Goal-oriented Behaviour for Intelligent Game Agents

Ying Ying She

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montréal, Québec, Canada

April 2011

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Ying Ying She

Entitled: Goal-Oriented Behavior for Intelligent Game Agents

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. A.M. Hanna

_____ External Examiner
Dr. H. Vangheluwe

_____ External to Program
Prof. L. Hughes

_____ Examiner
Dr. T. Fevens

_____ Examiner
Dr. O. Ormandjieva

_____ Thesis Supervisor
Dr. P. Grogono

Approved by _____
Chair of Department or Graduate Program Director
Dr. H. Harutyunyan, Graduate Program Director

April 13, 2011 _____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Goal-oriented Behaviour for Intelligent Game Agents

Ying Ying She, Ph.D.

Concordia University, 2011

This thesis concerns our innovation in game AI techniques, mainly game agents' modeling, planning and learning. The research topic involves the development of a game design software — Gameme. Our work mainly focus on the development of the core AI module.

In this thesis, after discussing the system design of Gameme, we explain our contributions in two parts: off-line design and real-time processing. In off-line design, we present goal-oriented behaviour design and related modeling methodology for game agents. The goal-oriented design provides not only an intuitive behaviour design methodology for non-professional game designers but also efficient support for real-time behaviour control. In particular, the goal-oriented design can be used in modeling agents in different games.

The real-time processing component includes planning and learning mechanisms for game agents. These mechanisms are placed in a layered architecture. Basically, a procedural planning mechanism allows game agents to have the ability of fast reaction to their environment. Then, the creative transfer and adaptive learning mechanism trains game agents to learn from their experience and cooperate in teamwork. Furthermore, the unique emergent learning mechanism can allow game agents to have the ability to analyze different PCs' behaviour patterns and to find the suitable strategy to defeat PCs in real-time.

Most of the experiments in this thesis are performed in fighting scenarios. We connected the core AI module with a 3D graphics engine in order to have visual testing results. All test cases show that our goal-oriented behaviour design along with planning and learning mechanisms can provide fast, autonomous, collaborative and adaptive behaviour instructions for game agent in real-time game play.

Acknowledgments

I would like to give grateful thanks to my supervisor, Dr.Peter Grogono, for his supports and guidance throughout my Master and Ph.D study. This research project would not have been completed without his concise instructions and flexible supervision.

I would also like to express my appreciation to my parents for setting high expectation and being consistently serious about my education. Finally, I would like to express my deepest gratitude to my husband and son, thank you for the miracle of love&life and the bounces of encouragement.

Contents

List of Figures

List of Tables

List of Abbreviations

Acronym	Meaning	Section	Page
AI	Artificial Intelligence	1.1	1
BBAI	Behaviour-Based AI	4.3	43
BDI	Belief-Desire-Intention	2.6.1	22
BT	Behaviour Tree	2.3.2	17
CAS	Complex Adaptive System	2.6.4	26
FSM	Finite State Machine	2.2.1	12
FuSM	Fuzzy-State Machine	2.2.2	15
GA	Genetic Algorithm	2.3.3	19
GOBD	Goal Oriented Behaviour Design	5.2	50
GOBT	Goal Oriented Behaviour Tree	1.1	2
LOD	Level of Detail	4.5.3	45
MAS	Multi-Agent Systems	8.3	104
MVC	Model-View-Controller	3.2	31
NPC	Non-player Characters	1.1	2
OOD	Object-Oriented Design	4.5.3	45
PC	Player Characters	1.2	3
PPS	Procedural Planning System	7.2	92
PRS	Procedural Reasoning System	2.6.1	22
RBS	Rule-based Systems	2.2.3	15
SDK	Software Development Kit	2.1.3.2	10
TC	Termination Condition	5.6.3	60
UI	User Interface	3.1	29

Chapter 1

Introduction

1.1 Motivation

Game design is a multidimensional process. Usually, the design of a game has to include different stages, such as computer science and visual arts techniques. Especially, in the aspect of computer science, even a simple game character needs several computer graphics and Artificial Intelligence (AI) techniques to prototype. So, in general, it is not easy for people without professional game design training to develop a game. In addition, designing games can be considered as just another way of learning while playing. It is a modern way of learning. For example, playing serious games is a very interesting educational activity for teenagers to understand certain topics of science. Designing a game can also enhance the ability of solving diversified science problems for teenagers. However, without programming knowledge, it is not easy to create a game since most game engines and tools require professional computer science knowledge. Consequently, the motivation of our research is to develop a system for people to generate games without programming.

In this thesis, I discuss my research into creating a programming-free game design system, the *Gameme System*. The goal of developing Gameme is to keep creative control in the hands of the game designers, without requiring a lot of custom programming and professional design. There are of course many different aspects that I have to consider in order to build

Gameme, but I have chosen to focus upon what I perceive to be the most essential part of Gameme: AI for games. Nowadays, game AI is something that is *planned* for, something that developers are deliberately making as important as the graphics or the sound effects [Buc05]. My research focuses mainly on the core AI module design since it is the "brain" of the whole Gameme system. I believe, other than the fascinating multimedia experience of game play, that the intelligence of game characters is another important factor in attracting players for games.

Potential users of Gameme are nonprofessional game designers. The more intelligence that Gameme provides, the less work that game designers have to do. However, simplifying the procedure of game design does not mean simplifying game characters' AI performance. A game with truly smart non-player characters (NPC) will attract game players automatically. The game AI research that we are doing should not only significantly promote the intelligent level for games, but should also provide a practical, simple application for game designers. So, in the development of the AI module in Gameme, we present an intuitive way to prototype game agents' behaviour—Goal Oriented Behaviour Design (GOBD) and Goal Oriented Behaviour Trees (GOBT). After game designers create GOBTs for their game stories, the core AI module can automatically generate related knowledge modules which are used in behaviour processing mechanisms. Furthermore, we divert the complexity of game design into the underlying AI core module in Gameme. There is a layered agent architecture which is in charge of real-time behaviour control for game agents. This architecture provides functionalities, such as planning and learning, for game agents.

1.2 Main Scientific Contributions

The background chapters of this thesis may appear to be somewhat eclectic, and other chapters describe several design and mechanisms. Therefore, it is important that I describe what I consider to be the main scientific contributions of the thesis. They can be divided into a general theoretical framework and a few specific experiments. While I consider all the methodologies and experiments described in this thesis to be scientifically sound and of at

least some interest, the two aspects which I claim to be the most important contributions in this thesis are the following:

- Unified goal oriented game agent modeling

  Game agent modeling might vary between different games. However, there are still common features for game agents in most games. Game agents have to interact with each other in order to achieve the playability of games. At least, the minimum interaction is between NPCs and Player Characters (PC). During interactions, game agents should have the ability to sense and react to their environment. The game agent modeling methodology we introduce in Chapter 6 can be used to build game agents in different games. By following this model, game developers can easily add basic autonomous functionalities to game agents during the processes of modeling. In particular, the underlying procedural behaviour knowledge representation is an intuitive design methodology which can be easily understand and used by game designers.

- Planning and learning for game agents

  In this thesis, we present an agent architecture which is in charge of the real-time behaviour control of game agents. There are two essential features for our design for the architecture: it is *layered* and it is *light-weight*. This architecture performs planning and learning mechanisms for game agents, and is based on goal-oriented behaviour design created off-line.

  - Procedural Planning

    Game rendering requires fast behaviour processing in real-time. Some agent architectures in traditional AI prefer to generate all agent behaviours in real-time. However, it is not suitable for game agents since the Graphics and Sound engine needs many system resources to render games. The procedural planning we explain in Chapter 7 is based on goal-oriented procedural knowledge. It is a reactive planning mechanism which mainly selects behaviours from goal-oriented procedural knowledge designed off-line. The planning mechanism not only ensures

game agents have basic reactions during interaction with their environment, but also provides elementary behaviours for game agents, allowing for further learning mechanisms.

- Transfer and adaptive learning for game agents

  Making game agents more intelligent and adaptive to different PCs has attracted the attention of more and more game developers. Sometimes, simple reactive behaviours are not adequate for advanced team behaviour control. We extend the procedural knowledge processing from the planning level to the learning level. Moreover, the idea of transfer learning has been used in team behaviour control. Game agents can learn from their environment and previous experience for the purpose of generating advanced team behaviours.

- Emergent learning for game agents

  Emergence in games is a topic that has interested game designers and developers for many years. It would be fantastic if we could achieve natural and open game play circumstances by emergent behaviours. However, emergence is still not accepted by most game designers since there is a trade-off between off-line game design and real-time emergence behaviours. Game designers have to be very careful about the emergence output in real-time. We introduced our emergence solution in Chapter 9. The emergent learning attempts to let NPCs learn from PCs and to discover the most effective fighting reaction in terms of PC's behaviours. The emergent learning makes NPCs be able to use and combine their basic behaviours to exhibit new behaviours to PCs. The emergent learning allows significantly more freedom and creativity in game play.

1.3 Structure of the Thesis

This dissertation is organized the following way:

- Chapter 2 reviews the general notions related to game design and game AI. We review

related AI techniques based on different application aspects for game agents' control. A personal view of pros and cons is proposed in most sections.

- Chapter 3 provides an overview for modules inside Gameme. In particular, we discuss our design approaches, such as light-weight and layered, for the agent architecture.
- Chapter 4 is the beginning of off-line game design. We introduce the procedural knowledge representation as the behaviour description for game agents.
- Chapter 5 extends on the above remark for behaviour representation to the level of goal-oriented design. Goal orientation is an important feature of most game agents. By combining GOBD and procedural knowledge representation, the data structure GOBT is presented with its characteristics and design methodology in this chapter.
- Chapter 6 provides a detailed definition for game agent modeling. The modeling is based on the AI nature of game agents. We extends the BDI-like agent modeling to a flexible level which can generate additional knowledge modules for game agents planning and learning.
- Chapter 7 starts the explanation about real-time game agents' behaviour processing mechanisms. This chapter provides a detailed discussion of the planning mechanism and related visual testing results.
- Chapter 8 discusses a mechanism and its testing result for game agents' transfer and adaptive learning. Especially, this mechanism can be used in team agents behaviour control in real-time.
- Chapter 9 presents a emergent learning mechanism for game agents in order to enhance the adaptability for NPCs when they face different PCs. Visual testing result is also provided at the end of this chapter.
- Finally, Chapter 10 summarizes the main advances of knowledge that are provided by the present work and concludes on the issues encountered in this study. Possible future projects that would complement and extend this work are also suggested.

Chapter 2

Background

This chapter consists of a review of current notions about artificial intelligence (AI) techniques which are related to game design. The goal is to present a fair description of the field, so the research problem introduced in later Chapters can be fully appreciated. Apart from the general introduction about game designs (Section 2.1), we present AI techniques which are related to our research area in Sections 2.2 and 2.3. In Section 2.6, brief background information regarding AI techniques that we used in our research is presented.

2.1 Introduction to Game Design

Game design is a process of designing contents and rules of a game. It is a multidisciplinary cooperative project and includes different aspects of design. All these concepts are key elements of game design, and make the game design process difficult and complicated.

- Software Engineering: The game is also a software system, and development requires a series of software engineering concepts from design pattern to system architectures.
- Game Logic: The game logic is the brain of a game. It introduces AI into a game and make it attractive to players.
- Interactive Storytelling: Most games have their own background stories, and game

players in games like actors and actresses in stories. So, game design has to adopt some idea from interactive storytelling.

- Multimedia: Multimedia elements are essential if a game is to fascinate players.
- Networking: Playing games with real people is more interesting and challenging than playing with in-game characters. It is the reason that modern game companies put huge effort in developing networked extensions of games.

There are many taxonomies of computer games, and there is no unified way to classify games. Generally speaking, games can be categorized into genres based on:

- Types of goals: Action, Adventure/role playing, Arcade, Strategy, Simulation, Driving and Puzzle.
- Platform: PC, Xbox 360, PS2 and Cell phone games
- Other criteria: serious, commercial

2.1.1 Commercial Game and Serious Game

Games assume different forms, each with different types and degrees of learning [Ber06]. Game playing functionality is not limited to entertainment. Moscovich indicates that games are classified in areas ranging from geometry and logic to probability, topology and perception [MS01]. In addition, war games and sports games can develop player's real time problem solving ability. Computer games have been used to train for serious endeavors from childhood to adulthood for a long time.

In contrast to commercial games, Bergeron [Ber06] defined a *serious game* as an interactive computer application, with or without a significant hardware component, that

- has a challenging goal;
- is fun to play and/or engaging;

- incorporates some concept of scoring;
- imparts to the user a skill, knowledge, or attitude that can be applied in the real world.

The term "serious game" is relatively new in game industry; however, it has been used in projects involving the use of games in education, training, health, and public policy.

Learning while playing is the characteristic of serious games. On the other hand, *Learning while designing* could also let people have the sense of achievement as playing games. Gameme has similar functions that serious games offer, and is the embodiment of the concept of education. It is designed to provide an environment that users could create a game step by step. Especially, Gameme could let users enjoy the process of designing games without worrying about techniques problems. So, Gameme is aimed for a large variety of audiences, including primary or secondary education and unprofessional game designers.

2.1.2 Game AI

Artificial Intelligence has a long history of contributing to game development. Recently, numerous AI methodologies for agent control have been introduced in the game industry. Game AI is an essential element of game-play, and has become an important selling point of games [LvL01, FL02].

Traditional AI research is concerned mainly with finding the best of a number of possible situations. In contrast, the purpose of game AI departs from the major goal of traditional AI. The goal of AI in games is not to compute optimal behaviour for winning against the player but rather to endure so that the outcome is as believable and entertaining as possible [Nar04]. The game AI focuses on creating entities which can act in a "human" way, no matter whether they achieve the highs or lows of human action.

In recent years, the computer games industry has discovered AI as a necessary ingredient to make games more entertaining and challenging and, vice versa, AI has discovered computer games as an interesting and rewarding application area [LvL01]. One of the interesting uses of AI is in the development of autonomous and believable agents in real-time strategy games.

Game agents have to deal not only with the militaristic aspects, but also the societal aspects of the game. Also, level designers and game programmers use A* pathfinding algorithms to steer agents' tracks in maps. A* probably is the most common AI algorithm used in Game AI. It is a heuristic search algorithm which uses a *best-first* search and finds the *least-cost* path from a source node to a target node in a graph. A* can be optimal, and is usually applied in sports game and strategy games.

2.1.3 Game AI Middlewares

How does a developer increase the quality and behaviour believability of the artificial intelligence in games? A number of software companies think they have a solution to that problem: AI middleware [Dyb03]. This section is a review of popular game AI Middlewares.

Some commercial game AI design applications are already launched in the market. Commercial game companies adopt these tools in their game design. These Game AI Middlewares target games that have complex animation and character control needs.

In addition, most Game AI Middlewares provide an intuitive user interface for game designers to work within. Game designers can work on either the programming model or graphical user interface. In addition, some AI middlewares provide scripted programming language for game designers, such as Lua and Python, in game AI design.

Different Game AI middlewares have different expertise. It is impossible to fulfill all the requirement of AI application in games. Middlewares provide functionality in different AI aspects, such as crowd simulation, path finding and animation control etc. Also, most AI middlewares can be integrated with 3D graphics engines. Users of these AI middlewares need professional game design and computer science knowledge.

2.1.3.1 AI.Implant

AI.Implant is one of the AI middlewares created by Presagis. It provides a sophisticated animation control engine that incorporates AI into the game animation development process. AI.Implant empowers game developers through the use of binary decision trees in game

characters behavioral control. AI.Implant is famous for crowd simulation and dynamic path finding. The features of AI.Implant are listed on the Presagis web-site:[1]

1. *The AI.Implant tool chain allows users to easily interactively, automatically or at run-time, author the AI world including a navigation mesh (agent map) and perception data of the simulated world used by agents to make them aware of their surroundings. This world mark-up is used by the agents for dynamic path-finding, path planning and as important meta-data used by agents for complex decision making such as finding cover or hiding.*

2. *The AI.Implant IDE allows creation of rules used for decision making. Based on a sense, think, do paradigm; both the environment attribution and agent to agent interactions are used to perform complex adaptable behaviours even within dynamically changing environments.*

2.1.3.2 Havok Behavior

Havok Behavior is an AI middleware which enable artists to control over state-transition logic and in-game animations which require a lot of programming in general. It empowers artists through the use of graphical hierarchical finite state machines and blending trees. The intuitive user interface and editing paradigm enables artists master this software easily. For example, artists can author complex behaviours quickly by combining numbers of animation assets into graphically created blended trees which are based on finite state machines with branches of different motions.

Havok Behavior is an innovative, cross-platform development system for creating dynamic event-driven character behaviors in a game. Havok Behavior accelerates the development of cutting-edge character performance by coupling an intuitive composition tool for artists and designers with a run-time Software Development Kit (SDK) for game programmers. Together, the Behavior tool and SDK provide "what you see is what you get" results, accelerating the development of cutting-edge character performances for current and next-generation

[1] `http://www.presagis.com/products/simulation/aiimplant/more/aiimplant_for_games/`

game titles.[2]

2.1.3.3 RenderWare AI Middleware

The RenderWare AI Middleware (RWAI) is a set of C++ base classes for AI. It is powered by AI techniques, such as finite state machines and neural networks; and focuses on designing and implementing character behaviour. In addition, RenderWare also provide physics, graphics and path generation modules for game designers.

2.2 AI Techniques for Behaviour Design

AI has been incorporated into games for many years. The motivation that drives AI programmers in the game industry is to create AI that makes the game entertaining and keeps the players playing, ultimately driving the sales of the game [Tog07]. Game developers have used some AI techniques to give seemingly intelligent life to game characters from the early classic games such as *Pac Man* to the modern games such as *Sim City*. The goal in game AI is not to compute optimal behaviour for winning against the player; instead, the outcome of game AI should be as believable and fun as possible [Nar04].

In the broadest sense, most game AI is used to control NPCs. For example, NPCs in *FX Fighter* make use of rule-based AI which allows the computer opponents to recognize patterns in PC's attacks. This implies a learning process for NPCs based on PC's action. Also, the monster in *Half Life* has the ability to watch you, smell you, hear you, and track you. In addition, monsters run away when they lose and fetch reinforcements. The AI used in this game is designed to avoid hard-coded if-then decisions by using a modular AI system which is a schedule-driven state machine, to provide flexibility for monsters.

The game AI research is a little bit disconnected between academic and industrial game AI since these two fields have different goals and use different AI algorithms. Typically, most games still rely on limited AI algorithms to provide the illusion of in-game intelligence. For example, state machine and A* path-finding algorithms is used to control NPCs in the

[2] `http://www.havok.com/index.php?page=havok-behaviour`

majority of games. On the other hand, academic game researchers are interested in using more advanced AI algorithms in game research, such as machine learning, genetic algorithms, and neural networks.

2.2.1 Finite State Machine

A *Finite State Machine* (FSM) or *Finite State Automaton* or simply "state machine" is an abstract machine that can exist in one of several different and predefined states. A FSM also can be defined as a set of conditions that determine when the sate should change. The actual state determines how the state machine behaves [BS04]. A FSM can be represented by a directed graph (digraph) in which the nodes symbolize states and the directed connecting edges correspond to state transitions [Gou88].

We can use several rules as below to define the FSM shown in Figure 1.[3]

CGameObject door;
CGameAttribute open, close;
CGameSate doorState1, doorState2;
DoorState1=door+open;
DoorState2=door+close;

If DoorState1 then DoorState2;
If DoorState2 then DoorState1;

The most popular application of common AI techniques used in games is the FSM. The application of FSMs to game programming can be dated back to the earliest days. Normally, game programmers use FSMs to describe game characters' behaviours. For example, in *Pac Man*, a FSM is used to model the state transition of ghosts.

FSMs are easy and intuitive to describe when dealing with Moore-style machines [Sch04]. Also, since FSMs are straightforward, they are easy to program and debug. One of the most important advantages of FSMs is that they can be acknowledged by non-programmers,

[3] `http://en.wikipedia.org/wiki/Finite-state_machine`

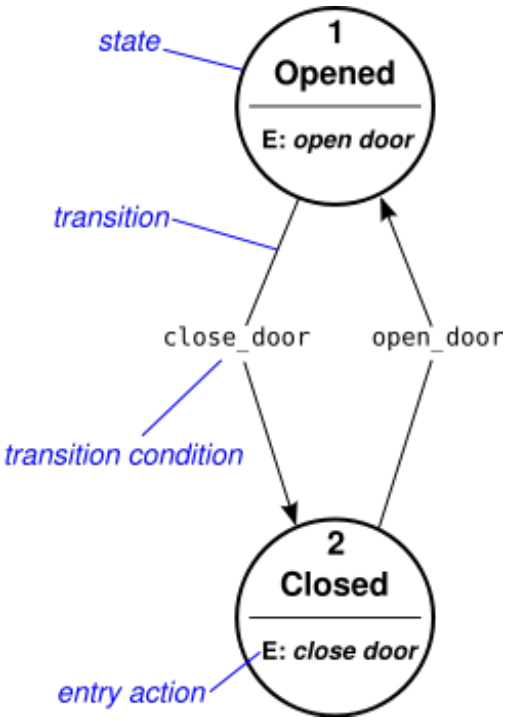

Figure 1: A Example of Finite State Machine

for example, game designers. Game designers can prototype game characters' behaviour as FSMs before game programmers start programming. FSM algorithms simplify game development, especially communication between game designers and programmers.

Nonetheless, FSM have some disadvantages.

- FSMs are not standardized. Game designers have to customize FSMs based on different circumstances, such as goal directed design.
- The state diagram is not useful when states and transitions increase exponentially.
- FSMs provide no easy way to synchronize multiple modular behaviours [Cha07].

2.2.1.1 Varieties of FSM

Based on the pros and cons of FSMs, game programmers use different FSM-related algorithms in games.

- Hierarchical FSMs

Sometimes, a given state in an FSM might be very complex. Hierarchical FSMs provide an effective way to add complexity to a FSM system without unnecessary connectivity. It groups similar states into more locally scoped area. By grouping similar states within their own state machine, the "super state" that contains this new machine can also house common functionality and shared data members [Sch04].

- Behavioural FSMs

  Behavioural FSMs define the animation capabilities of an game object. Each state consists of a collection of motion clips that represent a high-level behaviour, and each directed edge represents a possible transitions between two behaviours [LK05].

- Message and Event Based FSMs

  This FSM algorithm uses messages as triggers than checking transitions in a polling model.

- FSMs with Fuzzy Transitions

  FSMs can be written so that instead of events or some kind of perception trigger causing transition in the machine, fuzzy determination (such as simple comparisons or calculations) can be used to trigger state transitions [Sch04].

- Multiple-concurrent FSMs

  This FSM algorithm is suitable for FSMs between multi-characters and multi-FSMs controlling single character. It is a FSM for synchronizing and coordinating multiple FSMs.

- Data Driven FSMs

  This FSM algorithm can add new behaviours (states and transitions) into the system without much programmer involvement. The constructions of this FSM can be done by non-programmers, such as game designers and producers. There are two approaches of this FSM algorithm, one is scripted FSM and the nother is visual editors.

2.2.2 Fuzzy-State Machines

Fuzzy-State Machines (FuSMs) are built on the notion of fuzzy logic, commonly defined as a superset of conventional (Boolean) logic that has been extended to handle the concept of partial truths [Sch04]. FuSMs combine state machine and fuzzy logic technologies to create agents which can identify and respond to states which are within some decision threshold of a predefined condition [JW01].

