the Planner

The fundamental algorithm for planning is a state-space A* search. As described in [Orkin04a], plan formulation is akin to pathfinding, albeit with a non-topological graph. In this problem, the graph nodes are world states and the graph edges are actions which

transform the world between the two states. Because of the theoretically infinite nature of this graph, the search is somewhat more complicated than a typical A* pathfinding search. To reduce the size of the graph that needs to be searched, it is best to search backwards from the goal state and only push actions which satisfy an unsatisfied goal property or action precondition.

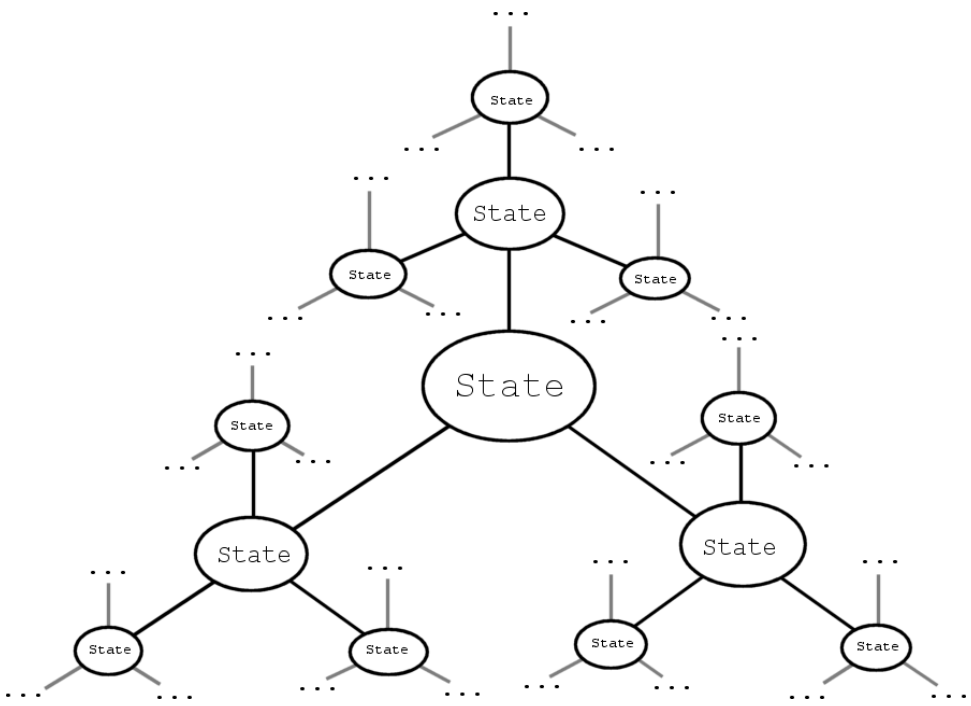

Figure 3.1 A theoretically infinite state-space graph

The lists for state-space A* search for the Planner were implemented using UnrealScript's built-in dynamically resizeable array type (the equivalent of a std::vector in C++). Sorting these lists was done with a custom quicksort function. It may generally be faster to use priority queues for A* instead of resorting a list every iteration, but the overhead of a custom container in UnrealScript did not seem worthwhile.

The planner must instantiate new instances of Actions when it pushes them to the list. This is because Actions contain references to specific objects (e.g., navigation nodes), and any Action may exist multiple times in one plan (and must persist for at least the duration of the acted behavior, while the AI uses its data). A newly created Action inherits contextual properties from the action or goal which depends on it (e.g., navigation nodes again).

Context preconditions (action preconditions which depend on data that is not represented in the world state) are also evaluated at this time.