FuSMs are an enhancement of FSMs and are often used in simulation games. For example, NPCs controlled by FSMs can avoid cube-shaped obstacles. NPCs controlled by FuSMs can be defined to avoid cube-like objects in a game. Thus NPCs can keep away from any 3D object embedded inside a cube. It is very useful, since typically, game developers use cube as a wrapper, or "bounding box", of 3D objects in collision detection.

FuSMs also provide the ability to model unpredictable elements in games. They extend the usage of FSM in modeling game agents. However, they do not provide the game agent with abilities to sense, react and adapt to their environment.

2.2.3 Rule-Based System

A game is defined by rules that result in a quantifiable outcome. Rules limit player behaviour and define the game, and every game has a quantifiable outcome or goal [SZ03].

Rule-based systems (RBSs) constitute the best currently available means for codifying the problem-solving know-how of human experts [HR85]. One advantage of RBSs is that they mimic the way people tend to think and reason given a set of known facts and their knowledge about the particular problem domain [Sch04]. Another advantage of this kind of RBS is that it is fairly easy to program and manage because the knowledge encoded in the rules is modular and rules can be coded in any order [Sch04].

The RBSs really comes down to a set of "if-then" style rules and a set of facts and assertions [BS04]. Typically, RBSs have two components, working memory and rules memory. The working memory stores known facts and assertions made by the rules while the rules memory contains "if-then" rules. The rules operate over the facts stored in the working

memory. Usually, there is more than one rule associated with each object.

In addition, making inference is also very important for RBSs. There are two basic inferencing algorithms; forward chaining and backward chaining.

The forward chaining consists of three phases:

1. Finding The Rule: *This phase is a procedure of checking the "if-parts" and finding the match rule in working memory.*

2. Resolving Conflict Rules: *During this phase, we have to examine all conflicts rules and find out which one we should fire.*

3. Firing Selected Rules: *In this phase, we fire (execute) matching rules.*

Backward chaining is the opposite of forward chaining in some ways. Instead of matching "if-parts" of rules in working memory, backward chaining attempts to match a rule in the "then-part". This algorithm is goal driven and more difficult to implement than forward chaining especially in fixed rules.

The technique RBS can be combined with FSM in creating the game world. It matches the intuitive way that game designers describe game characters and scenarios in games. However, a huge set of rules in games can cause searching difficulties for matching rules in real-time. Game programmers have to design a efficient data structure to represent rules and a customized algorithm to support real-time searching.

2.3 AI Techniques for Planning and Learning

2.3.1 Decision Tree

The decision tree is made up of connected decision points; the tree has a starting point; for each decision, starting from the root, one of a set of ongoing options is chosen [Mil06]. Decision trees are a hierarchical graph that structure complex Boolean functions and use them to reason about some situations [dB04]. Decision tree is a complement to the RBS in

making inference. Once constructed, a decision tree can also be decomposed into a set of rules [dB04].

Decision trees are used as a analytical and visual decision making tool. In the decision tree each internal node splits the instance space into two or more subspaces according to a particular discrete function of the input attributes values [RM02]. Usually, decision trees are constructed in a top-down manner.

Decision trees have been used to represent classifiers in various disciplines such as statistics, machine learning, pattern recognition, and data mining. The technique was first used in the game *Black & White* to determine the behaviour of creatures. The learning decision tree in *Black & White* allows game agents to learn new behaviour in real-time.

Decision trees provide a fast, easily implemented and understandable decision-making technique. They have advantages such as modularity and ease of creation. Also, the decision tree is designed and used as white box model—if a given situation is provided, the explanation for the situation is clearly replicated and implemented. Unfortunately, each decision tree is unique and has little common structure with other decision trees and therefore is not able to be reused in differing decision-making techniques [dB04]. In addition, decision trees require large amounts of sample data from which to learn, and if the data contains errors, so will the decision tree [dB04].

2.3.2 Behaviour Tree

People use natural language to describe their requirements as a behaviour in a complex system. And the complex system exhibits as a set of behaviours. In general, the behaviour tree is used to design complex systems in a graphical representation. The behaviour tree (BT) provides a way of amplifying our ability to deal with complexity. Each behaviour uses natural language expression and is represented as a tree node in the BT. BTs strictly use the vocabulary of the natural language requirements but employ graphical forms for behaviour composition in order to eliminate risk of ambiguity. By doing this they provide a direct and clearly traceable relationship between what is expressed in the natural language

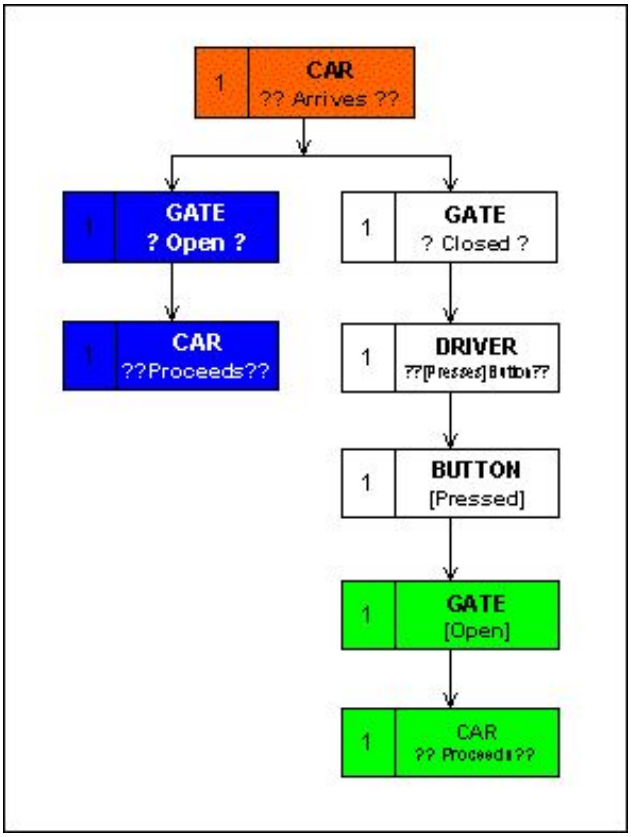

Figure 2: A Behaviour Tree

representation and its formal specification.

The BT has not been widely used in game design. Sometimes, the use of behavioural FSMs cannot fulfill all circumstances of game scenarios. The behaviour tree can be used as a supplementary model in game design and describe complex scenarios in games. In contrast, FSMs are better for describing single game objects' states and transitions; and BTs are better for describing scenarios and storytelling.

For example, Dromey [Dro07] used behaviour tree notation to describe a scenario about "car entering garage", as shown in Figure 2. The whole BT can be described as:

When a car arrives, If the gate is open the car proceeds; Otherwise if the gate is closed, When the driver presses the button it causes the gate to open. [Dro07]

2.3.3 Genetic Algorithms and Programming

Herbert Spencer coined the term "survival of the fittest" in 1864, 5 years after Charles Darwin published *On the origin of Species* in 1859.

Genetic programming, a concept coined by Koza [Koz92], is a variation of genetic algorithms that uses parse trees instead of binary strings to represent potential problem solutions [Pil04]. Genetic programming now routinely delivers high-return human-competitive machine intelligence and can automatically create a general solution to a problem in TEX form of a parameterized topology [Koz07]. Genetic programming can automatically create a general solution to a problem in the form of a parameterized topology whose nodes or edges represent components and where the parameter values of the components are specified by mathematical expressions containing free variables.

There are probably more published academic papers concerning generic algorithms/programming than anyone could read in a lifetime, and the variety of different types of algorithms is dizzying. For this reason we will here limit the discussion to the particular class of genetics algorithms (GA) that is used in game research.

In games, GA and GP (Genetic Programming) are simply a method of finding an optimum solution to a problem [BS04]. The most unpredictable element in game is the player. This allows the game AI to adapt to a situation the game designers might not have been able to predict [BS04]. In addition, GAs have been used with competitive convolution, in which the fitness guiding search is based on the outcome of competition between members of the population [RB96]. This has been applied into pursuer-evader and complex competition games. Also, GAs and other artificial life algorithms were used in games which have to hatch, raise and train NPCs.

2.4 Scripting Behaviour

A scripting language is any programming language that is created to simplify any complex task for a particular program [Toz02a]. Usually, scripting languages are embedded in the

applications they control. "Scripts" are distinct from the core code of the application, which is usually written in a different language, and are often created or at least modified by the end-user [Lou08]. Scripting languages can either be *compiled* or *interpreted*. Scripting files are run from within the game by *virtual machine*.

Scripting language can assist the development process in many ways [Buc05]:

- They can be used as a quick and easy way of reading variables and game data from initialization files.
- They can save time and increase productivity.
- They can increase creativity.
- They provide extensibility.

Scripting languages, such as Lua and Python, are rapidly gaining popularity in game development. Some popular game engines have their own preference in scripting language, such as UnrealScript in Unreal engine.

Usually, scripting language is worked as AI behaviour language which is a complementary programming language to C++ or Java in game development. A more advanced scripting language increase the interaction between the script and the executable, enabling you to not only initialize variables but to create game logic or even game objects, all from one or more script files [Buc05].

Scripts are the technique of choice in the game industry for implementing game AI, because they are understandable, predictable, adaptable to specific circumstances, easy to implement, easily extendible, and usable by non-programmers [Toz02b]. The use of scripting language vary from simple configuration to entire scripted driven engine [Poi02]. In general, scripting language can be used to describe state machines. Actually, most people find it is easier to use scripting language than C++ in representing FSMs and RBSs.

Scripts are generally static and tend to be quite long and complex [Ber02], which can lead to real-time implementation problems. First, due to the the complexity in behaviour description, there could be some weakness in logic. Usually, game players could find out these

weakness easily and find out the way to achieve supposed tough goals quickly. Second, the static scripting behaviour of game agents do not have the ability to forecast unpredictable behaviours from different game players. The static scripting can reduce the playability and entertainment value of games. In addition, since some scripting codes are written by people with very limited programming language skills, programs exhibiting redundancy, inefficiency and deadlock programs often lead to extra work for game programmers.

2.5 Agent Architectures

Agent architectures are blueprints of control systems of agents, depicting the arrangement of basic control components and, hence, the functional organization of the overall agent control system [SA04]. The assortment of architectures used by the autonomous agents community reflects our collective knowledge about what methodological devices are useful when trying to build an intelligence [Bry01]. AI Architectures are typically built to deal with restrictions and constraints by using classical, symbolic, or hierarchical AI models.

Different agent architecture design methodologies are introduced in the history of AI. Most agent architectures are *symbolic*, *connectionist* or *hybrid*. We can classify them based on their foundational distinctions.

- **Cognitive architectures**

  SOAR [LRN86, LRN87], ACT-R [ABB+04], and PRODIGY [VCP+95] are cognitive architectures. These architectures act like certain cognitive system, and try to simulate and understand human cognition. Usually, they not only cognise different intelligent behaviours, but cognise as a whole. They are not parameter tuning systems. Some of them only focus on internal information cognitions, and are lacking in perception, reaction and learning functionalities from external resources.

- **Layered architectures**

  Subsumption architecture [Bro91], ICARUS [DSAS03], and GRL [Hor00] are layered

architectures. These architectures are widely used in the area of simulation and robotics.

Subsumption architecture was the early cornerstone of layered architectures. Subsumption architectures provides a way to decompose intelligent behaviours into a set of simple behaviours. In general, it is a bottom-up architecture which allows agents to operate with increasing competence. For example, in robot behaviour control, low layers of this architecture can provide fast-adapting mechanisms such as reflexes while higher layers take care of overall goal achievement. Modularity is an important feature of this architectures. It not only has modular structure but also generates intelligent behaviour on a modular basis.

2.6 Related Agent Planning and Learning Techniques

2.6.1 Procedural Reasoning Systems and BDI Agents

The Procedural Reasoning System (PRS) was first created to control autonomous robots. Georgeff, Lansky and Schoppers [MPG87] described the PRS as a system for reasoning about and performing complex tasks in dynamic environments, and showed how it could be applied to control an autonomous mobile robot. The reasoning system that controls the robot is designed to exhibit the kind of behaviour expected of a rational agent, and is endowed with the psychological attitudes of belief, desire and intention [GL87]. The Procedural Reasoning System integrates both reactive and goal-directed deliberative processing in a distributed architecture [Jon08]. Within this system, a specific detection strategy can be pursued based on available situational data, but rapidly modified or replaced in response to incoming information.

The PRS used the Belief-Desire-Intention (BDI) software model. At any instant, the actions being considered by PRS depend not only on its current desires or goals, but also on its beliefs and previously formed intentions [MPG87]. The PRS has the ability to reason about its internal and external status, and reflect it on modify its desires and intentions.

The PRS provides agents ability in surviving in dynamically complex environments.

The PRS is a reactive planning system which consists of five components [Mye01]:

- A database containing current *Beliefs* or facts about the world;
- A set of current *Goals* to be realized;
- A set of plans, called *Acts*, describing how sequences of actions and tests may be performed to achieve certain goals or to react to particular situations;
- *Intentions* containing those plans that have been chosen for (eventual) execution;
- An *Interpreter* that manipulates these components, selecting appropriate Acts for execution based on the system's beliefs and goals, creating the corresponding intentions, and then executing them.

In general, the PRS is associated with BDI agents [RG91b, RG95, Bra99]. The BDI software model is a reasoning architecture designed for programming intelligent agents. The BDI agent model includes *Beliefs*, *Desires*, *Intentions*, and *Plans*.

PRS and its variants exist both as a planning engine and as a set of development tools [Bry01]. Many academic researchers appreciate the easy programming and modularized structure of PRS in their projects. Implementations of the PRS and BDI agent abstract semantics include dMARS [dKLW97], RCS [GI90], IRTNMS [RG91a], and others. Application areas include diagnosis for space shuttle, factory process control, business process management and network management monitoring.

However, there are still some limitations of PRS and BDI models.

- BDI agents lack of learning competence. The BDI agents' plans are based only on its current resources such as beliefs, desires and intentions. They cannot look forward or learning from their experience.
- BDI agents lack the ability to adapt successfully to changing circumstances.

- There is no explicit mechanism for multiple agent reasoning. The BDI agent model does not provide a clear structure for inter-agents communication.

Some researchers propose to use BDI agent alone without embracing the PRS. They use the BDI model as the representation of agents which embed in hierarchical reasoning structure. The design of multiple layer reasoning system is varied based on different application purposes. Usually, each layer of these these hybrid architecture has different reasoning concerns. For example, Sloman and Logan [SL98] designed an architecture which is describable in terms of the "higher level" mental concepts applicable to human beings. This architecture has three layers: the reactive layer; the deliberative layer; and the reflective layer.

2.6.2 Transfer Learning

Learning in one context or with one set of materials impacts on performance in another context or with other related materials [PS92]. It is a fundamental human capability. Computer scientists have been adapting this educational theory for computer science research for decades. For example, in the early stage, transfer of learning was described using production modeling, introduced into the analysis of human computer interaction by Card, Moran and Newell [CNM83]. A production model describes actions at the interface in terms of a set of if-then rules, in which a user recognizes some condition and then performs an appropriate action [Tet87].

As an AI research topic, the transfer learning leverages learned knowledge from a *source task* to a related but different *target task*. Recent work in transfer learning has succeeded in making reinforcement learning algorithms more efficient by incorporating knowledge from previous tasks [TKS08]. Formally defined, reinforcement learning is the learning of a mapping from situations to actions with the main objective being to maximize the scalar reward or reinforcement signal [Sut88]. Informally, reinforcement learning is defined as learning by trial-and-error from performance feedback from the environment or an external evaluator [Eng07]. If reinforcement learning algorithms can learn new tasks from limited experience, agents may be able to learn reliably on-line in the real world [TKS08]. One approach to

enabling such learning is to employ transfer learning to reuse knowledge gathered in previous tasks to learn a novel task better or faster [TKS08]. Transfer learning usually provides either a full model of different tasks or an explicit relation mapping of *source task* into the *target task*. Rather than having to learn each new task from scratch, the goal is to enable an agent to take advantage of its past experience to speed up new learning [Sto07].

Behaviour creation for AI game agents typically involves generating behaviours and then debugging and adapting them through experimentation [OR08]. This is typically a complex and time-consuming process that requires many iterations to achieve the desired effect [OR08]. For the developer of game AI, the use of transfer learning means that less time is spent in the design, coding and debugging in multi-agent behaviours; instead, more time is spent in "teaching" game agents to behave from previous or related experience. Banerjee and Stone present the idea of transfer learning on knowledge basis by using the technique of value-function transfer where general *features* are extracted from the state space of a previous game and matched with the completely different state space of a new game [BS07]. Sharma et al. present a multilayered architecture, which used a novel combination of Case-Based Reasoning and Reinforcement Learning, to achieve transfer learning while playing against the game AI across a real-time strategy game [SHS+07].

2.6.3 Adaptive Behaviours

In general, research on adaptive behaviours places emphasis on mechanisms that can be expressed in computational, physical, or mathematical models. In addition to elaborated and human-specific abilities, adaptive agents explicitly take into account environmental feedback. Research in adaptive behaviours is an approach complementary to traditional AI, in which basic abilities allow agents to survive and perform their mission in unpredictable environments.

The adaptive mechanism was first introduced in the area of intersection of computer science and commerce by Sandholm [San03]. Pardoe et al. [PSSTT06] explored the use of an adaptive auction mechanism: one that learns to adjust its parameters in response to

past empirical bidder behaviour so as to maximize an objective function such as auctioneer revenue.

We apply adaptive behaviour to NPCs' modeling for the purpose of creating human-like intelligent game characters. NPCs have to be attractive by showing unpredictable behaviour when they face different PCs. Scripting non-player characters is typically labor-intensive and results in simplistic NPC behaviours [CSS+06]. Hard-code off-line behaviours cannot guarantee NPCs' real-time adaptive intelligence. The capability of presenting unpredictable behaviours requires NPCs to be able to adapt to a dynamic game environment. Furthermore, one of the key impact on game environment is PCs' behaviours. Their in-game strategies directly lead to diversity of real-time game scenarios. Therefore, the adaptation of NPCs is a topic focused on behaviour adaptation relative to PCs' in-game behaviours. In general, the off-line NPCs' modeling are not adequate, the real-time game AI system plays an important role in creating NPCs' adaptive behaviour.

2.6.4 Complex Adaptive Systems

A system is an assembly of elements hooked together to produce a whole in which the attributes of the elements contribute to a behaviour of the whole [Jon03]. A complex system is any system featuring a large number of interacting components (agents, processes, etc.) whose aggregate activity is nonlinear (not derivable from the summations of the activity of individual components) and typically exhibits hierarchical self-organization under selective pressures [Roc99]. Before being studied in computer science, complex systems were investigated in physics, biology, and even philosophy. Complex Adaptive Systems (CAS) are special cases of *complex system*. The CAS is a system of units which has reactive, goal-oriented and planning abilities. CASs require agents that can observe and learn from their environment and adapt to it. In general, examples of CAS includes the human immune system, ecosystem, weather system and others which are constantly adapting to their environments.

In general, explicitly analyzing the input and output data is a way to understand mechanisms of a system. CASs have the feature of simple input but amplified output. Many

CASs have the property that a small input can produce major predictable, directed changes in an *amplifier effect* [Hol95]. The output is the global behaviour that the CAS presents. The behaviour of a whole CAS abounds in nonlinearities which do not satisfy the superposition principle. In most cases, CAS are characterized by behaviours such as emergence, adaptation, self-similarity and self-organization.

Agents are interconnected units in CASs. They interact with each other and their environment in an unplanned and unpredictable way. Mass interactions of agents can lead to the emergence of regularities and start to form a pattern which feeds back to the system and further iterates them. Emergence gives agents opportunities to learn from their experience, improve further activities and adapt to their environment. The adaptation ability of agents is an evolutionary ability which allows agents to live in their environment.

The CAS is a model for thinking about the world around us; furthermore, it can be a model for designing virtual reality world in games. Nowadays, games have become more and more complex in the way that they simulate different reality systems. We consider shaping characteristics of CASs into building game agents. By cross-disciplinary comparison of different CASs, we can abstract some general models of CASs into game design in order to create intelligent game agents. In this thesis, we introduce an autonomous adaptive game agent modeling theory along with a layered agent processing architecture for the purpose of creating autonomous adaptive game agents.

2.6.5 Emergence

Over 2,000 years ago, Aristotle first recognized the profound concept that *a whole can be more than the sum of its parts*. Emergence is what happens when an interconnected system of relatively simple elements self-organizes to form more intelligent, more adaptive higher-level behavior [Joh01]. The intuitive way to explain emergence is the way CASs and patterns arise out of a multiplicity of relatively simple interactions. Emergent behaviours cannot be predefined when we design systems, and they are not predicable in low level of system control, but occur at higher levels of system control. Systems that exhibit emergence have

a common set of elements, such as agents, rules or regularities. Simple sets of rules applied to large-scale systems can result in complex high-order organizations, such as an ant colony and world wide web.

We are everywhere confronted with emergence in complex adaptive systems—for example, ant colonies, networks of neurons, the immune system, the Internet, and the global economy—where the behavior of the whole is much more complex than the behavior of the parts [Hol00]. In general, techniques that can be used to explore emergence in various complex systems include AI and machine learning. In addition, some techniques that already or can be used in emergent phenomena of games include decision trees, neural networks, flocking, and evolutionary algorithms.

Emergence can be applied to games can be in various ways. Sweetser mentions that emergence can be embedded in the development of game world, characters and agents, emergent narrative and social emergence [Swe07]. We study the emergence for intelligent agents in games. It is based on the concept of Agent Based System(ABS). ABSs are based on the idea that the complex, global behaviour of a system is derived from the collective simple, low-level interactions of the agents that make up this system [Swe07]. Agents interact in the game environment, forming more complex behaviours as a collective. In games, the conditions of having emergence is to let game agents be capable of learning and adapting to their environment. In this thesis, we introduce a mechanism to offer learning abilities for game agents in order to let them adapt to their counterparts' actions.

2.7 Discussion

Game design applications are very popular in commercial or serious game development. People adopts various AI techniques in game development in order to have impressive game performance. In this chapter, we discuss AI techniques that have been used in Game AI. However, these techniques are not completely suitable for our research approach. We examined techniques and chose some of them for our use. In later chapters, we customized these AI techniques to what can actually been done in practice for Gameme.

Chapter 3

System Design

This chapter discusses system design methodologies for Gameme. Section3.1 provides an overview of the modules of Gameme. Section3.2 describes two design patterns that we used in the development of Gameme. In Sections3.3, 3.4, and 3.5, we describe the design and methodological choices we made to whole the agent architecture in Gameme.

3.1 The Modular Structure of Gameme

The Gameme system that we are developing is a game design application. Potential users of Gameme are non-professional game designers who may not have any programming knowledge. As shown in Figure 3, the Gameme system uses a modular architecture consisting of three components: the User Interface(UI) module; the Core AI module; and the Graphics & Sound module. The advantage of a modular architecture is that we can develop, implement and maintain various components as stand-alone programs. The architecture allows multiple parallel processing development cycles for the modules and the use of off-the-shelf products without having to alter their source code[Szi07].