2.5 Debugger

In order to successfully debug AI, it is useful to have a tool which records and displays information in a fashion which makes it easy to identify the cause of a problem after it occurs. The tool developed for this project takes "snapshots" of the state of every AI at frequent intervals. When undesirable behavior is noticed, a user can pause the game and rewind through an array of recent snapshots to find the desired information. It is possible for very brief events to be missed when the snapshots are taken at fixed intervals. It might be more ideal if a snapshot were taken at every AI event (generating a new plan and completing actions, for example), but this was not done in the project. It is desirable that the debugging tool can be turned off with a minimum of overhead left in the system. In C++, this could be accomplished by compiling out the debugger functionality using macros. In UnrealScript (as of version 3369), there is no support for macros, so it was decided instead that there would exist a single entry point to the debugging system, which would require only one conditional check if the feature was disabled (as opposed to various debugger hooks throughout the AI agent class).

The debugging toolset consists of three classes which interface with the AI class to record and recall AI data at runtime. The AIDebugger class is responsible for overseeing the debugging system in general. It maintains an array of AITrackers (one for each debuggable AI) and calls the "take snapshot" function on each one every time slice. It is also responsible for interfacing with the player class to give the user control over the debugger and to draw the appropriate data on the HUD. An AITracker is assigned to each debugged AI. It maintains a fixed-size array of AISnapshots in a rotating queue and is responsible for passing an AISnapshot reference to the AI to fill with data or to the HUD (by way of the AIDebugger) to visualize data for debugging. The AITracker is responsible for deciding how to visualize the data (i.e., as text, icons, lines, etc.). The AISnapshot class is essentially a struct for the AI to fill with relevant data. Some parts of it simply mirror the AI's own data (e.g., position and rotation) while other properties can be computed or converted (e.g., to strings) as desired for the visualization. The AITracker and AISnapshot classes can be

extended for each type of AI that will be debugged. For example, an AISquadSnapshot could be derived from AISnapshot and contain an array of Pawn references (for the units that belong to the Squad), and an AISquadTracker could be derived from AITracker to visualize this set of Pawns at runtime (perhaps by drawing an icon above each member of the Squad).

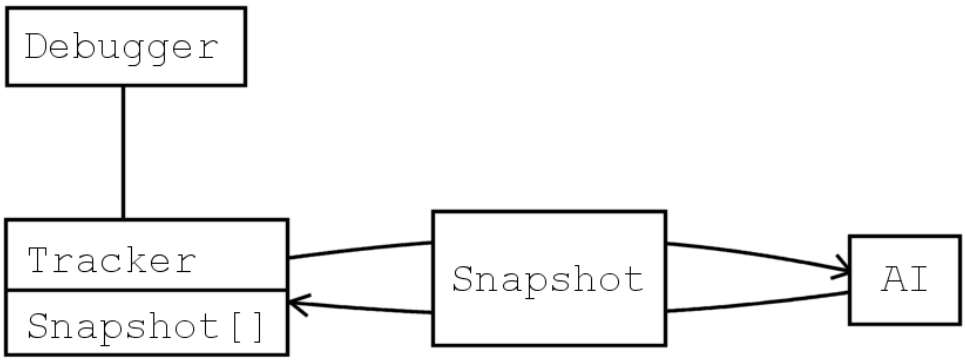

Figure 3.2 The relationship between the debugger classes

2.6 Optimizations

Performance became a concern on this project when simulating a significant number of agents. The planner requires a fairly complex searching algorithm and the entire system is implemented in UnrealScript (an interpreted language), which causes the potential for the frame rate to hitch if multiple units have large planning jobs to complete in a single frame. Two optimizations were introduced to assuage this problem. First, the A* search was made interruptible (an easy addition that simply requires saving the open and closed lists until the search can be continued on a subsequent frame). A simple scheme was used for this purpose, such that the search is interrupted after a fixed number of iterations. Second, a planning scheduler was added to manage the AI agent requests to the planner. In this paradigm, an AI agent which needs to formulate a new plan submits a request for planning to the scheduler, which adds the AI reference to a FIFO queue. Every frame, the front item is dequeued and submitted to the planner. If the planner is interrupted, the AI is resubmitted to the queue (such that frames are distributed evenly amongst multiple AIs with long planning tasks). In this implementation, the agent is left idle while waiting for a plan; if this behavior is problematic (e.g., if the idle state lasts long enough to be discernable to a player), an

alternative would be to provide simple, static one-action plans for common goals that the AI could fall back on during the wait. Further optimizations could include biasing the requests to the scheduler to give preference to AIs which are nearer to or otherwise more relevant to the player.

2.7 Measuring Performance

The performance of the implemented system was measured in two ways. First, a simple hook was added to the game class that would spit out a log message if a frame's delta time exceeded 16ms (the reciprocal of a 60fps frame rate). This helped identify if a planning search took too long and caused a frame time spike. A more thorough examination of the average performance of each component of the GOAP architecture was done using the built-in UnrealScript profiler.

2.8 Making the Planner Generic

At the beginning of the project, the atomic AI agent class (derived from Unreal's AIController) interfaced directly with the Planner class. This caused a problem when it came time to reuse the Planner for other types such as Squads. Because of the nature of the AIController class, abstract (i.e., not appearing in the game world) classes such as Squads could not be derived from the existing AI. Two solutions were apparent. Either the GOAP portion of the AI could be extracted from and refactored to interface with the AIController, or a Mediator class could be employed to handle communication between the Planner and any (unrelated) AI classes. The Mediator solution had the side effect of forcing some of the AI code to be duplicated between the agent and squad implementations, but it required less refactoring than the alternative solution and sufficed for the purpose of this project. In future projects, care should be taken to avoid this situation from the outset.

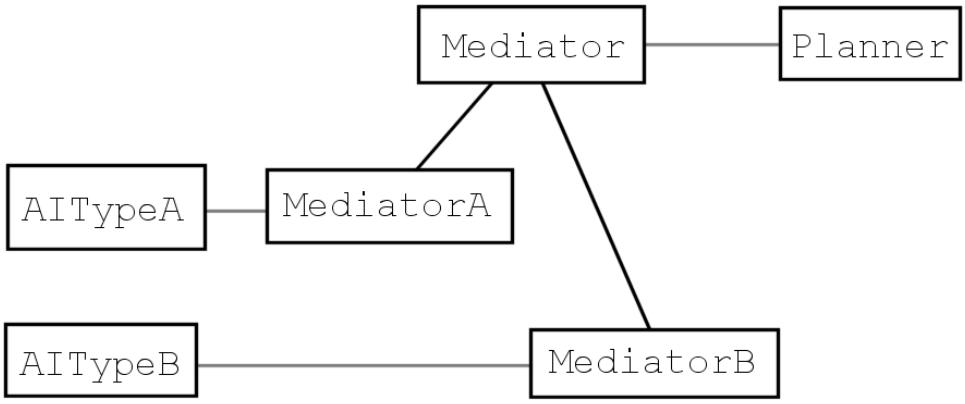

Figure 3.3 Using the Mediator design pattern

2.9 Command Hierarchy

The command hierarchy is handled in the following manner. Every AI class in the hierarchy implements a planning system as previously described (although not every class may have a literal agent in the world). The commanding unit formulates an action plan to satisfy its selected goal. Its actions may recommend certain goals to its subordinate units. It may also discourage the selection of other goals. In this implementation, the commanding unit can only influence the selection of Goals already in the subordinate's Goal set. It cannot add new Goals to the Goal set. The subordinate units use these biases when picking their next goals by adding the bias value to the evaluated relevancy for the given Goal. (This means that an AI could still select a Goal which was not recommended, provided its relevancy was greater than the relevancy of any recommended Goal after the addition of the bias value.) The commanding action may also suggest targets to the subordinate units (e.g., to mark a shared target for a coordinated strike by all members of a squad, or to designate different movement targets to the members of the squad in order to arrange a pincer movement). For example, in a squad shooter, a Commander AI class might select and formulate a plan for an AssaultTarget Goal. The plan could begin with a SetUpAmbush Action which suggests movement Goals and separate movement targets to the subordinate Squad AI classes. The formulated plan for these movement Goals would include Squad-level Actions which in turn recommend movement Goals and targets to the individual units in the Squads. A consideration that must be made is how to determine when the commanding unit's action is complete. Because the subordinates may be autonomous, there is a chance

that the order will not be fulfilled. A solution is to make the subordinates report to the commander when they pick a recommended goal and when that goal is completed. The commander keeps a record of the units which are fulfilling its action and checks each time one reports completion if the action can be considered completed. It is also ideal if the commanding action has the potential to "time out" or otherwise fail if the subordinates do not complete it, such that the commander unit AI does not stall. In the case of a non-autonomous subordinate, the selected goal is by definition the most highly-recommended goal, but the considerations on how to handle failure still apply because the outcome of the simulation is not predictable and the subordinate may fail to complete its goal.