We use AI techniques in describing the whole game logic, which includes game agents' profiles, relationships among agents, etc. A game created by Gameme is a set of game logics which is a ramification of the core AI module in Gameme. The core AI module, performing

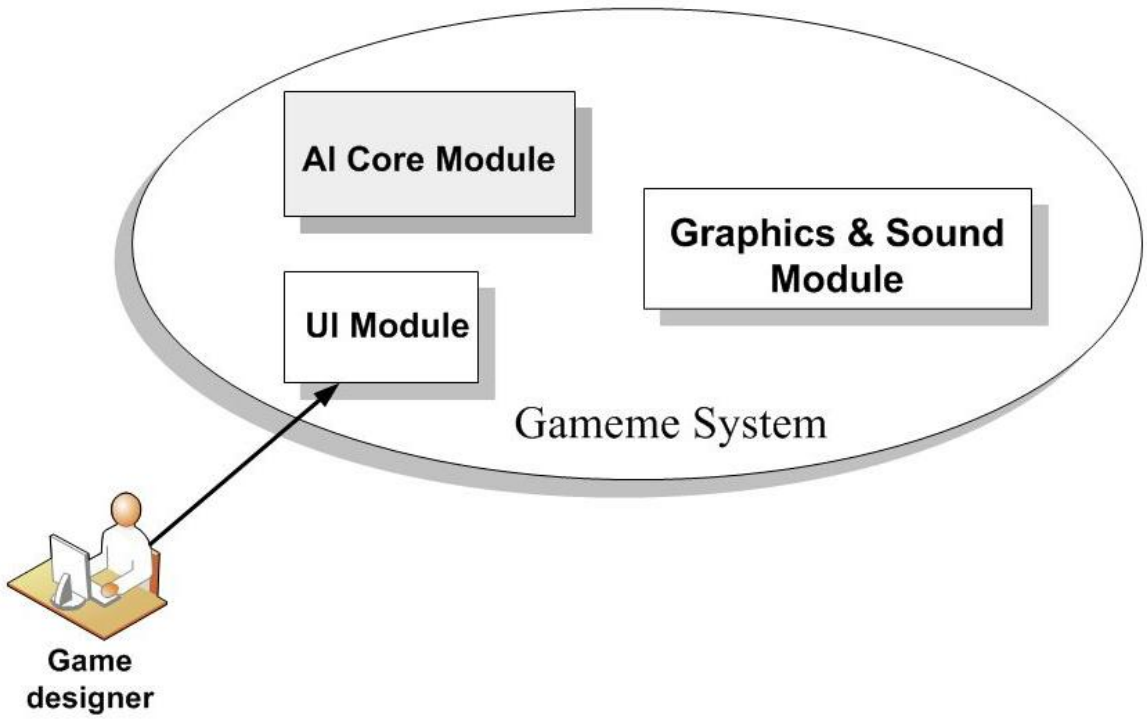

Figure 3: The Gameme System

as a AI middleware, interacts with the UI module and the Graphics & Sound module. The idea of the core design is that the core is composed of a set of discrete sub-modules, such as the perception module, the knowledge module, the control module and the action module. These modules perform together as generators and controllers of the game logic. The detailed description of each module in the core AI module is in Section3.3.

In Gameme, there are two controls over game agents: off-line game agents' behaviour designs and real-time behaviour control. The core AI module takes on most of the tasks for these two controls. During the off-line design, pure game logic is created based on the game designers' ideas. The pure game logic describes game agents in a particular knowledge format. It represents predefined game agents' profiles, and relies on the core AI module to implement it. In addition, the core AI module not only generates knowledge representations of game agents, but also controls logical relationships among game agents. The real-time control is game environment changes reflected on game agents' behaviour change during game play. The behaviour control is mainly done by planning and learning mechanisms in

the control module of the core AI module. Planning and learning mechanisms implement real-time controls over the knowledge module in order to output intelligent behaviours for game agents. Furthermore, during game rendering, the pure AI logic data is sent to the Graphics & Sound engine for multimedia rendering. Thus a game player can view a real 3D graphics game characters with different behaviours.

Figure 4 and Figure 5 depict communication among modules, game designers and game players.

- In offline design, game designers design games by using the UI. The UI design data simply describes UI objects that created by the game designers. The UI module sends UI design data to the AI core module. The AI core module generates pure game logic based on information transfered from the UI module.

  During offline design, game designers could also test game scenarios or the whole game. The graphics & sound module receives the pure game logic and renders the general game environment. Importantly, during interactions of NPCs and PCs, the core AI module sends their decisions of NPCs' behaviours to the graphics & sound module for visual rendering result.

- In real-time, game players play the game by controlling PCs. Behaviour changes of PCs affects behaviours of NPCs since the core AI module makes reasoning decisions for NPCs based on PCs' status. The Graphics & sound module gets information from the core AI module, and renders the pure game logic data in the desired multimedia format.

3.2 Design Patterns used in Gameme

- **Model-View-Controller Pattern**

  The Gameme architecture uses the *Model-View-Controller* (MVC) pattern. The Model component is the pure game logic created by Gameme. The View components are representations of the game seen from different perspectives. The Controller component

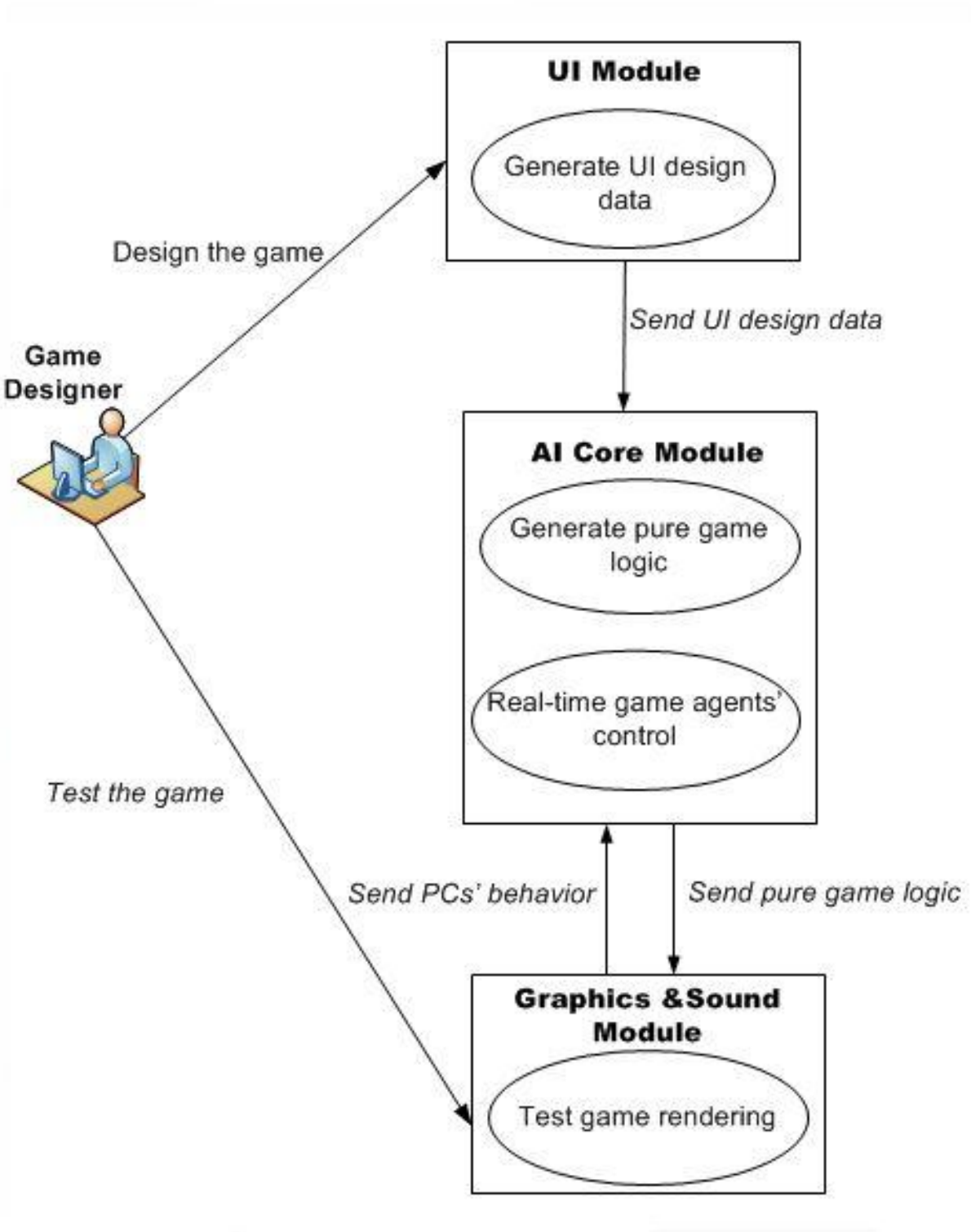

Figure 4: The Offline Design

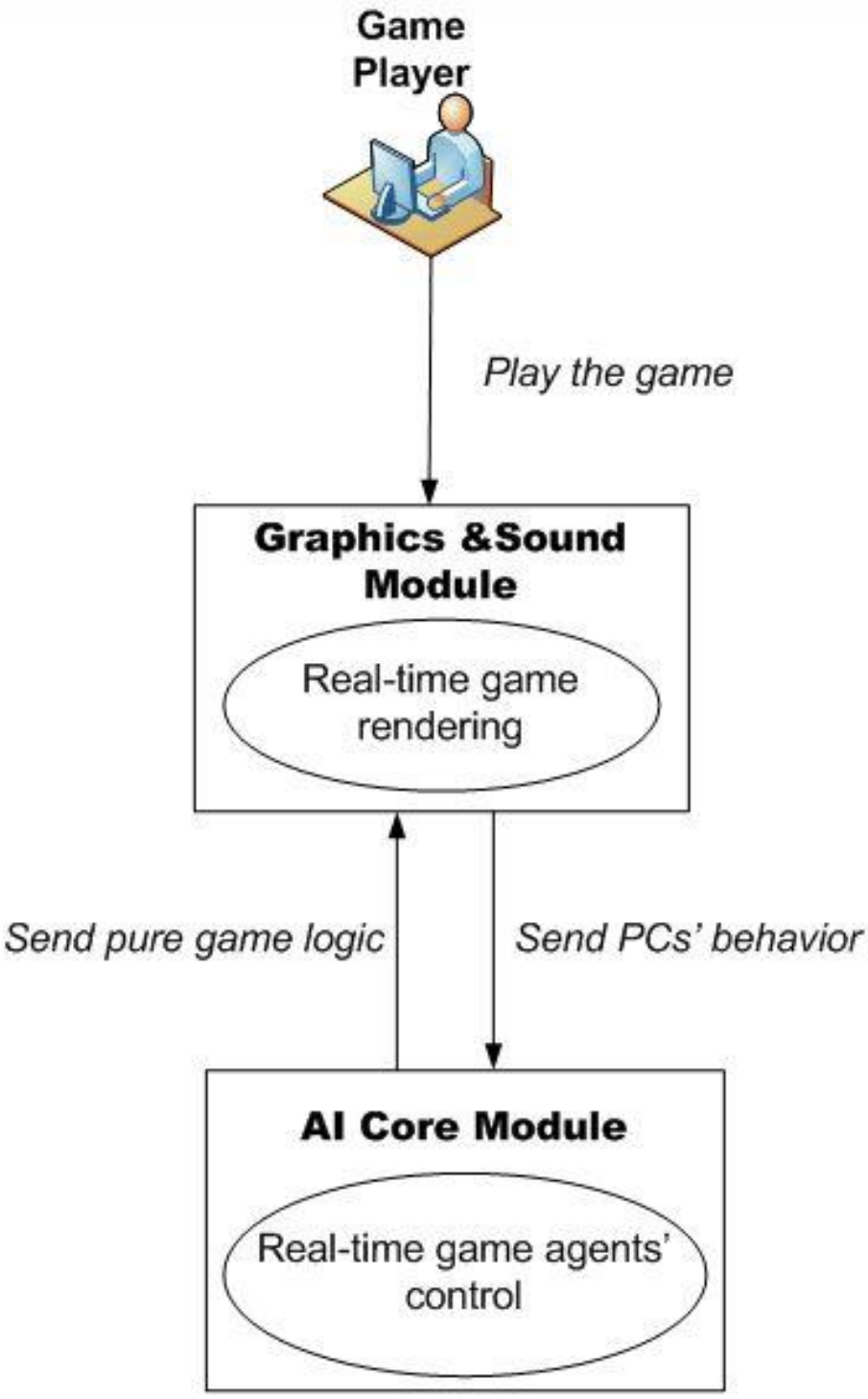

Figure 5: The Real-time Control

sends messages to the model and the views. The "one model, multiple view" represented in Gameme is described below.

- One model

  The pure game logic is used to represents the whole game world. It could be considered as the original model of the game world. It not only represents game agents' behaviour but also represents relationships among game agents. It exists in the core AI module.

- Game Designers' view

  Game designers have a designer view of the game. They design games by editing graphics elements in the UI. The designer view can be either a set of game logic which expresses the pure logical relationship of game contexts in graphical representations, or the rendering of particular scenes or scenarios in a game with editing options.

- Game Players' view

  The player view, on the other hand, is the rendering of the whole game without any evidence of Gameme. The rendering is the execution of Graphics & Sound engines which are controlled by the pure game logic from the core AI module. Game players can only manipulate player characters in order to let the game story continue.

- **Adapter Pattern**

  In order to let the AI module communicate with the 3D graphics engine, we use the *Adapter pattern* to translate knowledge from the pure logic format to the format that 3D graphics engine can understand, or *vice versa*. From the object-oriented point of view, we design the this adapter pattern as a multi-inheritance class structure as in Figure 6. The class CXman inherits from both class CGameCharacter and class

COgreCharacter. The class CGameCharacter is associated with the Gameme AI module; the class COgreCharacter is associated with the OGRE 3D Graphics engine.[1]

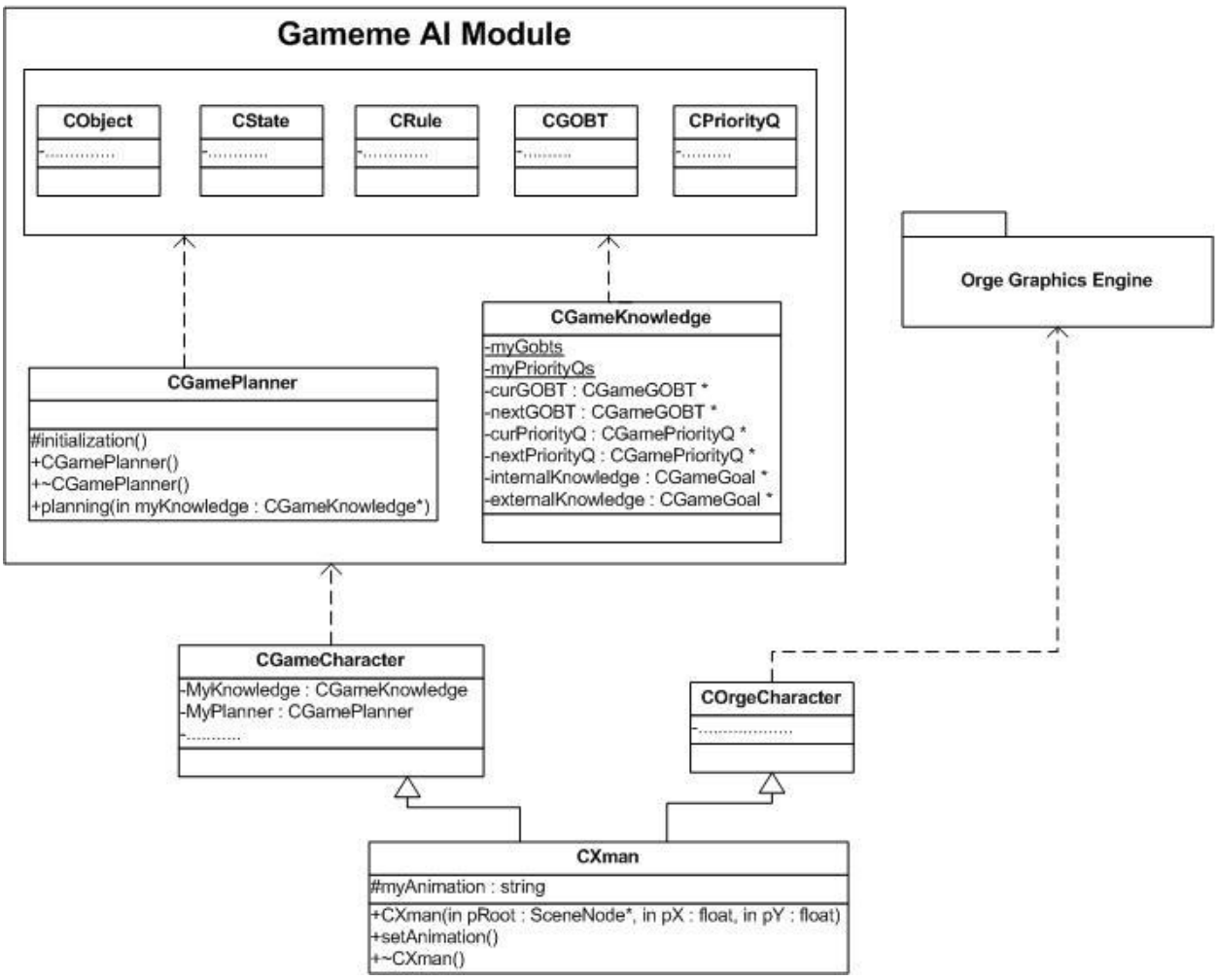

Figure 6: The CXman class inherits features from the Gameme AI core module and the OGRE 3D graphics engine.

3.3 The Model of Agent Architecture

We defined an agent architecture model, shown in Figure 7, to be used in the design of the planning and learning system. This architecture was inspired by Nilsson's Teleo-Reactive Architecture [Nil01]. There are three parts of this architecture: the *perception module*;

[1] `http://www.ogre3d.org/`.

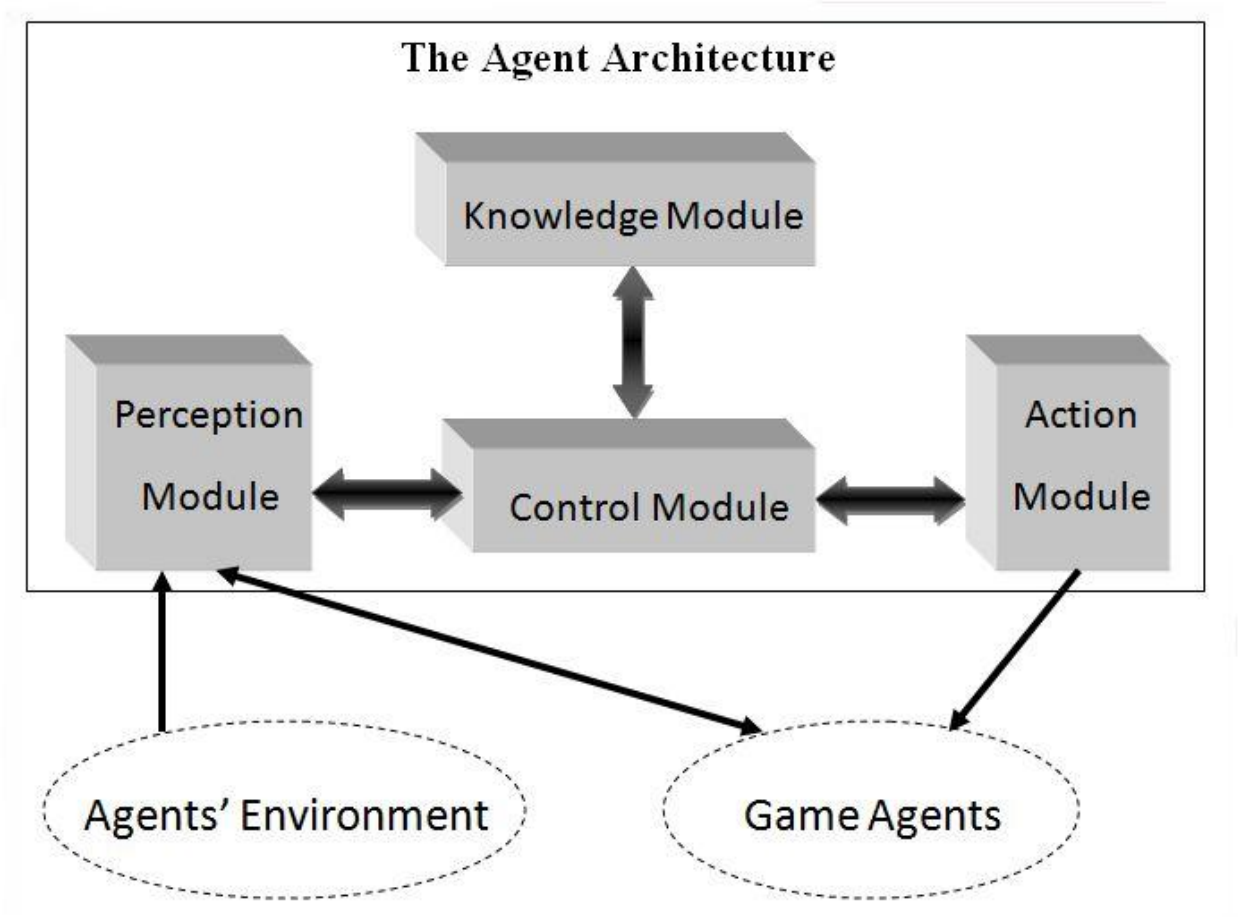

Figure 7: The Model of Agent Architecture in Gameme

the *control module and knowledge module*; and the *action module*. These modules combine the functionalities of sensor, goal processing, decision making, and sending. The sensor functionality is to perceive state changes in each agent and deposit these perceptions into the architecture. The goal processing and decision making functionality consists of evaluating agents' statuses and making decisions for agents. Decisions are sent to agents regarding their next execution actions.

The agent architecture runs in cycles. In each cycle:

- The **perception module** accepts information from working memory. The game environmental information, game agents' information and states are sent to the perception module. For example, the input information includes agents' position, current state of

agents etc. In addition, since the perception module is directly used in connecting with Graphics & Sound engines it should have the ability to fill data. It fills data to pure game logic data, and sends these data to the control module for the agents' planning and learning.

- The **knowledge module** mainly contains off-line design knowledge for game agents, and can be combined dynamically and hierarchically to increase the flexibility and adaptability of game characters either in off-line or in real-time.

- The **control module** include planning and learning mechanisms which operate on data in knowledge modules in order to manipulate game agents' real-time behaviours. It is the key portion in the agent architecture. For example, in agents' planning processes, the control module allocates information to preloaded rules and priority factor tables; proposes possible actions based on rules. Finally, it decides which game state is to be executed in this cycle and sends it to the perception module for memorizing and to the action module for sending.

- The **action module** is similar to a trigger for agents' activity. It sends decision to agents. For example, the decision might define the next state in a goal for an agent.

The model for the game agent architecture design is extended in the planning and learning processes design. One important feature is that we separate the knowledge module with perception, control and action modules. Only the control module has the ability to access data in the knowledge module. It is different from other agent architectures which embed sensing and acting functionalities in the representation of knowledge. We consider the Gameme system is a game design system that needs to be improved step by step. We might need to add more modules into this system. Keeping different modules with explicit functionalities can be convenient for further extensions. In addition, keeping the knowledge module isolated from perception and action modules can avoid retrieving data from outside of game agents unnecessarily.