2.10 World State Representation

This project follows the model suggested in [Orkin04a] by using an agent-centric world state that is updated by various sensors attached to the agent. Each sensor has the capacity to interrupt the AI if the changes made are potentially significant to its plan. In this case, the active goal is recalculated and the AI will generate a new plan. (It is not necessarily possible to continue using the existing plan just because the goal is the same; the targets referred to by the world state may have changed.) In the command hierarchy proposed in this project, a commanding unit which is interrupted should interrupt all its subordinates, such that its new goal (and any information it may give its subordinates) is propagated immediately through the hierarchy.

2.11 Designer Interface

Designers can place AIs (Pawns and Squads) in the world and customize each one's action set, goal set, and sensors list. With these tools, special behaviors can be created per AI. For example, a fixed-position sniper could be set up by excluding any movement actions or goals from its sets. In the UnrealEd environment (as of *UT2004* v3369), instantiated Actor references cannot be linked in any intuitive way to other Actors, so associations (e.g., between Pawns and the Squads they belong to) are made by name, and the references are fixed up at runtime.

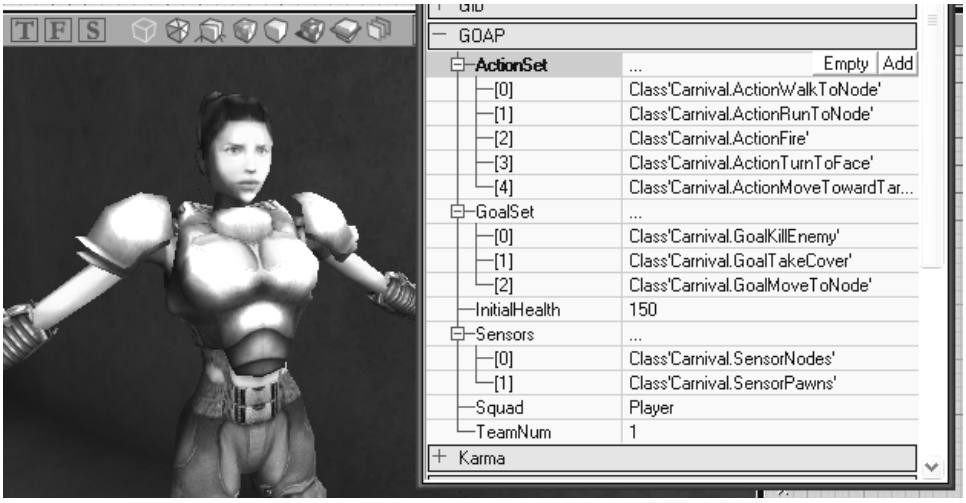

Figure 3.4 The UnrealEd interface for editing GOAP AI properties

2.12 External Evaluation

In order to evaluate the usability of the designer interface and the system architecture, programmers and level designers from The Guildhall at SMU were invited to participate in a research study. The first part of the study involved assigning Goals and Actions to agents in UnrealEd to set up AI behaviors according to a given design. One agent was required to exhibit wandering, taking cover, and attacking behaviors, while a second agent in a sniper tower was required to hold that position and attack. The second portion was to fill in a template of the GoalKillEnemy and ActionFireWeapon classes in UnrealScript. Participating students were given a brief introduction to the GOAP technique, advised on the nature of the scripts they were to implement, and presented with a list of names of world state keys. Further advice was only given upon request if the participant was unable to proceed or did not understand a critical aspect of the system. Following the study, each student was asked to discuss how they felt about using this architecture.