3.4 Light-Weight Agent Architecture

The core AI module needs to complete planning and learning for the entire game world without interrupting the overall performance of the game. The trade-off between off-line design and real-time goal reasoning is the key point. The agent architecture in the core AI module must therefore be "light-weight" in the sense that it uses minimal resources. A light-weight and functional agent architecture is the most important aspect in the agent architecture design. It is designed to have planning and learning in real-time as soon as possible.

The better the arrangement of predefined pure game logics, the lighter the real-time game rendering architecture can be. For instance, GOBTs are based on lower level production rules in the knowledge module. This data structure arrangement make transformations and searches among them very convenient. Also, the execution order of GOBTs for each goal is made off-line, which also reduces time spent in real-time planning at certain levels. For example, if the control module decides on no change of current goal for an agent at runtime, no search for the next action is necessary. The agent only has to execute the next state in the execution order of the current goal. In addition, the data processed in the PPS is also important in the light-weight reasoning. The pure game logic data transfer in the whole agent architecture is made to be as simple as possible. Most of them are primitive data types; string variables are used in many places. Theses data will be transfered into the format that Graphics & Sound engines can understand through an adapter program before rendering in a multimedia style.

3.5 Layered Planning and Learning

Multiple agents' planning and learning should prepare for the unexpected, and have adaptive interaction protocols. For preparing for unexpected situations, game agents should have the ability to react in cooperate or competition scenarios. For preparing adaptive interaction protocol, game agents should have the ability to learn from their mistakes and build up their

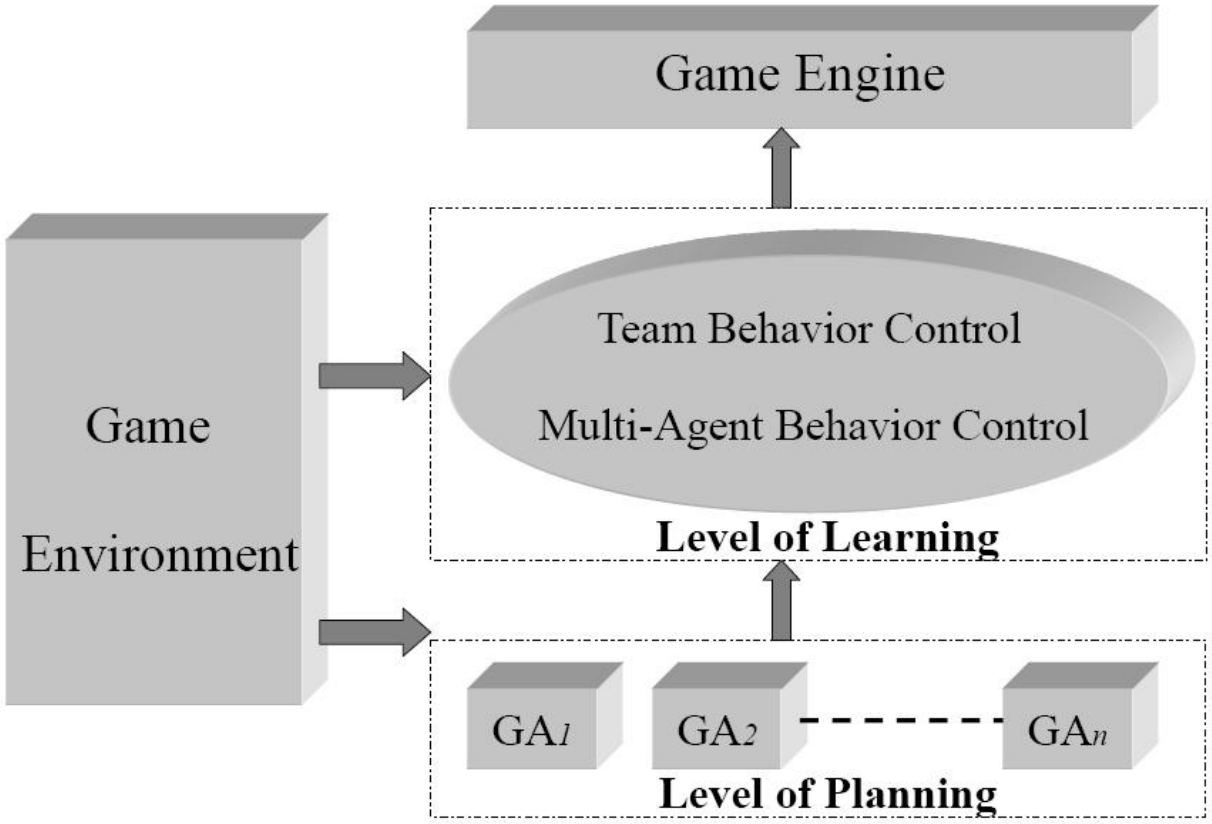

Figure 8: The Layered Planning and Learning for Game Agents(GA_1 to GA_n)

own experience to be able to act smarter in time.

Layered learning is a multi-agent learning paradigm designed to allow agents to learn to work together in a real-time, noisy environment in the presence of both teammates and adversaries [SV98]. In our research project, we present a multi-layer architecture for game agents' planning and learning as shown in Figure 8.

In the layered architecture, the lower level focuses on the individual game agent's behaviour planning. The upper level focuses on game agents' learning in order to adapt the game environment. The important feature of this architecture is that knowledge planned in one layer feeds into the knowledge in the next learning layer. The learning level revises the planning outcome by processing it in adaptive and transfer learning algorithms in order to provide dynamic performance for the team of game agents. In addition, the emergent learning mechanism analyzes game players, and lead NPCs to have intelligent strategies versus

PCs.

In addition, the hierarchical layered learning structure adopts the modularity advantage of the subsumption architecture [Bro91]. The subsumption architecture layers control modules, allowing high-level controllers to override lower-level ones [Sto07]. The modular based bottom-up planning and learning architecture divides complex team behaviour control into manageable individual agent behaviour control.

Chapter 4

The Behaviour Design

4.1 Offline Knowledge Design

Game design is a complicated and time consuming process. After game designers describe the detailed characteristic of each type of game characters, the game developers have to formalize these character descriptions by using a unified knowledge format. This knowledge is the individual information embedded in each game character. The knowledge is usually a substitute for the game character in computer programs.

In order to have dynamic planning and learning results, knowledge defined off-line is essential for game agents' behaviour control. Since limited time is available for agent planning and learning in games, we do not propose that agents perform much searching in real-time. So off-line knowledge design is very important because it provides the foundation for real-time planning.

In addition, we do not apply dynamic processing at every level of the goal planning and learning. For certain low-level goals, our implementation is based on static execution orders. This is also a consideration of design of a light-weight agent architecture. So, knowledge modules in Gameme should be defined off-line and retrieved and parameterized by the interpreter and coordinator in real-time.

4.2 Knowledge Representation for Game Agents

A game world is a virtual world that contains different characters. Most of the convincing performance given by NPCs in games come not only from their outward appearances and animations but also from the programming that drives their behavior [dB04]. During game development, developers have to translate game designers' idea into computer programs in order to make the virtual world vividly present in computers. First, we have to find a way to describe characters in the virtual game world.

The knowledge representation is a set of ontologies or symbolic representations about the game world. It is a medium between the virtual human design game world and the real data computation. A suitable knowledge representation for game agents leads to intelligent agents behaviours in real-time. The knowledge not only describes game agents' behaviours, but also is the foundation of planning and learning for game agents. It affects the efficiency and correctness of agents' behaviour control in real-time.

As Sowa [Sow00] defines it, knowledge representation is a multidisciplinary subject that applies theories and techniques from three other fields:

1. *Logic* provides the formal structure and rules of inference,
2. *Ontology* defines the kinds of things that exist in the application domain,
3. *Computation* supports the applications that distinguish knowledge representation from pure philosophy.

Agent is a term borrowed from philosophy, meaning an actor or an entity with goals and intentions that brings about changes in the world [Bry01]. In our research project, game agents are NPCs and PCs. We designed a knowledge representation to describe game agents and relationships among them. In detail, the knowledge representation is used to describe behaviours of game agents.

4.3 Behaviour Design

In general, the term *Behaviour* refers to an object's action or reaction. Behaviour representation is the foundation of the description of the whole game system. A game agent's behaviours are related to its environment which refers to its neighboring game agents. When people are playing games, game agents' behaviours is presented as the result of interaction among PCs and NPCs. However, these behaviours are not generated in real-time but defined off-line. Each game agent has a set of predefined behaviours represented by certain knowledge representation. In real-time, planning and learning mechanisms select and combine appropriate behaviours for game agents.

We used Behaviour-Based AI (BBAI) as a reference to the behaviour design in our system. BBAI is a methodology concerning modular decomposition of behaviours. Brooks's subsumption architecture was one of the earliest attempts to describe a mechanism for developing BBAI. The BBAI is the decomposition of intelligence into simple, robust, reliable modules [Bry03]. In Brooks's original proposal, these modules are finite state machines organized into interacting layers, which are themselves organized in a linear hierarchy or stack [Bry03]. However, the BBAI is not suitable for complicated behaviour reasoning and multiple agent systems, since it is strongly related to reaction planning.

Based on the idea of BBAI, we consider that complicated game agents' behaviours could be depicted as a combination of several simple behaviours.

- It is a modular design methodology which is akin to object-oriented design. Game designers can create different complex behaviours based on a set of basic behaviours.
- Our agent planning and learning mechanisms operate between modular behaviours.
- The decomposition of complex behaviours into atomic behaviours can be easily manipulated by game designers and engineered by game programmers.

4.4 Procedural Knowledge and Behaviour Design

The game environment is much simpler than the real world in which we live. However, it is still too dynamic to anticipate all circumstances. Game knowledge has to be represented in certain particular concise format. We have to consider how to arrange simple behaviours together when design games. This is an important distinction because representing knowledge is only useful if there's some way to manipulate the knowledge and make inferences from it [Jon08]. By finding the relevant planning and learning procedures, game agents may stand a better chance of achieving its goal effectively. So, the way to represent the knowledge has to be based on the most effective way to use it. Since Gameme uses a procedural planning and learning system to control game agents, representing knowledge as procedures is a suitable solution.

4.5 Key Steps for Defining Behaviour

In this section, we discuss several key steps that must be taken when people define behaviour based on procedural knowledge. Designing intelligence for game agents requires *identifying* the behaviour, finding out *how* and *when* to do it, and *what* is necessary to do.

4.5.1 Identify the Behaviour

When we design a behaviour, it is not enough to observe the action or reaction of game agents. In addition, it is important to identify the behaviour. Moreover, we must have an explainable relationship between the cause and result of the behaviour. In some situations, cause and result behaviours could be added to the procedure.

4.5.2 How

Knowledge about "how" to do something is procedural knowledge [TA98]. It is about the procedure used to carry an action out. It is an intuitive way to describe a *plan* for game

agents to react to their environment. The way to design each module in a procedure is a process of *behaviour decomposition*.

Game designers can organize a set of simple behaviours into an execution procedure. For example, in order to present "how a cat eats a fish", we can describe that in order to "eat the fish", the cat has to follow a course of behaviours which are "look for the fish" and "find the fish". This procedure can be described as in Figure 9.

Figure 9: The Behaviour "How a cat eats a fish ?"

Using procedural notion for behaviour design is not only appropriate to off-line game design but also particularly important for reasoning about how to achieve a given goal. By directly setting the particular procedure in game design, the game agent can effectively access steps for particular goals in real-time game rendering.

4.5.3 What

After the behaviour decomposition, there is a series of behaviours listed in a procedure. We can consider each behaviour in the procedure as a module. Game designers have to decide *what* granularity that each module in the procedure would be.

The way to decide Level of Detail (LOD) of each module in a procedure is a process of *behaviour specification*. LOD was introduced by Clark [Cla76]. This technique describes how to represent a 3D object at different levels of complexity as it moves away from the viewer or according to other metrics. The metric we used to specify the LOD of each module in a procedure is making each module have the same level as the submodule under it.

The fundamental problem of using a modular approach is deciding what belongs in a module—how many modules should there be, how powerful should they be, an so on [Bry03]. It is a design issue also occurs in Object Oriented Design (OOD). Different people might

have different ways to decompose behaviours. In general, we suggest that game designers should decompose behaviours based on the same LOD. For example, we can have another way to describe "how a cat eats a fish" in Figure 10. However, the new behaviour "MEW" does not represent a necessary step in the procedure of "How a cat eats a fish". Or, we can say the behaviour "MEW" does not have the same LOD as other behaviours in the same level. The solution is we could add the behaviour "MEW" as a sub-behaviour in the behaviour "Find the Fish".

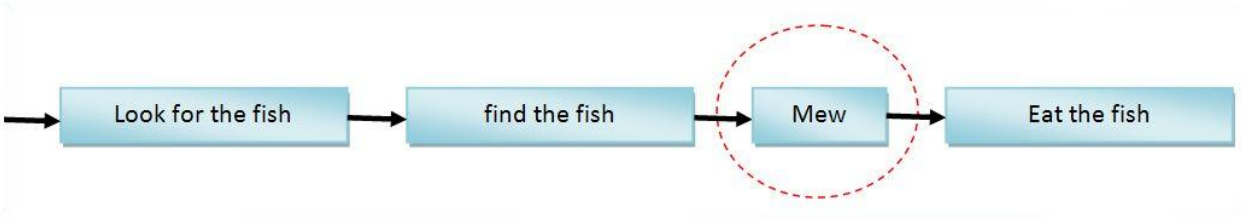

Figure 10: Another Way to Decompose "How a cat eats a fish ?"

4.5.4 When

The way to decide transition conditions between behaviours is a process of rules definition. Game designers have to point out *when* one behaviour has to be executed or transmit to the next behaviour. It is a simple decision-making and reaction planning process. At any given time, we have to explain under what circumstances the game agent need to do another action.

Since our consideration is to reduce the complexity in the design level and provide a powerful real-time planning and learning agents' learning and planning architecture, we use *rules* to control the decision making process between behaviours. Each rule contains an *antecedent* and a *consequent*.

IF *antecedent* THEN *consequent*

For example, there is an arrow between two modules in Figure 9. The arrow denotes that there is a transition between the two behaviours. The responsibility of a rule is *behaviour*

arbitration. Game designers can specify that the rule for behaviour transition between behaviours. For example, we have three rules (R1, R2, and R3) to arbitrate three behaviours in the Figure 9.

- R1: IF *the cat feels hungry* THEN *the cat looks for the fish*.
- R2: IF *the cat smells the fish* THEN *the cat goes to the fish position*.
- R3: IF *the cat arrives the fish position* THEN *the cat eats the fish*.

4.6 Modularity of the Behaviour Design

Modularity is one of the most important aspects of procedural behaviour design. It simplifies the off-line game design process, and provide explicit information for game agents' behaviour control in real-time.

- Behaviours in different modules can be developed and maintained as a way of separating concerns.

  We could have two different GOBTs (see Section 5.6) "Eat" and "Fight" for an agent, which describe detail procedures (subgoals) for achieving goals eating and fighting.

- In addition, knowledge in modules demonstrates maintainability by enforcing logical boundaries between components in the same modules.

  If an agent has a series of goals, these goals can be approximately classified to several categories based on their trigger properties. For example, in Section 7.6, the agent Xman has Goals "Idle","Eat" and "Rest" grouped together and goals "Evade", "Defense" and "Fight" grouped together. The trigger property we used for this discrimination is whether or not the monster is visible. These two groups of goals are stored in two priority queues, and ordered based on priority factors of itself. Also, the details of each goal can be established and maintained separately.

- This explicitly modular structure gives the interpreter in the planning level and the coordinator in the learning level a clear idea of how to interact with different modules, and what it can get from modules in real-time data processing.

  For example, given a set of *Strategies* for a team of game agents to defeat the adversarial team, the coordinator can evaluate the current situation of the team, and decide which *Strategies* has the best match for the team to defeat the opponent.

- The modular knowledges can be combined and decomposed either in off-line or in real-time.

  As different features of game agents, a set of basic behaviours knowledge modules can be selected and combined in order to represent different game agents. Especially, in real-time, modulars can be retrieved from existing modules in order to create new knowledge. For example, the *Experience* record that we discussed in the learning layer is created in real-time by combining several knowledge modules. The bidirectional arcs in Figure 37 on page 103 between *Experience* and other knowledge modules indicate the composable aspect of modular knowledge.

- Modules in our system have a pure logical knowledge format.

  The knowledge representation in our system is kept as its own format without any knowledge of the 3D graphics engine to which it is connected. This is a consideration for routine system development and future system extension and maintenance. So the game can be debugged and tested within the AI module initially; if the tests pass, 3D graphics engines can be connected for full game testing. For example, in Figure 6 on page 35, we present the Adapter pattern used in our program in order to have communication between pure AI logic modules and the 3D graphics engine.

Chapter 5

Goal Oriented Behaviour Design

The procedural behaviour design is the starting point for describing game agents. We have to provide a clear indication of the guiding principle for designing game agents. In this chapter, the goal oriented design approach is incorporated into our design methodology. In addition, we provide a new data structure called *Goal Oriented Behaviour Tree* to realize our idea about goal oriented design for game agents.

5.1 Goal and Behaviour

First, we have to provide a clear idea about the relationship between the terms *goal* and *behaviour*. Basically, game designers design possible goals and behaviours for game agents. And, all in-game goals and behaviours are based on game agents' pre-designed knowledge. In real-time, goals that a game agent possesses are affected by its knowledge and environment. Game agents' pursuit goals are embodied in their behaviours which are responses to the environment in real-time. When a discrepancy is revealed in a game agent's current goal and the state of the environment, the game agent's current goal changes. Or, we can say the reason of changing goal is the conflict between the game agent's current goal and the discordant voice from their environment. Along with the changes of goals, game agents' next behaviour would be changed. On the other hand, after perceiving the environment,

if there is no obvious reason to change their current goal, the game agent's reaction to the environment is the next behaviour toward its current goal.

5.2 Goal Oriented Design and Procedural Behaviour Design

Game design is a goal-oriented design since playing a game requires achieving goals. The intuitive way to design a game is to set goals for players to achieve. The goal oriented design can be the starting point of the entire game design. We designate *goal oriented* as the key point in modeling game agents' behaviour.

We introduce Goal Oriented Behaviour Design (GOBD) as a methodology for constructing intelligent game agents. Game development is similar to component based development which combines large-grained components, running in their own threads of control, into large-scale applications. A goal in a game can be divided into a serious of sub-goals. And, the process of designing a game consists of a serious of subsystem design, and each subsystem has its own goal.

In engineering, an autonomous device can be goal-oriented; in artificial intelligence, agent rational behaviour is also goal-oriented. In order to pursuit a goal, game agents have to implement a series of actions toward their purpose. These actions can be arranged in procedures as game agents' behaviours.

The procedural knowledge provides the essential model for creating behaviour for game agents. However, it is not adequate to depict intelligent game agents and their environment. For example, there are different ways to achieve the same goal. We cannot simply list all these procedures because this would lead to real-time reasoning inefficiency for game agents. A suitable data structure is needed to arrange goals. So, we provide a symbolic, behaviour-based, goal oriented and autonomous model, GOBT for game agents' behaviour design.

GOBT is a tree data structure created in off-line game design. The GOBT can be decomposed or combined as requirements of game agent's prototype. The root of a GOBT

is a particular goal for a game agent. A whole GOBT describes how to perform sets of behaviours or subgoals in order to fulfill a certain goal. The unique data structure—GOBT is discussed in Section 5.6. Parts of this chapter were originally published as [SG08a].

5.3 General features of GOBD

Goal Oriented Behaviour Design is an incremental process. GOBD allows combining top-down and bottom-up aspects (top-down thinking and bottom-up acting [SP06]) to model game agents' behaviour. The key principles of GOBD are listed below.

- Goal identification

  Goal identification is a process of looking for necessary goals for the game agent. It is a top-down approach which starts by considering top-level goals. After determining top goals, game designers have to decompose top goals to a sequence of sub-goals. There may of course be more than one possible way of decomposing a goal. Identifying essential goal decompositions is also very important. Game designers have to avoid redundancy in goal identification.

- Top-down and bottom-up convergence

  The GOBD obtains benefit from the combination of top-down and bottom-up principles. Goal identification is a top-down analysis process which designate the series of subgoals for top goals. The bottom-up approach provides basic components of these goals, constructing paths to achieve these goals.

- Modularity and hierarchy

  GOBD inherits modularity approach from the procedural behaviour design. The modularity approach subdivides a goal into small components, and makes overlap among components as small as possible. In addition, modularity offers benefits such as augmentation and exclusion. Game designers can add new modules to an existing module

in order to extend additional behaviours to the game agents. In contrast, deleting certain behaviours can simplify the behaviour procedure to achieve a goal.

In addition, the hierarchical approach is introduced into the GOBD. The hierarchical approach distributes goals in a tree hierarchy. Higher level goals are more abstract than lower level goals. Also, higher level goals are the consequence of achievement of lower level goals. The hierarchical approach provides straightforward clues for game agent's behaviour control.

Game designers can combine behaviour design as modular and hierarchical approaches. Modularity and hierarchy are two main design methodologies for game agents behaviours. They not only affect the design complexity of game agents, but also affect the real-time planning and learning algorithms. Both game agents' planning and learning are based on off-line design knowledge. These knowledge modules are retrieved by planning and learning mechanisms in real-time. The behaviour design systematically places knowledge in modules and hierarchical structures, which leads to explicit searching mechanisms for planning and learning in real-time.

5.4 Atomic Components for GOBD

To design a complex structure, one powerful technique is to discover viable ways of decomposing it into semi-independent components corresponding to its many functional parts [FP99]. The design of each component can then be carried out with some degree of independence of the design of others, since each will affect the others largely through its function and independently of the details of the mechanisms that accomplish the function [FP99]. In this section, we introduce the basic component design for GOBD. It is the bottom-up approach which construct components for intelligent agents.

The GOBD is influenced by Object-Oriented Design (OOD) techniques for objects and their behaviour. Our design ideas are:

- All entities in a game are objects.

- Each object can have different actions.
- An action a can have several attributes with values.
- A state S is composed of object(s) in certain action(s) .
- A rule R denote the transition from one game state (antecedent) to another game state (consequent).
- A goal G is triggered by a game rule.
- A GOBT consists of goals and states as its tree nodes.

However, GOBD applies techniques for building plans and decomposing behaviours that are analogous but not identical to the OOD methodology for building inheritance hierarchy. In GOBD, the behaviour hierarchies in GOBT denotes paths to pursuit goals. There is no correspondence between the OOD notion of inheritance and the GOBT hierarchies.

We have to provide functionality to allow game designers to create various entities for the game during the design process. We have to provide the functionality of dynamically creating various classes in the underlying C++ source code. In our system, we defined seven basic classes as below. These classes represent basic components for the GOBD. In detail, combination of these classes in C++ code describe GOBTs for game agents.

- Class CGameObject

  A game object can be any entity in a game. It can be a player, in game character or simply a tree in the game.

  We simulate the way of dynamically creating objects by using C++. The GameObject class can be used to represent various entities in the game. Each type of entity is a class; also it is a subclass of GameObject class. The GameObject class is a super class for all classes defined in the system.

- Class CGameAttribute

Objects of Class CGameAttribute represent simple attributes of a game object; such as color, brightness, position (x, y, z) etc.

- Class CGameAction

  Objects of Class CGameAction represent actions of a game object, such as "Hidden", "Visible", "Run", and "Walk". A game action consists of a number of game attributes. For example, when an Game Object is Running, it has its own specific lighting, position and other attributes.

  $$Action = Attribute_1 + \cdots + Attribute_N \qquad (N \geq 1)$$

- Class CGameActionList

  An object of Class CGameActionList represents a group of actions.

  $$GameActionList = GameAction_1 + GameAction_2 + \cdots + GameAction_N \qquad (N \geq 1)$$

- Class CGameObjectList

  An object of Class CGameObjectList represents a group of objects.

  $$GameObjectList = GameObject_1 + GameObject_2 + \cdots + GameObject_N \qquad (N \geq 1)$$

- Class CGameState

  An object of Class CGameState represents state(s) of an object or a group of objects. For example, we can represent the fighting (GameAction) status of a monster (GameObject) as an object of the Class CGameState.

  $$GameState = \begin{cases} GameObject + GameAction \\ GameState = GameObject + GameActionList \\ GameState = GameObjectList + GameActionList \end{cases}$$

- Class CGameRule

  Game rules consist of an *antecedent* and *consequent* in the form:

  IF *antecedent* THEN *consequent*

Objects of CGameState class can be used to represent as antecedent or consequent in game rules. The antecedent or consequent is a composition of game states by using logic set operators *AND*, *OR*, *NOT*.

- Class CGameGoal

  A game goal can be a node in the GOBT. Each game goal is associated by a game rule.

- Class CGameGOBT

  The object of Class CGameGOBT represents a GOBT which contains game goals.

- Class PriorityQ

  In order to cooperate with game agents' modeling and autonomous reasoning, we design the Class PriorityQ. Objects of this class represent the *Intentions* of game agents (Section 6.3.4) .

5.5 Hierarchy of the Goal Oriented Behaviour Design

Hierarchical structure is a traditional software engineering approach which makes software design systematical and coherent. Many large systems can be decomposed as hierarchical structures, including software systems. This is why trees and hierarchies appear so often in software engineering. Using a hierarchical approach in the behaviour design can reduce complexity of procedures. The data structure GOBT is a tree hierarchy which represents game agents behaviours in the way of achieving goals.

The hierarchy structure helps game designers organize game agents' behaviours in an intuitive way. The decomposition of a goal into sequences of subgoals makes both game design and programming easy and efficient. Conversely, a hierarchy can be used to collect information for planning and learning. The hierarchy structure makes paths of access a goal node in the tree explicit.

5.6 The Goal Oriented Behaviour Tree

We provide a symbolic, behaviour-based, goal oriented, and autonomous model, Goal Oriented Behaviour Tree (GOBT), in Gameme. GOBTs are a graphical representations used to denote complex game systems. The GOBT is a data structure created in off-line game design. However, GOBT is not simply a data structure such as FSM or RBS. It is customized for goal oriented behaviour design in game development.

5.6.1 Why GOBTs?

In order to fit the goal oriented nature of game design, we design the data structure GOBT to describe game agents. The fundamental idea behind GOBT is that intelligent behaviour can be created through a collection of simple behaviour modules; a complex goal can be accomplished by a collection of simpler sub-goals. The GOBT captures fragments of behaviours and puts them in tree hierarchies, and describes how to perform a set of behaviours in order to fulfill certain goals.

The GOBT is intuitively reactive in nature, meaning that, ultimately, the tree architecture can simply map inputs to goals without planning. The basic premise of reactivity is that we begin with a simple behaviour or goal at lower levels of GOBTs, and once we have succeeded there, we extend with higher-level goals.

Translation of discrete behaviours into a GOBT and the subsequent integration of behaviours into the hierarchical structures in GOBT help us uncover problems with original textual game design ideas. The GOBD gives game designers guidelines to follow for editing all transitions between behaviours.

In addition, the GOBT is no less efficient than a well-written script. The clear AI logical relationships presented in GOBTs is easier to introspect than scripting.

5.6.2 Formalization

A Goal Oriented Behaviour Tree (GOBT) has three components: a condition C, and goal, G, and an action, A. A GOBT is written as a triple, (C, G, A). The condition and action of a GOBT may both be GOBTs: thus GOBTs have a recursive structure. Here is the formal definition:

A GOBT consists of either:

- a single node, which may be:
  - a simple condition, C, or
  - a conjunction, $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, or
  - a disjunction, $C_1 \vee C_2 \vee \cdots \vee C_n$, or
  - an action, A; or
- a tree, or triple, (C, G, A).

Furthermore, a node may be *continuous*, in which case it is marked with a star. For example, $B^\star$ indicates that B is evaluated continuously (see Section 5.6.3).

The basic logical relationship of a goal G and its condition C and action A is shown in Figure 11. The diagram expresses the rules:

IF *condition* is *TRUE* THEN *goal* is *TRUE*;
IF *goal* is *TRUE* THEN *action* is performed.

A GOBT may also be represented by a tree diagram. For this purpose:

- The triple (C, G, A) is drawn as a tree with root G (Goal), left subtree C (Condition), and right subtree A(Action): see Figure 11.
- In general, each node is drawn as an oval, possibly with a name inside it.
  In particular, a continuous node is drawn as a double-oval, possibly with a name inside it.

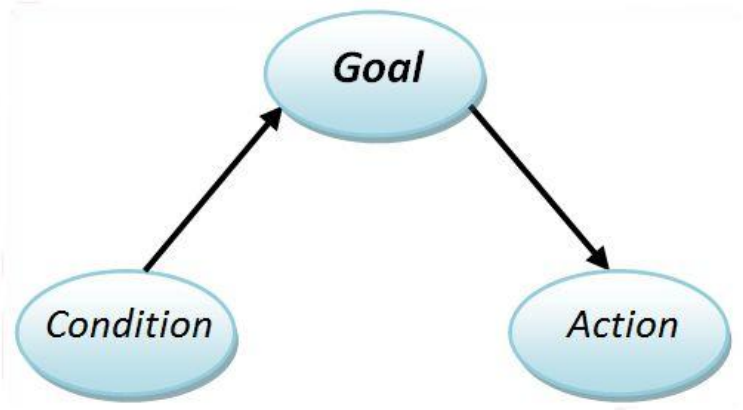

Figure 11: The Basic Logical Relationship between Nodes in GOBTs

- A directed edge (upwards) refers to the link from a condition C of the node to the node itself; a directed edge (downwards) refers to the link from a node to its action A.

Figure 12 shows an example of a GOBT. The node *B1* is the root of the GOBT. If we assume that the node *B1* is the current goal node, then nodes *B2* and *B3* are conditions of *B1*, and *B4* is the action of *B1*. Furthermore, *B2* is the goal node of the subtree in the dashed circle; or we can say *B2* is the root of the subtree. The corresponding expression, according to the formalism, is:

$$((B5, B2, (B9, B6, B10)) \vee B3, B1, (B7^{\star}, B4, (B11 \wedge B12, B8, B13))).$$

A GOBT may be *executed*. Informally, execution consists of the following steps:

1. Evaluate the condition and store the Boolean result in the goal.
2. If the goal is *TRUE*, then perform the action.
3. Return the result.

5.6.3 Properties

A GOBT is a data structure which has the following special properties.

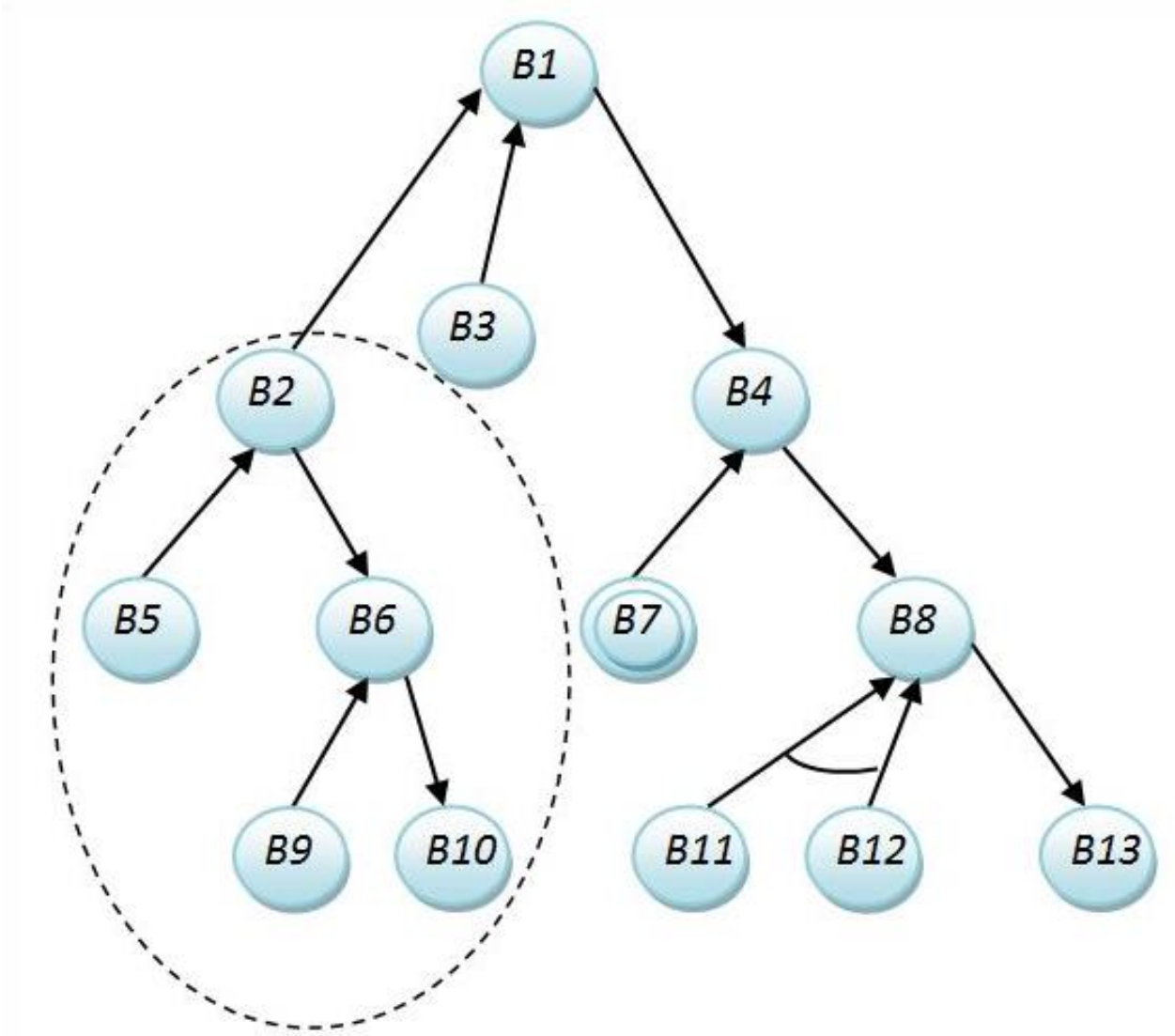

Figure 12: The Structure of a GOBT

- The GOBT is a directed tree in which each node has exactly one parent. A GOBT consists of a root and a set of zero, one, or more subtrees.
- There is exactly one root node in a GOBT. The root node of a GOBT should be a goal G, and is the node without parents.
- A node except the root node can either be a goal G or a condition C or action A of its parent.
- A leaf node in GOBTs has no successor.
- A *continuous* node has a corresponding Termination Condition (TC). Each continuous node is continuously executed by single steps; in each step, the TC is continuously evaluated. The continuous node is executed continuously only so long as its TC remains *FALSE*. If the TC becomes *TRUE*, the continuous node is terminated, and the follow-up node will be *TRUE* and executed. For example, *B7* in Figure 12 is a continuous node.
- An AND nodes have the same parent, have edges direct to their parent, and are connected by arcs. The logical relationship between *B11* and *B12* in Figure 12 is conjunction.
- A OR node is a node without an arc connected to other nodes with the same parent; if we consider the nodes indicated as *B2* and *B3* in Figure 12, their logical relationship is disjunction.

As well as specifying the tree-structure properties of GOBTs, we also have to consider the logical rationality when constructing GOBTs.

- GOBT G_1 is equal to GOBT G_2 ($G_1 = G_2$) if and only if they have exactly same nodes and structure.
- A Node $N1$ is equal to another node $N2$ ($N1 = N2$) if and only if they both have same name and value. The node could be a goal node, a condition node or an action node.

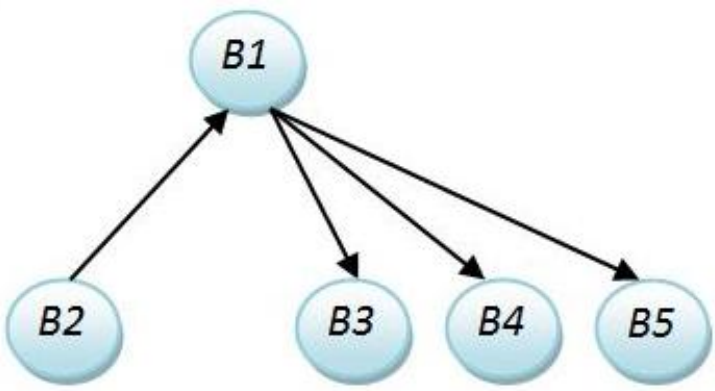

Figure 13: The GOBT with OR Action Nodes

- Usually, leaf nodes are states S (see Section 5.4). Using goal nodes as leaves is not recommended in GOBTs. If so, the goal node is set to *TRUE*, and its condition and action are set to NULL.

- Several condition nodes with the same parent can either be AND or OR nodes.

- Several action nodes with the same parent node can either be AND or OR nodes. Especially, disjunction nodes have their special execution order.

  For each goal G, game designers have to provide precise description to indicate its actions A. Conjunction actions means these actions have to be executed synchronously. For example, after fulfilling a goal, a game agent can dance and sing at the same time. In the GOBT, nodes with value *dance* and *sing* are two conjunction action nodes.

  On the other hand, if we do not specify execution order of disjunction action nodes, it could lead to a confusing situation since there is no clear indication for which action should be executed first. So, we define if there are several action nodes with the same parent node are disjunction nodes, the execution order is from the most left node to the most right node. For example, in Figure 13, action nodes are executed as the order $B3 \rightarrow B4 \rightarrow B5$.

- A goal node could have no condition, action or both. In these situations, the link connected the goal node and its condition or action node should be set as NULL.

5.6.4 Execution

We now consider the process of executing GOBTs more carefully. The execution of GOBTs shows how a GOBT work in order to fullfill a goal (root). In off-line game design, execution of GOBTs can be used to validate the designation of GOBTs. Game designers can test different execution paths toward the root in GOBTs. In addition, executing GOBTs can create goal execution orders of GOBTs off-line (see Section 6.3.3). Furthermore, goal execution orders are stored as plans for the usage of real-time game agents' planning and learning.

Informally, execution is performed as follows. Given a GOBT node N:

- if N is a simple condition, evaluate it and return the result;
- if N is a simple action, perform it and return $TRUE$;
- otherwise, the node is a triple $N = (C, G, A)$. In this case:
  - execute C recursively and assign its value to G;
  - if G is $TRUE$, then execute A;
  - return G.

Formally, the procedure for execution is named *exec*, and the result of executing a node N is the Boolean value of *exec*(N). Figure 14 shows its definition. Note that *exec* is called recursively on both subtrees, ensuring that all nodes are processed. Figure 15 shows the execution of the GOBT in Figure 12 by this procedure, assuming that all conditions yield $TRUE$(all *eval*(N) functions return $TRUE$). Recursive calls of *exec* are shown indented. In addition, if there is goal nodes without condition or action, functions *eval*(N) or *perform*(N) returns $TRUE$ in order to let the execution procedure proceed.

Here are some of the other possibilities for executing the GOBT of Figure 12:

- If $B5$ is $FALSE$, then $B2$ is set to $FALSE$, the tree rooted at $B6$ is not executed, and action $B10$ is not performed. However, $B3$ is $TRUE$ and therefore $B2 \vee B3$ is $TRUE$, so everything else remains unchanged.

```
bool exec(N):
    if N is a condition then
        return eval(N)
    else if N is an action then
        perform(N)
        return TRUE
    else
        let N = (C, G, A)
        G := exec(C)
        if G then
            exec(A)
        return G
```

Figure 14: Executing a GOBT

```
exec(B1)
    exec(B2)
        exec(B5)
        B2 := exec(B5)
        exec(B6)
            exec(B9)
            B6 := exec(B9)
            exec(B10)
    exec(B3)
    B1 := exec(B2) ∨ exec(B3)
    exec(B4)
        exec(B7*)
        B4 := exec(B7*)
        exec(B8)
            exec(B11)
            exec(B12)
        B8 := exec(B11) ∧ exec(B12)
        exec(B13)
return B1
```

Figure 15: Executing the GOBT of Figure 12

- Assuming $B1$ is *TRUE* and $B7$ is *FALSE*, then $B4$ is set to *FALSE*, and the tree rooted at $B8$ is not executed. No actions in the right subtree of $B1$ are performed.
- If either $B11$ or $B12$ is *FALSE*, action $B13$ is not performed.

We could also use the execution of GOBTs to generate execution orders (Section 6.3.3) for GOBTs. For each GOBT, there could be more than one execution order. The number of execution orders in a GOBT depends on the number of disjunction condition nodes. And, it is notable that a set of conjunction nodes can only generate one execution order. For example, there are two execution orders in Figure 16 for Figure 12.

We have to do some modifications in the *exec(N)* function in order to output execution orders for GOBTs.

- We do not have to evaluate conditions (function *exec(N)*) as well as perform actions (function *perform(N)*) since what we need is to output the name of each node in execution orders.
- In practice, we have to assume all conditions are *TRUE* in order to let the *exec(N)* function traverse the whole GOBT and generate execution orders.
- In addition, during the execution of the *exec(N)* function, each disjunction condition node leads to a new execution order. For example, in Figure 12, nodes $B2$ and $B3$ are list in two execution orders since they are OR nodes of the same parent. But, nodes $B11$ and $B12$ are bonded together in one execution order since they are AND nodes of the same parent.

- $(B5 \rightarrow B2 \rightarrow (B9 \rightarrow B6 \rightarrow B10)) \rightarrow B1 \rightarrow (B7^{\star} \rightarrow B4 \rightarrow ((B11 \wedge B12) \rightarrow B8 \rightarrow B13))$
- $B3 \rightarrow B1 \rightarrow (B7^{\star} \rightarrow B4 \rightarrow ((B11 \wedge B12) \rightarrow B8 \rightarrow B13))$

Figure 16: Executing orders of the GOBT of Figure 12

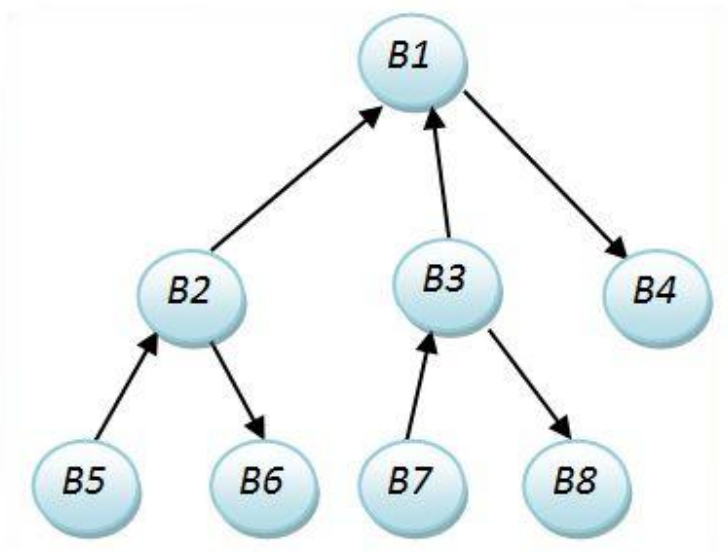

Figure 17: Another GOBT Example

There is no lazy evaluation in the execution of GOBTs. Most languages use the following equivalences ("lazy evaluation"):

$$P \wedge Q \equiv \textbf{if } P \textbf{ then } Q \textbf{ else false}$$
$$P \vee Q \equiv \textbf{if } P \textbf{ then true else } Q$$

Thus Q is not evaluated if P is **false** in $P \wedge Q$ or if P is **true** in $P \vee Q$.

However, this may cause problems in the execution of GOBTs. Consider the GOBT in Figure 17, the interesting feature of this GOBT is that the conditions $B2$ and $B3$ both have actions, $B6$ and $B8$. The designer might assume that both of these actions will be performed before $B4$.

Suppose $B5$ and $B7$ are *TRUE*, action $B6$ is performed, and $exec(B2)$ returns *TRUE*. There is no need to execute B_3 because we know that $B2 \vee B3$ is *TRUE* anyway. But then action $B8$ will not be executed, even though $B7$ is *TRUE*. This the designer's assumption is incorrect if lazy evaluation is used.

A similar example could be constructed with a conjunction, $B2 \wedge B3$. In this case, the question is whether $B3$ is executed when $B2$ is *FALSE* and we therefore know that $B2 \wedge B3$ must be *FALSE*. In both of these examples, the key point is that $B7 = TRUE$ does not

ensure that $B8$ is executed.

Furthermore, lazy evaluation might lead to incomplete execution orders generated for a GOBT since it could not guarantee traverse each node in a GOBT when all conditions are *TRUE*.

5.6.5 Editing

During off-line game design, operations of adding or deleting goals G or states S from GOBTs happen all the time. In this section, we introduce basic editing operations for GOBTs. These operations are embodiments of GOBD's properties, such as exclusion and augmentation. Game designers could add, delete, combine or decompose nodes of GOBTs as the development of game scenario.

5.6.5.1 Add or Delete Nodes

Game designers can add or delete nodes from GOBTs. Since GOBTs have distinct logic expression and execution procedure, game designers cannot simply add or delete nodes from GOBT; at the same time, logic relationship among relative nodes (parent, sibling and descendants nodes) should be considered in order to prevent confusion of the logic.

Here, we list different situations in deleting a node from a GOBT.

- Deleting a goal node means deleting a goal for a game agent.
  - Deleting a root node means deleting a whole GOBT from the behaviour design of a game agent.
  - Deleting a goal node in a GOBT means cutting the whole subtree which has the goal node as its root from a GOBT. For example, in Figure 12, if we delete the node $B2$, it leads to delete the whole subtree which has $B2$ as the root.
- Deleting a condition or action node.

- If it is the only condition or action node of a goal node, we have to set the link from goal node to this node as NULL.
- If it is a conjunction condition node or action node, the logic relationship should be rearranged for its sibling nodes. For example, in Figure 12, if we delete the node $B11$, the node $B12$ becomes the only condition node for the node $B8$. The conjunction relationship no longer exists.
- If it is a disjunction condition node, one possible execution path to the goal no longer exists. For example, if we delete the node $B3$ in Figure 12, the execution order included $B3$ in Figure 16 should be deleted also.
- If it is a non-leaf condition or action node, we also have to cut all its descendants nodes.