CHAPTER 3

RESULTS AND ANALYSIS

3.1 Introduction

The purpose of this project was to evaluate the practicality of the Goal-Oriented Action Planning technique in terms of its cost (or time) to implement, its efficiency, and its effectiveness for various types of gameplay scenarios. This was accomplished by implementing a GOAP architecture and creating a variety of demo behaviors for it. Furthermore, the project proposed a method for handling command hierarchies using GOAP, and an example was developed using the implemented GOAP system.

3.2 Development Costs

The base GOAP architecture (including the debugger) was developed within one term (three months) by the author, whose time was divided between this project and a team project of comparable magnitude. Several weeks of research and planning preproduction were done prior to the start of actual development. This core system was utilized in the following term for the development of game demo scenes. The success of this project in terms of the cost and the usability of the code suggests that a single dedicated AI programmer could create and maintain a GOAP-based AI on a full-scale commercial project. The resulting code has a clearly delineation between the core planning elements and the more game-specific behavioral components. An additional programmer or scripter could be responsible for developing new AI behaviors without having to modify (or even fully understand) the underlying behavior of the planner.

3.3 Performance

The performance of the implemented GOAP code was profiled and shown to be comparable to or faster than various other components of the *Unreal Tournament 2004* script code. Due to the infrequent nature of planning, GOAP AI produces spikes in computation time rather than using a consistent number of cycles. The goal is to keep these spikes from pushing the frame time so high that the frame rate drops below 60fps and causes a visible hitch. This was successfully accomplished by implementing a scheduler and interruptible A* search for the planner. These optimizations guarantee that the time spent in the planning

system will never exceed some fixed amount. The tradeoff is some potential latency in generating a plan. Because only one AI can plan in a given frame and the planner may take multiple frames to complete a search, the planning process may take many frames if multiple AIs need to generate complex plans simultaneously.

The following screen shots represent profiler statistics from the first demo scene (a squad shooter game). It should be noted in particular that although the Plan function is fairly expensive (see incl. per call in the second image), it is low enough to not cause frame rate hiccups. The overall percentage of time spent in the planner is comparable to other important game classes (xPlayer.PlayerTick and xPawn.Tick in the first image).

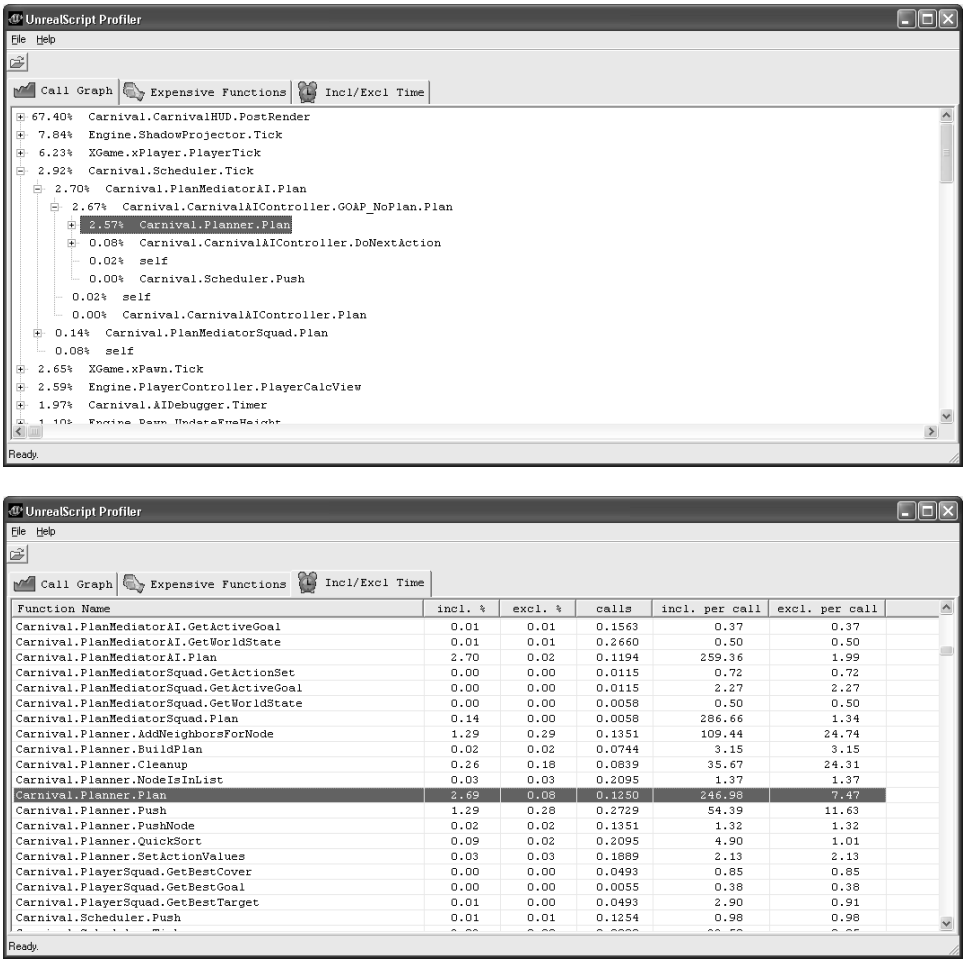

Figure 4.1 Profiler statistics for the first demo scene

3.4 External Evaluation

Every student who participated in the research study grasped the relationship between Goals and Actions with ease. The first portion of the study (setting up Goal and Action sets for two agents) presented no difficulties to any students. (All of the participants had prior experience with the UnrealEd interface.) Each student correctly identified the necessary three Goals from a list of nine and selected Actions which would properly satisfy those Goals. The scripting part of the study tended to require a further explanation of the key-value pair world state properties than was originally presented. A majority of the participants misunderstood the nature of the Goal properties and initially defined the desired state as {TargetIsDead:false} instead of {TargetIsDead:true}. After the concept was further explained and understood by each participant, filling in the appropriate keys went smoothly in every case. A post-study briefing with each participant revealed that they generally understood the nature of the architecture well enough to feel comfortable developing new behaviors in this implementation of GOAP.

3.5 Applicability to Genres

The implemented GOAP architecture was used as a base for developing a small number of gameplay demos. The first of these was a squad shooter in the style of *Brothers in Arms: Road to Hill 30* [Gearbox05]. In this demo, it was shown that a squad of three units could use shared knowledge and coordinated behavior to (usually) defeat three opponents working independently. Most plans generated were only one or two actions long, but the modular nature of the goals and actions were still beneficial to easily tweaking behaviors to get the desired response from the characters.

Figure 4.2 Squad shooter demo

The second demo produced was another example of coordinated behavior in a squad shooter-type game. This demo utilized more of the principles of the proposed command hierarchy scheme and featured a squad of four units advancing through a series of rooms in an organized fashion. The squad used one goal (clear the room), which was developed into a two-action plan (stack up outside room, then clear the corners of the room). These actions recommended movement goals and targets to the individual members of the squad. As each unit completes the movement, it reports back to the squad. After each member checks in, the squad progresses to the next action or goal. The result is a highly coordinated SWAT-style room clearing behavior.

Figure 4.3 Room clearing demo

The third demo was a condensed version of behaviors which might be desired in an RPG. A pirate character was given one goal, to get drunk. One action (drinking rum) would satisfy that goal, but the environment provided a variety of methods of obtaining the rum. The pirate could be lawful or unlawful (as controlled by the user). An unlawful pirate might simply steal a bottle of rum and be done with it, while a lawful pirate would work to earn a gold coin and purchase the bottle. If the user removed the bottle from the world, the pirate could instead collect a cask of molasses and brew his own rum. The user was given complete control of the world state to experiment with the various behaviors. Dynamic replanning could be observed in this demo by removing an item after the AI had built a plan which depended on that object. When the character arrived at its location, it would check that the item still existed. If not, the plan would be invalidated and the AI would replan using the remaining objects to satisfy the same goal.