As in deleting a node in a GOBT, adding a node to a GOBT should consider the logic effect on the structure of the GOBT. Here, we list situations of adding a node to a GOBT.

- It is not allowed to add a second root to a GOBT.
- If we add a goal node as a leaf node in a GOBT, links to condition and action of this node should be set as NULL.
- Adding a goal node to be a non-leaf node in a GOBT, we have to add this node's condition and action nodes as well. Actually, it is the same as adding a new subtree to the GOBT (see Section 5.7.2).
- Adding a condition or action node to a GOBT. We have to reconsider the logic relationship with its sibling nodes. Game designers have to specify this node as either conjunction or disjunction with its siblings.

5.6.5.2 Composition and Decomposition

In the previous section, we introduce situations of adding or deleting a node from a GOBT. In this section, cases of composition and decomposition of GOBTs are presented. As usual,

game designers should still consider logic relationships among GOBTs which are composed or decomposed.

Here, we present two most used composition cases for GOBTs.

- Matching Composition

  The matching composition combines two GOBTs by adding a GOBT to be a subtree of another GOBT based on a matching point. The matching point refers to the position of a node in one of the GOBTs. This operation requires deleting a node in the matching point of a GOBT and adding another GOBT at the matching point. In Figure 18, we delete the node $S2$ in the GOBT $G1$, and add the GOBT $G2$ as a subtree to GOBT $G1$ at the position of the matching point, giving the new GOBT $G3$.

  The deleted node $S1$ and the subtree root $G2$ could be something in common. In practice, game designers might need to extend a GOBT by changing a condition or action node to a goal node. The matching composition could be the choice. For example, from Figure 26 to Figure 27, game designers delete the node $S5$ in Figure 26 and add the new GOBT with goal node $G4$ as the root. Nodes $S5$ and $G4$ has the same value, but different node type. By using the matching composition in GOBTS, game designers extend the GOBT For the purpose of describing more complicate behaviour of game agents.

- In-order Composition

  In-order composition occurs when several GOBTs are used to describe several different goals. One goal has to be processed after the accomplishment of another goal. The solution is to create a new GOBT, and a new node will be created as the root of the new GOBT. The root of the new GOBT represents a new goal.

  > **In-order Composition:** if we have GOBTs G_1 and G_2, then we can construct a new GOBT (G_1, G, G_2).

  In Figure 19, the GOBT *G2* has to trigger after *G1* (GOBT with root *G1*) is executed. Composition consists of creating a new GOBT with new root node G. The GOBT *G1*

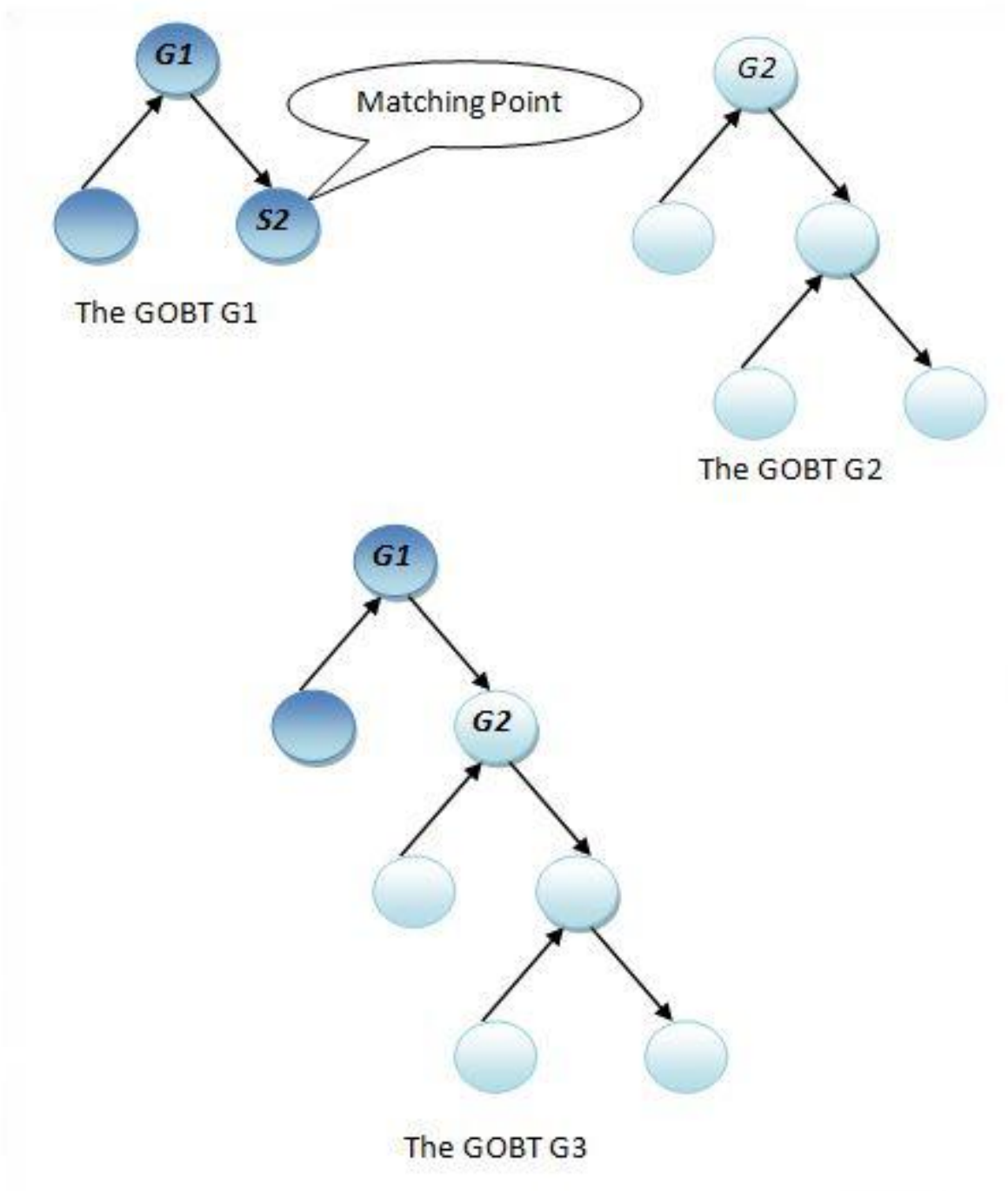

Figure 18: The Matching Composition

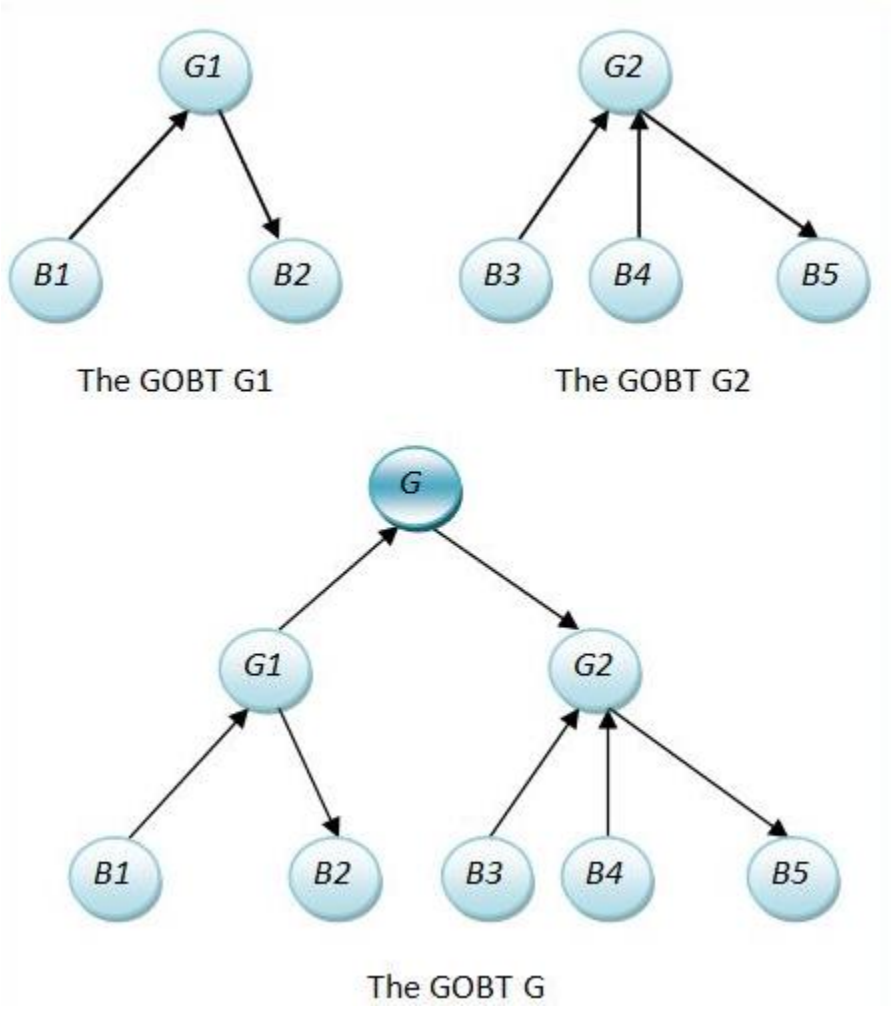

Figure 19: The In-order Composition

becomes the left subtree of the root G and the GOBT $G2$ becomes the right subtree of the root G.

5.6.6 Traversal

Traversals of GOBTs offer search guideline for game agents behaviour control. The traversal procedure in a GOBT corresponds to the *exec* function in Section 5.6.4 which visits each node in a GOBT. In this section, we give a detail description for two basic operations in GOBT traversals. Normally, these two traversals operations work together to get the autonomous search to work.

- Level Visiting

  This operation visits conditions of a goal node successively. Since AND nodes and OR nodes represent different logic relationship for nodes which have the same parent node, we discuss two situations in the level visiting.

  Since nodes $B2$, $B3$ and $B4$ are AND nodes, the level visiting orderis : $B1 \rightarrow B2 \rightarrow B3 \rightarrow B4$ in Figure 20. If there is a condition that is *FALSE*, the traversal terminates and returns a *FALSE* value to the goal node $B1$. If all conditions are *TRUE*, the goal node is set to be *TRUE*; and the action node Bm is performed.

  IF (*B2*==*TRUE* AND *B3*==*TRUE* AND *B4*==*TRUE*)
  THEN (*B1* is *TRUE*)
  AND (perform Bm)

  Nodes $B5$ to Bn are OR nodes, so each OR node corresponds to a execution order. If one of the OR node is *TRUE*, the goal node $B1$ is set *TRUE* and action node Bm is performed. For example, the node $B5$ is a OR node, so there is an execution order for the GOBT in Figure 20:

  IF (*B5*==*TRUE*) **THEN** (perform Bm)

- Depth-order Visiting

  In the depth-order visiting, we always attempt to verify that the current goal node is *TRUE*. So we have to start accessing the condition node of the current, and recursively verify all conditions/sub-conditions nodes of the goal node are *TRUE*. If there is a node is *FALSE*, the traversal is terminated and returns a *FALSE* value to the action node. Figure 21 indicates this visiting process. In order to verify $B1$ is *TRUE*, we have to start with $B2$; for $B2$, we have to check $B5$ and so on checking $B9$. If $B9$ is *TRUE*, then $B5$ is set to be *TRUE* and $B10$ is executed, then $B2$ is set to be *TRUE*, and finally, $B1$ is set to be *TRUE*.

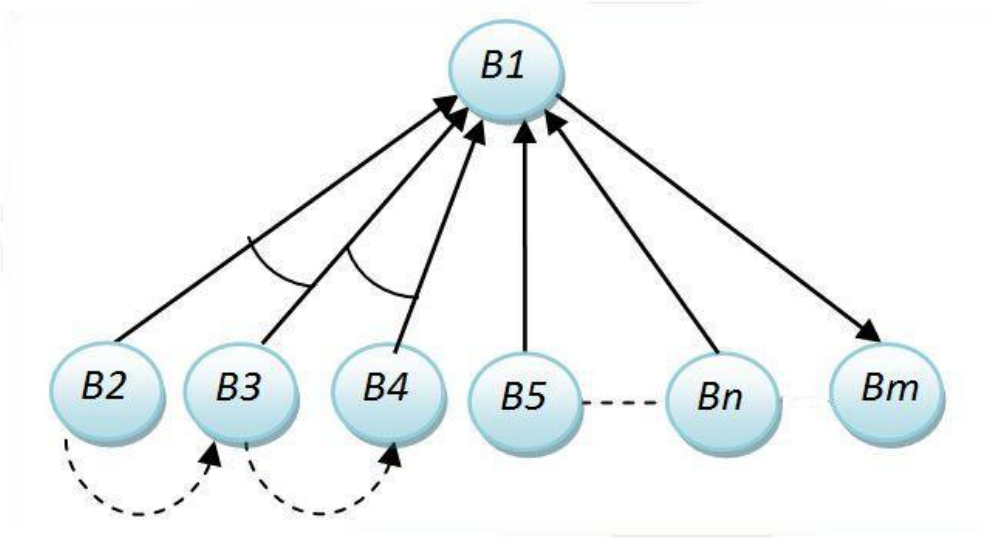

Figure 20: The Level Visiting

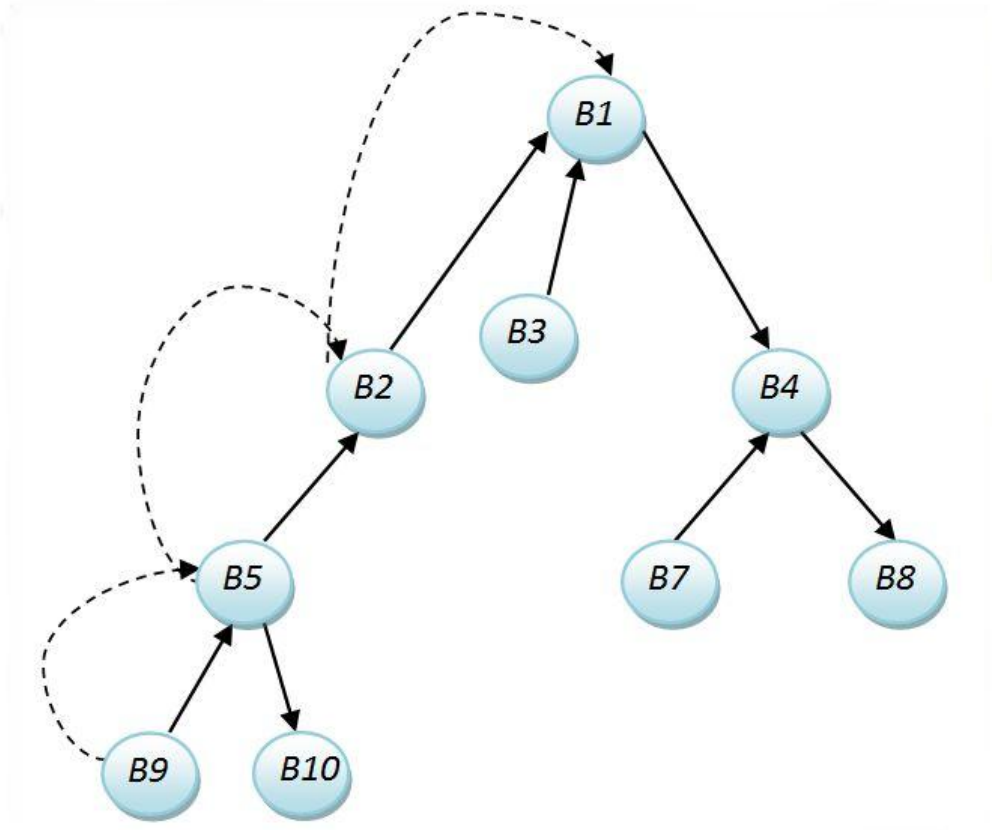

Figure 21: The Depth Visiting

5.6.7 AI Nature of the GOBT

In this section we discuss the key reasons about how game AI developers can benefit from the form of hierarchical logic of a GOBT.

The GOBT is an agent control structure that directs the agent towards a goal in a manner that takes into account changing environment circumstances. Every node except the root is the regression of its parent node through the action node linking to it. It matches the goal oriented design for game scenarios in working backward from a goal condition; a goal's condition is its left children. It is executed by searching for the shallowest True node and executing the result nodes linking to nodes; a goal's result is its right children.

In addition, the GOBT is a simple automatic system which has the ability of regressing conditions through continuous nodes. Initially, the termination condition (TC) is *FALSE* for a continuous node. The GOBT keeps evaluating the TC in each step. If the TC keeps being *FALSE*, no further execution for the follow-up node. The game agents still executes the action denoted by the continuous node. Once the TC becomes *TRUE*, the game agent turns to execute the follow-up node of the continuous node.

The GOBT provides a solution for state-space search which offers the basic search mechanism for game agents' planning and learning. Each node in the GOBT represents a goal or state. The GOBT decomposes the root goal state into sequences of subgoals. Execution orders for each GOBT indicates the paths of traversing the tree in order to accomplish the root goal. For condition nodes, the logical relationship with its sibling, parent and result nodes shows what would be taken as the next action in the execution order. The depth and level traverses provide different possible operations for search in GOBTs.

In addition, the AND-OR nodes used in the GOBT extends the goal state-space search to achieve more problem reduction. Successors of AND nodes represent goals to be jointly achieved, and successors of OR nodes represent different ways of fulfilling a goal. The AND-OR nodes allow us to represent both cases in which a whole set of subgoals should be achieved, and in which any of subgoals could achieve a goal.

Furthermore, traversals of GOBT directs real-time planning and learning mechanisms to

find right paths in achieving goals. The level traversal could guarantee all AND nodes are *TRUE* before the upper level goal node becomes *TRUE*. The depth traversal could ensure goal regression into the root goal node.

The GOBT is a hierarchical logic model which is customized for game development. We can extend the usage of GOBTs to control complex agents. It is arguable that the very same behaviours could be built with a FSM as the GOBT example. Indeed, FSMs have become extremely popular over the last decade in game industry, and have been used to build some successful games. However, FSMs still have problems, and game developers are seeking more reliable logic models.

- GOBTs can have multiple conditions and action nodes. With OR and AND nodes, GOBTs can describe complex logical relationship in more simpler hierarchical structure than FSMs.

- GOBTs are deliberately based on the natural aspect of the tree traversal. They are capable of searching ahead in real-time, and provide different solutions for agents' behavioural reasoning. On the other hand, an FSM is a linear automation, and cannot provide long-term goal planning.

- GOBTs are suitable for design logic in layers and modules. It is similar to providing options for game designers to design logic in different levels of detail. It can avoid agents' behavioural planning and decision being affected by minor animation details in games.

- GOBTs provide flexibility in scale. The composition and decomposition of GOBTs is much easier than that of FSMs. FSMs, even hierarchical ones, are not suitable for many levels of logic.

5.7 Example of generating GOBTs

5.7.1 The Generation of GOBT "Eat"

This section describes the creation of the GOBT "Eat" by following three steps in Figure 23. This GOBT describes the procedure of eating food for a game agent Xman. First, the Xman has to look for food. And, after finding the food, he goes to the food position and eats it.

1. The first step is to decide main goal and subgoals.

2. The second step is to expand the goal sequence by adding rules and states to generate the GOBT.

3. The third step is to generate execution order for a GOBT.

Besides normal nodes, we also define continuous nodes in this GOBT. For instance, the *S3* node in GOBT "Eat". It keeps looking for food. The TC for this node is "find food". If so, the node *G2*, which is the follow-up node in the execution order, becomes *TRUE*. Figure 22 on page 77 shows the resulting GOBT.

5.7.2 Extensions of GOBT "Eat"

We prefer modular design for game agents' behaviour. GOBTs suits for modular design and maintenance. Game designers can create different GOBTs individually, and combine GOBTs when it is necessary. In addition, nodes in a GOBT can be replaced by other GOBTs as the requirements of behaviour extensions.

For example, after game designers create the GOBT "Eat" as Figure 22, they may think about adding two more conditions before Xman is ready to eat food. Xman has to check the enemy status; indeed, he has to feel hungry. So, game designers remodel the original GOBT as in Figure 24 on page 78. There is a new node (the node "G") is created to be the root of the GOBT "Eat"#1. And, the original GOBT "Eat" could be added into the GOBT "Eat"#1 as a module. Furthermore, if game designers want to change the state "S4" to a

goal "G4" in order to have more actions on Xman. The GOBT "Eat"#1 could be changed as GOBT "Eat"#2 in Figure 27 on page 80. If the Xman checks there is no Monster around, he should be in Idle status.

5.8 Conclusion

Taking inspiration from robotics, we present an approach of GOBD which can be used in agents' behavioural design. GOBTs and related goal processing architecture allow game agents handle goal-directed behaviours gracefully. The GOBT is one of the sub-modules of the core AI module in Gameme. Inside Gameme, agents are capable of planning and learning their own goals based on environmental information. Agents are directed toward a goal based on continuous evaluation of perceptual inputs. We believe the GOBT data structure and the goal processing architecture with the arbitrator can be applied to a wide range of game types.

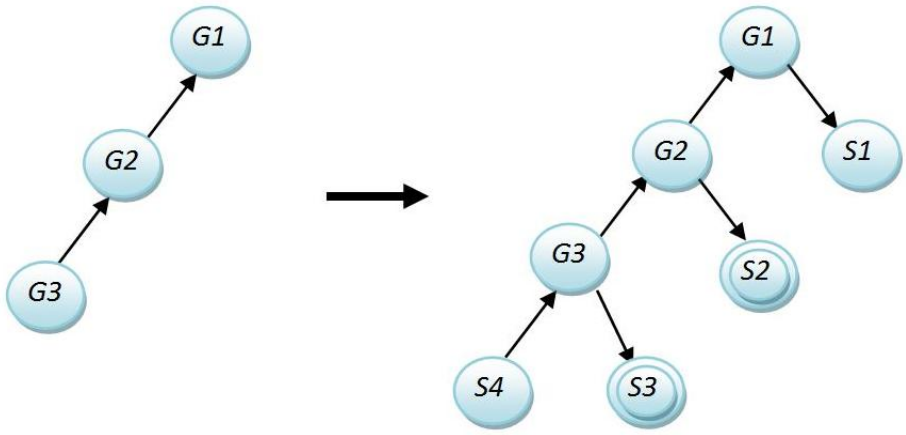

Figure 22: The Generation of the GOBT "Eat"

Goals for "Eat":

G1: *Xman eats food*
G2: *Xman goes to food*
G3: *Xman searchs food*
$G3 \rightarrow G2 \rightarrow G1$

Rules for "Eat":

R1: *If food is reachable, eat it*
R2: *If food is visible, go to it*
R3: *If there is no food, search for it*

States for "Eat":

S1: *eating food*
S2: *go to food (TC=arrive food position)*
S3: *search for food (TC=find food)*
S4: *there is no food*

Execution Order (Section 6.3.3) :

$S4 \rightarrow G3 \rightarrow S3^* \rightarrow G2 \rightarrow S2^* \rightarrow G1 \rightarrow S1$

Figure 23: The GOBT "Eat"

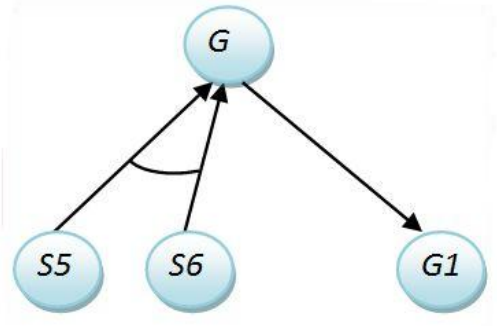

Figure 24: The GOBT “Eat” #1

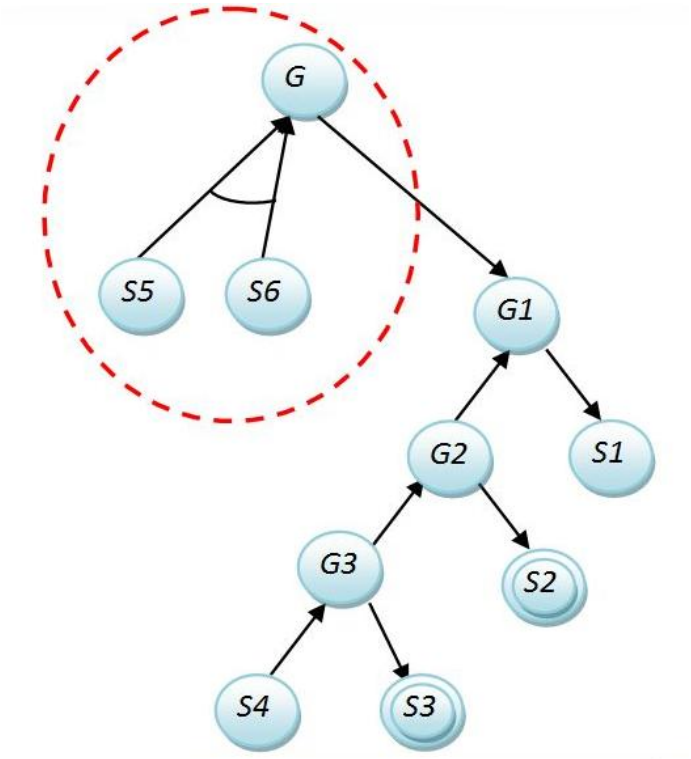

Figure 25: The Full Picture of GOBT "Eat" #1

New goals and states for the GOBT "Eat"#1:

G: *Xman is ready for food*

S5: *There is no Monster around*

S6: *Xman feels hungry*

New rule for GOBT "Eat"#1:

R: *If "there is no Monster around" ∧ "Xman is hungry", Xman is ready for food*

Execution Order (Section 6.3.3) :

$(S5 \wedge S6) \rightarrow (G \wedge (S4 \rightarrow G3 \rightarrow S3^* \rightarrow G2 \rightarrow S2^*)) \rightarrow G1 \rightarrow S1$

Figure 26: New Goals and States for GOBT "Eat"#1

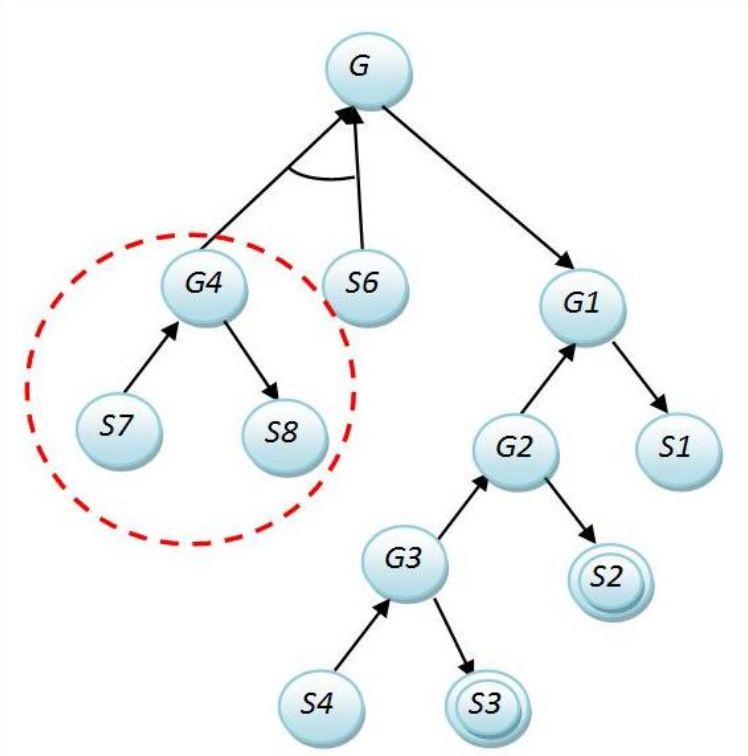

Figure 27: The GOBT "Eat" #2

New goals and states for the GOBT "Eat"#2:

G4: *There is no Monster around*

S7: *Xman checks whether Monster is in surrounding area*

S8: *Xman is Idle*

New rule for GOBT "Eat"#2:

R4: *If Xman checks there is no Monster around, he is in Idle*

Figure 28: New Goal, States and Rule for GOBT "Eat" #2

Chapter 6

Game Agent Modeling

Modeling is the act (or process) of *identifying* appropriate phenomena and concepts and of choosing appropriate abstractions in order to construct a model (or a set of models) which reflects appropriately on the *universe* of *discourse* being modeled [Bjo07]. When modeling game agents, we analyze the domains of game agents, and identify characters of game agents. Finally, we specify the abstract model of game agents. The abstraction of game agents that we describe in this chapter is not for a specific game: it can be used in modeling characters in many games.

6.1 The Nature of Game Agents

The term "agent" was introduced into the computer industry in the 1990s. An *agent* is a system that tries to fulfill a set of goals in a complex, dynamic environment [Mae94]. An agent is situated and involved in the environment: it can sense and act upon the environment. An agent's goals can take many different forms: they can be "end goals", or particular states the agent tries to achieve; they can be a selective reinforcement or reward that the agent attempts to maximize; they can be internal needs or motivations that the agent has to keep within certain viability zones, and so on [Mae94].

Game development has as a main goal the *building* of intelligent entities. It turns out

that such entities are very important in the game world. We consider these entities as game agents. As the game industry has developed, the agent concept has been associated more and more with game characters' design. We already use the term game "agents" to designate NPCs and PCs. Usually, PCs are characters controlled by human players. By contrast, NPCs are in game characters whose behaviour is not controlled by human players.

A PC may be based on a real person or a fictional person. Behaviours created by different human players have huge differences. PC's behaviours have a certain level of uncertainty. However, PCs' real-time behaviour are still traceable. In order to confine PC's behaviours, game designers might provide a set of basic behaviours to human players. Human players can arrange these basic behaviours to create complicate behaviours in real-time. For example, in fighting games, human players can combine different basic KongFu actions together to form complex actions.

A NPC is a game character designed by game designers off-line. It may have similar basic behaviours as PCs in the same game. However, the NPC is completely controlled by the game in real-time. In our system, the core AI module control NPCs' behaviours. The AI module has the capability to provide real-time planning, learning, and team behaviours control for NPCs. With the knowledge of the basic behaviour set, the AI module can find the correct reaction for NPCs in order to interact with PCs.

Nowadays, simple code and simple data structures are not sufficient for game characters. Game agents are parts of the virtual reality world, know things, and carry out reasoning. Both knowledge and reasoning are important for game agents because they enable intelligent behaviours that would be very hard to achieve otherwise.

We have already discussed the knowledge representation: Goal Oriented Behaviour Design (GOBD) for game agents. The GOBD provides the general format to describe knowledge for game agents. Game agents can benefit from knowledge expressed in the form of GOBD and recombining information to suit different purpose. Also, game agents can combine general knowledge with current percepts to infer hidden aspects of current state by reasoning.

In addition, during the interaction with the environment, game agents can learn new knowledge about their environment and adapt to changes in the environment by updating relevant knowledge.

We use the agent concept for game characters as a tool to analyze and design the game system. So, both the game environmental factors and agent theory are counted into the game agent modeling. In this chapter, we provide a brief explanation of how to model game agents based on GOBD.

6.1.1 The Environment of Game Agents

Usually, a game is a virtual reality world which consists of different game agents. Game agents interact with each other and generate a vivid game environment. The behaviour is the result from the interaction dynamics between the agent and the environment [Ste94]. In our discussion of the modeling of the game agent, we have to start by specifying the game environment.

In general, game environment is different based on miscellaneous game genres. However, PCs and NPCs are main entities in the game environment. We consider the *environment* for a game agent to be other game agents' activities which affect the game agent's activities. The environment is the main factor that leads game agents to change their goals. In detail, NPC's action induce the human player change PC's action. On the other hand, the PC's action result in the NPC's action change.

In this research project, we pay particular attention to NPCs' behaviour control. PCs' behaviour is the decisive factor in game environment which affects NPCs' behaviour. In detail, we create a fighting game environment that we use throughout the main testing cases in this research project.

6.2 Characters of Game Agents

An intelligent game agent operating in a game environment will often need to interact with other game agents to achieve its goals. Agent modeling—the ability to model and reason other agents' knowledge, beliefs, goals, and actions—is central to intelligent interaction.

If we are to understand the interactions of a large numbers of agents, we must first be able to describe the capabilities of individual agents [Hol95]. Nwana provides a typology for software agents [Nwa96]; for example, agents can be classified based on mobility, deliberative or reactive, roles.

Since each game type has its own strengths and deficiencies, the best way for modeling game agents is to use hybrid agents modeling. The hybrid agent combines two or more agent philosophies within a single agent. The key hypothesis for having hybrid agents or architectures is the belief that, for some applications, the benefits accrued from having the combination of philosophies within a singular agent is greater than the gains obtained from the same agent based entirely on a singular philosophy [Nwa96]. These philosophies can be flexible based on different game types. Different game types refer to different game environment for game agents. The flavor of the game environment directly affects the design for game agents.

After reviewing different agent philosophies, we include autonomous, adaptive, and collaborative philosophies in the modeling of NPCs in our project. These agent philosophies often relate well to the way we naturally think about complex tasks, and thus agents can be useful to model such tasks in games.

- Autonomous agents are systems that inhabit a dynamic, unpredictable environment in which they try to satisfy a set of time-dependent goals or motivations [Mae94]. An autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to reflect what it senses in the future [FG96]. Such agents can produce everyday phenomena such as ant colonies, traffic jams, stock markets, forest ecosystems, and supply chain systems [Ode00].

- Adaptation implies sensing the environment, discovering the problem solving strategies and reconfiguring in response. Agents are said to be *adaptive* if they improve their competence at dealing with these goals based on experience [Mae94]. It can be the choice of different problem-solving-rules when agents face alternate interaction characters.

- Collaboration happens when game agents are grouped and working together against antagonistic game agents.

We use these philosophies in the modeling of NPCs. This classification allow us to employ AI methodologies to agents which creates an in-game complex environment.

6.3 Modeling of Game Agents

In developing an agent to act as a gaming NPC or PC, we are particularly interested in developing a suitable structure for supporting varied in-game behaviours. The BDI agent model (see Section 2.6) consider agents to have *beliefs*, *desires* and *intentions*. In order to cooperate with game development and GOBD design methodology, we consider each game agent is a BDI-like agent which consists of four basic components (B, D, I, P) (*beliefs, desires, intentions, plans*) connected by an interpreter as showed in Figure 29.

6.3.1 Beliefs

The first component is a database of *beliefs* B, or knowledge. It is the information of the game agent itself and its opponents. Each agent has its own beliefs including *external* and *internal* beliefs. External beliefs consist of other agents' knowledge which affects behaviours of the agent. External beliefs comprise the information perceived by the game agent. Or, we can say external beliefs form the environmental information for the game agent. The internal beliefs form the internal factors of the agent. In addition, for a team of game agents, external beliefs refer to the opponent's status; internal beliefs refer to the game agent's own status.

A game agent's current external and internal beliefs are used as parameters of both the planning and learning algorithms. The separation of internal and external beliefs for each

game agent comes from the nature of game agents. Game agents have their current desire to pursue a goal. However, environmental factors could affect their next reaction; furthermore, change their intentions. The internal beliefs represents game agents' personal status. It could help the planning and learning algorithms to find the most suitable solution for game agents. For example, when a group of game agents fight with the PC, the PC launches an all-out offensive. Since each game agent has different level of power, agents might choose to fight (high power), defense (medium power) or escape (low power). So, the external beliefs refers to the PC's fighting status which is offense; the internal beliefs refer to each game agent's power level which is high, medium or low power .

6.3.2 Desires

The second component is a set of *desires* D, or goals. Goals are the embodiment of a game agent's personality. Each game agent has a number of embedded goals. These goals were prototyped by game designers. They are predefined off-line and take forms of GOBTs. Conventionally, we use the root name of GOBT as the goal name. Game agent's current goal is its motivational state of action. If a game agent has a set of goals, these goals represent the availability of real-time behaviours that planning and learning algorithms can retrieve.

Goals provide restrictions for game agents in order to achieve consistent behaviours. We restrict each agent to pose only one goal at a time. Planning and learning algorithms decide whether or not to change the game agent's current goal in real-time. For example, a game agent must choose either to *sleep* or to *eat* even if both of these goals are desirable.

The idea of specifying desires for each game agent tightly is consistent with the GOBD. Game designers can start modeling game agents by defining goals and GOBTs. The component *plans* is generated automatically after the creation of GOBTs. One extra task is to specify goal priorities for the purpose of creation the component *intentions*.

6.3.3 Plans

The third component is a list of *plans* P, or goal execution orders. Plans are abstracted from GOBTs. They are in the form of a set of goals or states in sequences. Plans are associated with GOBTs. For example, we list two plans (execution orders) for a GOBT in Figure 16 on page 64. Each plan is a list of subgoals or states toward achieving a particular goal. Each GOBT contains execution orders which are sets of behaviours or goals, which in turn are the execution sequence of achieving the highest level goal (the root of the GOBT). The execution order supplies quick, simple control in situations where actions reliably follow one from another. For each GOBT, there can be several execution orders to indicate different ways to achieve the same goal. For example, in Figure 12 on page 59, in order to achieve the root goal *B1*, we could start from either leaf node *B5* or *B3*. Consequently, goal execution orders could be different for each leaf node.

6.3.4 Intentions

The final main component is a set of priority queues of *intentions* I, or priority factors. A priority queue for an agent indicates that only one *desire* (goal) is actually driving the agent's activity at a time, but multiple *desires* (goals) may be on the priority queue. Different priority queues represent different intentions for game agents. For example, in Figure 32 on page 97 and Figure 31 on page 97, based on the visibility of Monster, the game agent Xman has two priority queues to indicate different intentions in different circumstances. Priority factors in the same priority queue represent importances of goals for an agent. In Figure 32, different desires have different priority values.

6.3.5 Formal Representation of Knowledge Modules

Formally, knowledge modules used in game agents' planning and learning are defined as below. We assume that there are N game agents in a team, and M records in the experience database.

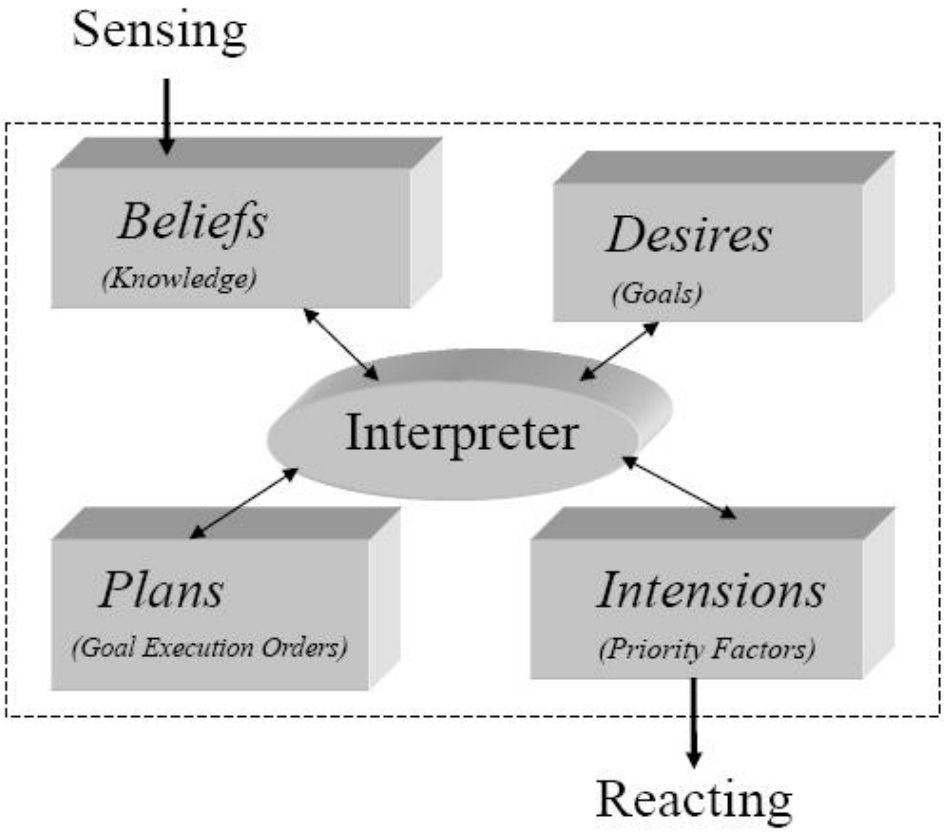

Figure 29: The Model of Game Agent

First, *beliefs*, *desires*, *intentions* and *plans* (B,D,I,P) are the four basic modules used to model a game agent. These four modules contain not only off-line design information but also real-time statuses for a game agent.

- A set of *beliefs*, $B = \{B_e, B_i\}$ (Section 6.3.1).
  B_e refers to external beliefs; B_i refers to the internal beliefs.

- A set of *desires*, D (Section 6.3.2)

- A set of *intentions*, I (Section 6.3.4)
  In the processes of learning and planning, a team of game agents intentions can be specified as Cartesian products $\prod I_k$, $\prod I'_k$ and $\prod I''_k$ ($k \in [1, N]$) which are outputs of planning level, first learning revision and second learning revision.

- A set of *plans*, P. (see Section 6.3.3)

In order to cooperate with game agents learning precesses, we extended basic knowledges modules to these three modules as below. Modules *experience*, *strategies* and *behaviour pattern* rearrange game agents' real-time status as referential information for learning processes and enhance game agents behaviour from individual behaviour to team behaviour.

- A set of scalar *experience*, E_j ($j \in [1, M]$); A set of reward values, γ_j ($j \in [1, M]$) (see Section 8.4).

  In the learning level, in order to process game agents learning, we add an additional knowledge module *experience* which is regarding a team of game agents' previous performance. Records in the *experience* serve as the source task for the transfer learning.

- A set of production rules *strategies*, S. (see Section Figure 8.4.2)

  Strategies are used in the learning process for game agents adaptive learning. A set of production rules is predefined by game agents in order to control team behaviours in the adaptive learning.

- The *behaviour pattern* is denoted as ω. (see Section Figure 9.3.1)

  Behaviour patterns are reference factor for game agents' emergent learning. It is abstracted from game agents' real-time performance during a period of time. It is a supplementary knowledge module which is abstracted in real-time and used to optimize game agents behaviour in emergent learning.

6.4 Conclusion

We present a methodology to model game agents in this chapter. This methodology is based on procedural behaviour representation and GOBD. In order to add basic autonomous ability to game agents, we adopt the idea of BDI-like agent design. Each game agent has four basic knowledge components, *beliefs, desires, intentions, plans*, or showed as (B,D,I,P).

Furthermore, in order to cooperate with further game agents' planning and learning, we defined a list of additional knowledge modules, such as *experiences* E, *strategies* S and *behaviour patterns* ω. These modules are all based on those four basic components in game agents. In fact, game agents' planning and learning processes described in Chapters 7, 8 and 9 are processes of manipulating these knowledge modules for the purpose of game agents' behaviour control.

Chapter 7

The Procedural Planning

As we discussed layered planning and learning in Section 3.5 and showed in Figure 8 on page 39, the planning layer is the lowest layer in the Layered Planning and Learning structure. This layer focuses on individual game agent's planning. We provide a detailed explanation of game agents' procedural planning in this chapter. Parts of this chapter were originally published as [SG08b] and [SG09c].

7.1 Descriptive vs. Procedural

The concept of distinguishing between procedural and descriptive has been advocated for a long time, and it still has exploratory power. For game designs based on Gameme, game designers have the rough and symbolical design of the whole game during the game design phase. Then, Gameme has to transfer the game designer's idea into C++ code that can actually run the game in computers. It is a process from loose validation to strict validation. Consequently, we consider that the process of generating a game is a process that moves from the descriptive stage to the procedural stage. If we zoom in to the agent's control level, the descriptive (or logical) planning of agent's activity, which describes states of agents in different situations, has to become the procedural (or presentation) format, which specifies how the planning should be presented in real-time.

7.2 PRS and PPS

After reviewing a couple of traditional agent architectures, we adopted procedural agent control into game development. The Procedural Planning System (PPS) is based on the PRS [GL87] which has been used in robotics for about a decade. PPS and PRS are similar in spirit. However, the embodiment of the procedural knowledge philosophy in the two systems is different. PRS is a general-purpose reasoning system designed in a broad range of control systems. The PPS is a redesigned game agent planning system that relies on goal-oriented pre-designed plans and provides reactive planning in real-time. In addition, the PPS supports modular GOBTs as primitives, which have their own design methodology.

7.3 Features of Procedural Planning System

The planner's search is the last obstacle in reliably planning fast enough for real-time [Ork05]. The PPS is a simplified and efficient game agent architecture. It is ideal for planning where actions can be defined by predetermined procedures that are possible in the game environment. This simplifies the game agent architecture because it selects plans based on the run-time environment and the highest priority goals instead of generating plans. While planning is more about selection than search or generation, the interpreter ensures that changes to the environment do not result in inconsistencies in the plan [Jon08].

In addition, the PPS can efficiently avoid slow planning in dynamical game environment. All predetermined plans are early selections based on possible circumstances during the game design process. Early selection of goals can eliminate most search in unnecessary branches, making goal-driven search more effective. While the selection is the main planning operation in each planning cycle, the 3D engine can accept the planning and learning result in real time, and render the correct visual reaction for game agents.

The planning repeatedly operates over a short processing cycle. Each game agent can only focus on one goal at a time. So the partial plan is a part of a goal's execution order of an agent. The interpreter decides either switch to another goal or stay with the same goal in the

next planning cycle. If there is no change for the current goal, the game agent will continue to execute the next state in the same plan. This result is intentional goal oriented planning in which game agents' activities are expanded in a manner analogous to the execution of subroutines in procedural programming systems.

The dynamical procedural planning has two aspects of consideration. The first one is multi-goal planning; and the second one is multi-agent planning. These two aspects require that the planning system have the ability to create plans that consider information about not only its own status but also other agents' information related to it.

7.4 The Interpreter of PPS

The PPS consists of four main components connected by an interpreter, as shown in the center of Figure 29.

The interpreter is an inference mechanism which controls other components in the PPS in the reacting planning process. The interpreter exchanges information with these four components (*Beliefs, Desires, Intentions, Plans*), and drives the process cycle of sensing, reacting, and planning.

The interpreter is the key component in the PPS. It repeatedly executes the set of activities depicted in Figure 30. In planning cycles of the PPS, certain goals are established and sent to the game rendering engine or the learning level. There are seven interpreter activities included in the PPS.