Figure 4.4 RPG demo

These demos collectively suggest some of the possibilities of the GOAP system and were successful given their scope. In the latter cases, the behaviors were developed in the span of a few hours for a very limited environment. Developing more generic behaviors and balancing the priority of the goals and actions would become an increasingly more complex task (especially in the case of an RPG, in which the AI would be picking from among goals as varied as killing demons to eating a meal). This would require a coordinated effort by a designer and programmer, likely for the duration of a project's development, to achieve the desired behavior. It is not an especially difficult problem (it is similar to the balancing which game designers already do), but it is one which deserves its own consideration.

CHAPTER 4

DISCUSSION AND CONCLUSIONS

4.1 Conclusion

This project has discussed the development of a GOAP-based game AI architecture. This architecture was successfully implemented in an *Unreal Tournament 2004* mod using the UnrealScript language. It has been shown to be a practical system with potential for various game genres. Solutions to its performance and maintenance issues have been suggested. Additionally, a method for implementing command hierarchies using GOAP has been proposed and briefly examined as a part of the implemented project. Based on the results of this project, further use of GOAP for AI in character-based games is considered advantageous.

4.2 Further Research

The timeline of this project dictated that the scope of the game demos remain small. Utilizing the GOAP architecture as presented here in a large-scale RPG would be an important and interesting extension to this project. Developing a plan to steal gold is easy when the item is readily available, but perhaps in a real game, this goal would involve breaking into a bank vault. In such a case, it might be more practical to plan sub-goals and plan recursively until each goal was resolved into atomic actions. This is more like hierarchical task network (HTN) planning. Likewise, a fuller demonstration of the command hierarchy proposed in this project would be worth developing. The example developed for this project used a two-layer hierarchy (squad and units). Higher-level AIs implement coarser-grained actions, and it remains to be shown whether action plans remain feasible at these high levels in a deeper hierarchy. HTN planning has been used for the purpose of command hierarchies in prior research [Muñoz-Avila06], and it would be interesting to compare the values and challenges of using each method.

REFERENCES

[AIISC] Artificial Intelligence Interface Standards Committee of the AI Special Interest Group of the IGDA, *http://www.igda.org/ai*.

[Atkin05] Atkin, Marc, and John Abercrombie, "Using a Goal/Action Architecture to Integrate Modularity and Long-Term Memory into AI Behaviors." *Game Developers Conference*, 2005.

[Bethesda06] *The Elder Scrolls IV: Oblivion*. Bethesda Softworks, 2006.

[Epic04] *Unreal Tournament 2004*. Epic Games/Digital Extremes/Atari, 2004.

[GameSpot96] GameSpot, *http://www.gamespot.com/pc/strategy/battlecruiser3000ad/review.html*, 1996.

[GameSpot05] GameSpot, *http://www.gamespot.com/pages/features/bestof2005/index.php*, 2005.

[Gearbox05] *Brothers in Arms: Road to Hill 30*. Gearbox Software/Ubisoft Entertainment, 2005.

[Guildhall06] *Ransacked!*. Overthinking Games/The Guildhall at SMU, 2006.

[Irrational04] *Tribes: Vengeance*. Irrational Games/Vivendi Universal Games, 2004.

[Irrational05] *SWAT 4*. Irrational Games/Sierra Entertainment, 2005.

[Millington06] Millington, Ian, *Artificial Intelligence for Games*. Morgan Kaufmann Publishers, 2006.

[Monolith02] *No One Lives Forever 2: A Spy in H.A.R.M.'s Way*. Monolith Productions/Sierra Entertainment, 2002.

[Monolith05a] *F.E.A.R.: First Encounter Assault Recon*. Monolith Productions/Vivendi Universal Games, 2005.

[Monolith05b] *Condemned: Criminal Origins*. Monolith Productions/SEGA of America, 2005.

[Mori70] Mori, Masahiro, *The Uncanny Valley* (originally *Bukimi no tani* in Japanese). *Energy*, 1970.

[Muñoz-Avila06] Muñoz-Avila, Héctor, and Hai Hoang, "Coordinating Teams of Bots with Hierarchical Task Network Planning." *AI Game Programming Wisdom 3*, Charles River Media, 2006.

[O'Brien02] O'Brien, John, "A Flexible Goal-Based Planning Architecture." *AI Game Programming Wisdom*, Charles River Media, 2002.

[Orkin04a] Orkin, Jeff, "Applying Goal-Oriented Action Planning to Games." *AI Game Programming Wisdom 2*, Charles River Media, 2004.

[Orkin04b] Orkin, Jeff, "Symbolic Representation of Game World State: Toward Real-Time Planning in Games." AAAI Challenges in Game AI Workshop Technical Report, 2004.

[Orkin06] Orkin, Jeff, "Three States and a Plan: The A.I. of *F.E.A.R.*" *Game Developers Conference*, 2006.

[Tozour02] Tozour, Paul, "Building an AI Diagnostic Toolset." *AI Game Programming Wisdom*, Charles River Media, 2002.

APPENDIX: RESEARCH STUDY CODE

```
class STUDY_GoalKillEnemy extends AIGoal;

function float EvaluateAIRelevancy( CarnivalAIController AI )
{
	// This function tells the AI whether this goal is relevant
	// to the AI in the given context.

	// Write a function to return a value between 0.0 (meaning this
	// Goal is not relevant in this context) and 1.0 (meaning this
	// Goal is completely relevant in this context).

	// Possibly useful things:
	//AI.Pawn.Health		(int)
	//AI.Pawn.Weapon.HasAmmo()	(returns boolean)
	//AI.EnemyTarget		(Actor)
	//AI.EnemyTarget.Health	(int)

	return 0.0;
}

defaultproperties
{
	// The goal is defined in terms of specific properties of the
	// desired state of the world.  Fill in one or more DesiredState
	// properties to define a "KillEnemy" goal.

	// The Key can be one of the following:
	// WSKEY_CanSeeTarget
	// WSKEY_FacingActor
	// WSKEY_TargetIsDead
	// WSKEY_AtNode
	// WSKEY_CoverIsActive

	// bValue is a boolean (true or false)
	// aValue is an Actor reference
	// Instead of setting bValue, bVariableValue=true can be used to
	// define a property whose value will be set in a function later.

	DesiredState(0)=(Key=/* fill this in */,bValue=/* this too */)
	//DesiredState(1)=...
	//DesiredState(2)=...
}
```

```
class STUDY_ActionFireWeapon extends AIAction;

function SetAIContextProperties( CarnivalAIController AI )
{
	// This functions gets called during the planning process.
	// It may be useful to set some values in here, e.g.:

	//Preconditions[0].aValue = /* some Actor */;
	//Preconditions[1].bValue = /* true or false */;
}

defaultproperties
{
	// The Cost defines how likely this action is to be chosen by
	// the planner.  Low cost actions will be chosen over higher
	// cost actions when they are available.  This should be a
	// number equal to or greater than 1.0.

	Cost=/* assign this */

	// An Action is defined in terms of its preconditions (properties
	// which must be true for this action to be executed) and its
	// effects (properties which may be made true by the execution
	// of this action).  Fill in one or more Preconditions and one
	// or more Effects to define a "FireWeapon" action.

	// The Key can be one of the following:
	// WSKEY_CanSeeTarget
	// WSKEY_FacingActor
	// WSKEY_TargetIsDead
	// WSKEY_AtNode
	// WSKEY_CoverIsActive

	// bValue is a boolean (true or false)
	// aValue is an Actor reference
	// Instead of setting bValue, bVariableValue=true can be used to
	// define a property whose value will be set in a function later.

	Preconditions(0)=(Key=/* fill this in */,bValue=/* this too */)
	//Preconditions(1)=...
	//Preconditions(2)=...

	Effects(0)=(Key=/* fill this in */,bValue=/* this too */
	//Effects(1)=...
	//Effects(2)=...
}
```