1. The external beliefs are sensed. Also, the interpreter includes current intention which were memorized in the previous planning cycle.

2. The external beliefs are used as conditions to search for matching results in rules related to GOBTs.

3. If the agent needs to change its current intention, several candidate priority queues are checked, and the top goal in the queue is chosen.

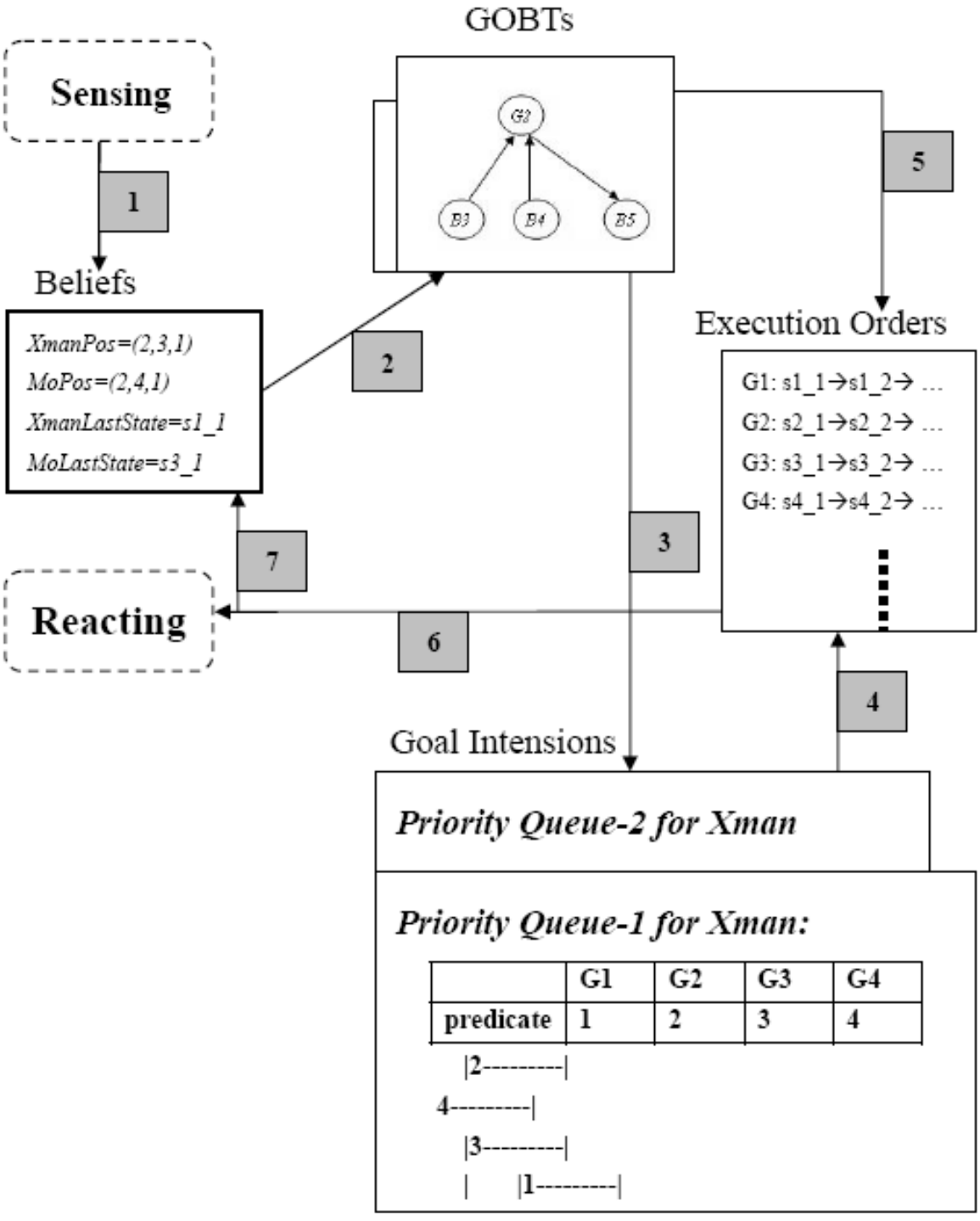

Figure 30: The PPS Planning Cycle

4. Based on the new goal, the new state of the agents can be found in execution orders.

5. On the other hand, if there is no need to change current intention, the agent's next state are obtained from execution orders directly.

6. After planning, the agent's new state is sent to the action module.

7. Finally, the agent's new intention is recorded for the next round of planning.

Each game agent has an interpreter. The interpreter is responsible for managing the sensing, planning and reacting processes of each game agent. The procedural planning layer is a reactive module that provides functionality of goal selection. It has a unified mechanism for making decisions, but the output is different for each agent based on their own situations.

A procedural knowledge based game agent is represented as $\{B, D, I, P\}$, a tuple of four elements: *Beliefs*, *Desires*, *Intentions*, and *Plans*. The *interpreter* of each agent can be represented as a function

$$G : B \times D \times I \times P \mapsto I_k$$

with the output of new *intention* I_k of this agent. The I_k is a priority queue which indicates the next intention and the desire (the top goal in the priority queue) for the game agent. k is the agent ID in a team.

7.5 The Planning Cycle

The planning level is a process concerned with decentralized team behaviour. It provides each game agent with an opportunity to develop its personality. In order to have the most appropriate reaction, each game agent takes into account the opponent's status B_e and its own status B_i. Essentially, the planning is based on the idea of "external belief decides the current *Intention*, internal belief decides the current *Desire*". Initially, the game agent has its current intention I_k and desire $D_{current}$. The $D_{current}$ is the top goal in the I_k priority queue.

Here is the planning algorithms that the interpreter executes in each planning cycle.

1. *Analyze current B_e;*

2. *If B_e has no change, no change in the current I_k , and go to step 4.*

3. *Else, The interpreter decides which new intention I_{new} the game agent should divert its attention to. Then, the interpreter retrieves the I_{new} from the predefined knowledge and replaces it with current intention, $I_k = I_{new}$.*

4. *Analyze current B_i;*

5. *While the priority queue of current I_k is not empty, check the top goal $D_{current}$;*

   - *If $D_{current}$ is not matching B_i, pop $D_{current}$;*
   - *Otherwise, return $D_{current}$ as current goal;*

6. *If the priority queue I_k is empty, the interpreter can't find a matching goal. It returns the previous status (before last pop) of the priority queue as I_k;*

Goal arbitration happens all the time during planning. Some hybrid architectures consider this problem to be in the domain of "deliberation" or "introspection"—the highest level of a three-layered architecture [Bry03]. But the PPS in Gameme treats this problem as a general problem of goal selection. The interpreter is a reactive module that provides functionality of goal selection.

The planning level provides intuitive reaction for game agents, and leaves deliberative agent behaviour process to the learning level. It is the design methodology regarding lightweight game agent architectures. In addition, not every game agent needs further behaviour control from learning level. Game designers might consider having game agents with simple and fast behaviour reaction. The separation of planning and learning processes also is the consideration of different intelligent levels in designing behaviours for game agents.

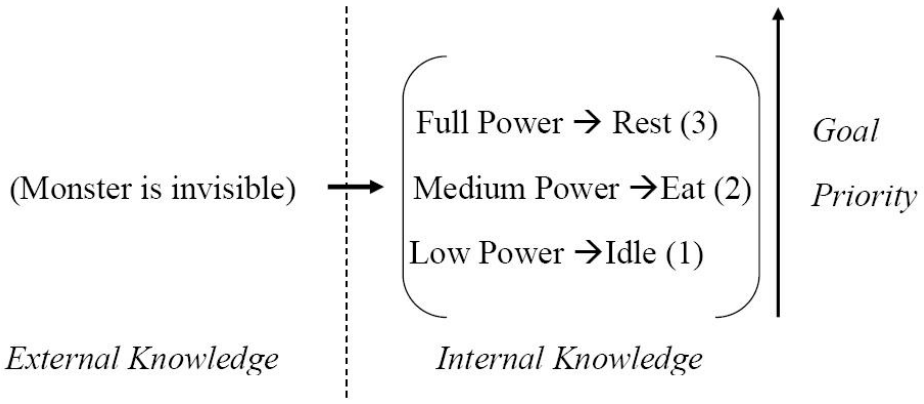

Figure 31: Priority Queue: Monster is Invisible

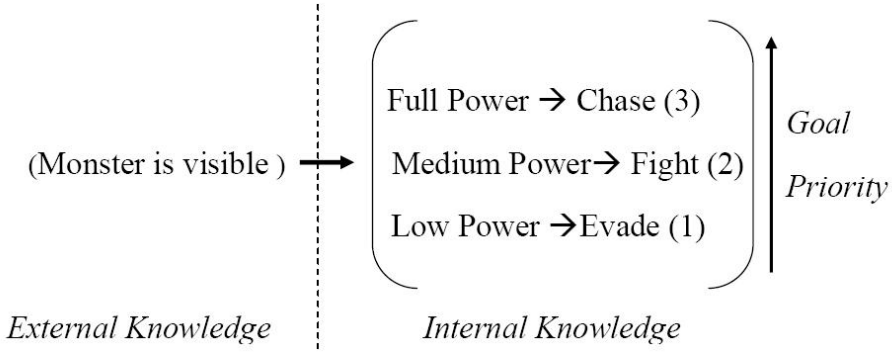

Figure 32: Priority Queue: Monster is Visible

7.6 Testing for Procedural Planning

During each cycle of planning, beliefs typically cause the change of the goal attention (priority queue), and then change the goal (the goal in the top of the priority queue). In the latter example, suppose that the Xman detects that the monster becomes visible (external beliefs), but its current goal is "Eat" in the priority queue (Monster-Invisible) and Xman is in less power status (internal beliefs). Consequently, during the planning for the next state of Xman, the interpreter decides to switch to the priority queue (Monster-Visible) and to choose the goal "Evade" based on Xman's power level.

The agents' behaviours are defined statically off-line and planned dynamically in real-time. Game agents have to react to the environment appropriately and quickly in real-time. Usually, game designers define several goals for an agent. These goals can be grouped in

two priority queues based on the visibility of the Monster. If the monster is invisible, Xman can select either "Rest","Eat" or "Idle". If the monster is visible, Xman has goals "Evade", "Defense" and "Fight". Figure 31 and Figure 32 indicate goals and their priorities queues for Xman.

During planning, the agent has to make correct decisions based on knowledge changes. We treat distance between the monster and Xman as the external beliefs, and Xman's power value as the internal beliefs. Xman can gain power when it is eating and resting; keep power when it is idle; and other activities (goals) reduce the power value.

In the Xman example, if the monster is invisible at the beginning, the Xman focuses on the queue *Monster-Invisible* as in Figure 31 on page 97. If Xman detects the monster, it switches to the queue *Monster-Visible* as in Figure 32 on page 97. Here, we explain how the interpreter works when Xman meets a monster.

(1) The interpreter decides to focus on the *Monster-Visible*. The priority queue is initialized. If Xman is not in the full power level, the top goal "Chase" is popped.

$$\begin{aligned} & Heap[1..3] &=& [3(Chase)][2(Fight), 1(Evade)] \\ \xrightarrow{Pop} \quad & Heap[1..2] &=& [2(Fight)][1(Evade)] \end{aligned}$$

(2) The goal "Fight" becomes top goal in the queue. If Xman is in medium power level, the goal "Fight" becomes the current goal.

$$Heap[1..2] \;=\; [2(Fight)][1(Evade)].$$

(3) If Xman is still in the medium power level, the interpreter changes nothing in the queue that it is focusing on. So Xman keeps fighting with the monster, and the goal "Fight" remains active.

(4) If Xman defeats the monster, the interpreter detects that the monster is not around. So it switches to the queue *Monster-Invisible*. The queue is initialized. After fighting, the power level of Xman is not full, so the top goal "Rest" is popped. And the goal "Eat" becomes the current goal.

$$\begin{aligned} & H[1..3] &=& [3(Rest)][2(Eat), 1(Idle)] \\ \xrightarrow{Pop} \quad & H[1] &=& [2(Eat)][1(Idle)] \end{aligned}$$

7.6.1 Visual testing result fro Planning

We use the open source 3D graphics engine OGRE to simulate the game scenario. The 3D modeled robot is used to represent the monster; and the 3D model jaiqua is used represent Xman. The monster is controlled by a human player. The monster can move around in the scene, and Xman's behaviours is controlled by the procedural planning mechanism. These results show that the agent planning is efficient, and there was no delay in 3D animation rendering.

Xman's behaviour is controlled by its PPS. In Figure 33, when the monster was close, and Xman was in low power status, Xman ran away from the monster. And in Figure 34 on page 100, Xman ran out of the range of his range of visibility of the monster, Xman switched to the priority queue Figure 31, and sneaked around since Xman was still in low power level. Figures 35 and 36 on page 101 show interactions of two Xman and the Monster. When a Monster is visible, two Xmans have reactions based on their own power level.

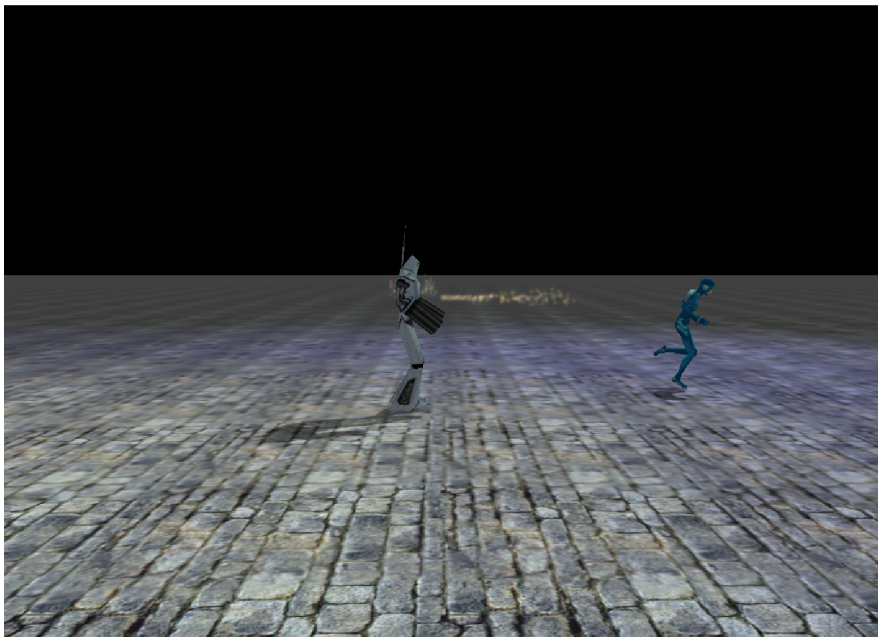

Figure 33: When the monster is visible, Xman runs away.

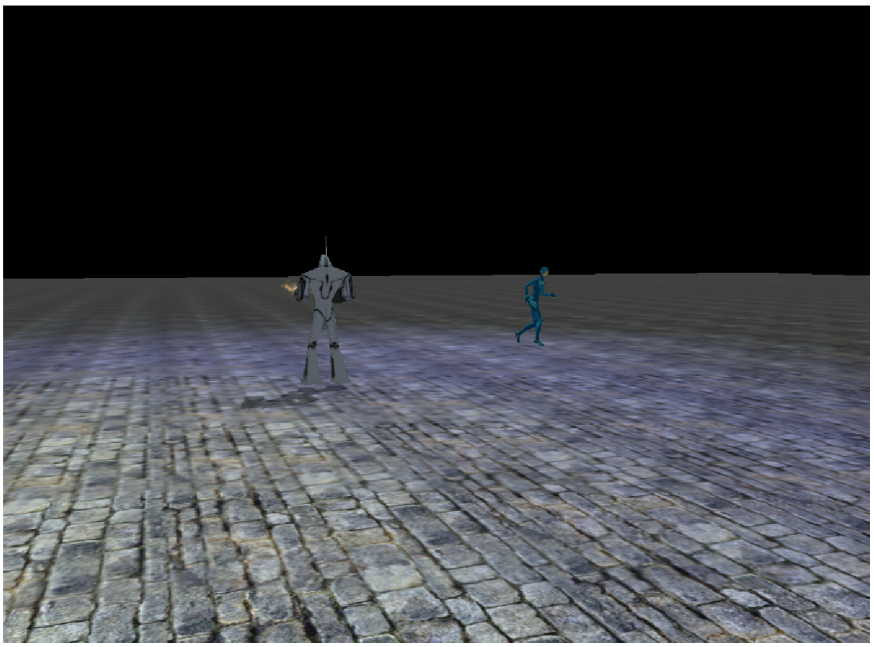

Figure 34: When the monster is invisible, Xman sneaks around.

7.7 Conclusion

We have presented a "Procedural Planning" technique to be used in the game agent architecture of Gameme. The main contribution is the procedural knowledge approach used for game agent planning in real-time. It allows action selection to be reactive within a complex runtime game environment, rather than relying on memory of predefined plans. Also, the procedural planning benefits from the modular design from the knowledge design to the whole system architecture. In addition, the idea of handling a light-weight agent architecture by aspects of balancing off-line and real-time design and usage of the controller pattern is also useful in other game agent systems. Evaluating this type of system requires applying them to real situations. Our visual testing results indicate the efficiency and flexibility of procedural planning. It is suitable for game agents' real-time planning and further extension in complex behaviour control.

Figure 35: Two Xmans are both in medium power; they both stagger.

Figure 36: The Xman in the left side has medium power, so it staggers. The Xman in the right side has less power, so it runs away.

Chapter 8

The Transfer and Adaptive Learning

As the game industry has evolved, game players have wanted NPCs to act more and more intelligently and unpredictably. Game players prefer NPCs to appear as though they understand their environment and adapt to changes in the environment. Dynamic NPC behaviour require on-the-fly decision to be made by the NPC not only to react in their environment but also to learn from their environment. To create such abilities, functionalities provided by the planning level are not enough for NPCs. So, we turn to the level of learning in Figure 8 on page 39. Parts of this chapter were originally published as [SG09d] and [SG09b].

8.1 Overview of the Learning Level

The learning layer extends individual game agent's planning to a multi-agent learning, especially the team cooperation in real-time. The functionalities of transfer learning and adaptation are two main aspects of the coordinator. Our approach combines both the transfer learning and the adaptive mechanism in order to optimize the team behaviour in real-time. The transfer learning is applied to coach the team based on its *Experience*. The adaptive mechanism is used to enhance the whole team performance by *Strategies*.

In games, designing NPCs that are unlikely to lose requires building highly adaptive models that can substantially adapt to opponents with a rapid shifts in play strategies.

The adaptive mechanism updates and optimizes the transfer learner. The adaptive mechanism involves reevaluating the assumptions made about the team's experience, and making decisions for each agent as the purpose of maximizing the team performance.

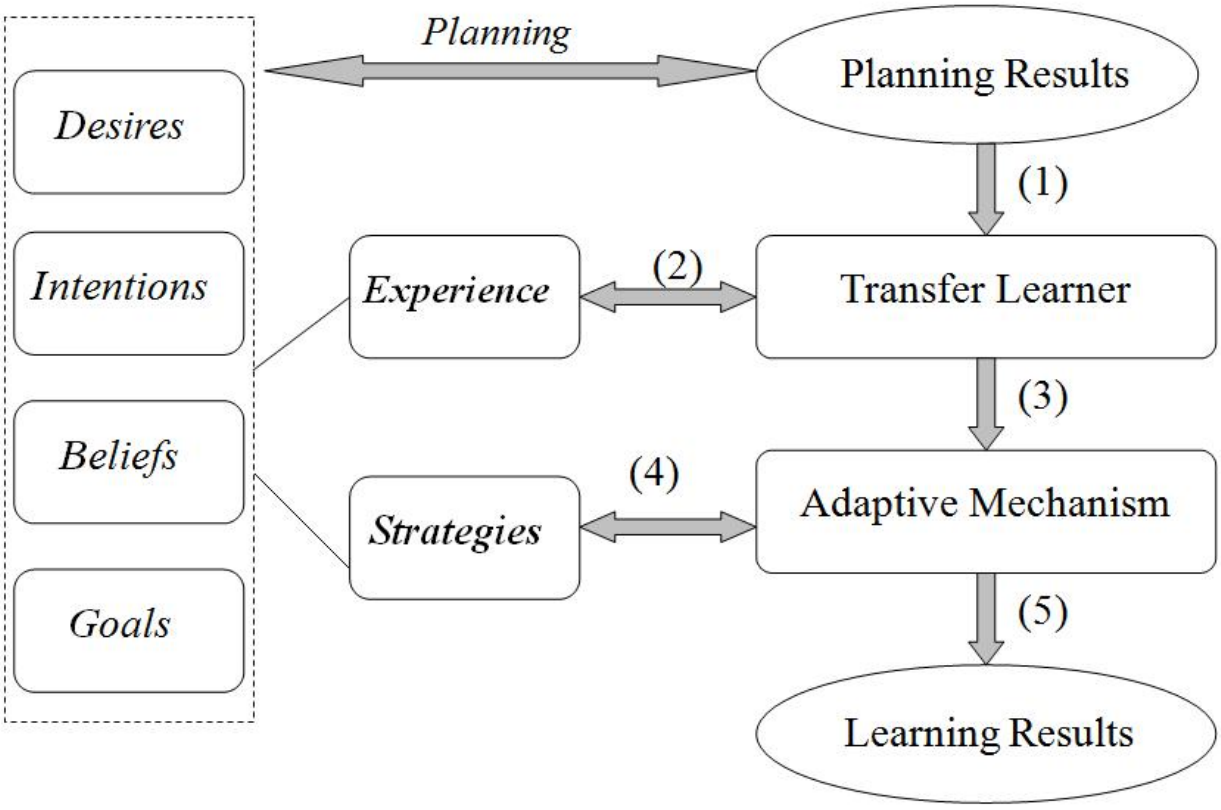

Figure 37: The Transfer and Adaptive Learning Process

8.2 Learning vs. Planning

The planning level is more about action selection while the learning is more about action reinforcement. The planning level provides real-time reaction for each game agent. However, in the planning level, each game agent has a limited viewpoint, and therefore has an incomplete capability to solve global problems, such as team cooperations and adaptation. The learning process is based on the output from the previous planning level. It centralizes each NPC's procedural planning result, and revises them for the adaptive and transfer learning

purpose. These learning mechanisms extends individual game agent's planning to a level of team behaviour control.

In general, learning occurs on a wider scale than the planning process, and considers more resources than the planning level, inclduding, for example, *behaviour patterns*, *Experience* and *Strategies*. These resources are accumulated from a number of rounds of learning cycles. In addition, the cooperator in the learning layer analyzes interactions between PCs and NPCs, generates PC's behaviour pattens, and maps the most suitable *Intentions* to NPCs.

8.3 The Team Behaviour Control

In the domain of multi-agent system, communication is an important characteristic for supporting both coordination and the transfer of information [Jon08]. After having individual planning abilities, game agents should have abilities of cooperation in team work. Most Multi-Agent Systems (MAS) assume that agents can interact via a well-defined language or through observation of one another's actions. In this learning level, game agents cooperate in a team. Agent's cooperation is controlled by the coordinator inside the team. In addition, the team interacts with their opponent by observing opponent's action B_e.

The squad behaviour control requires different agent control mechanisms used in the learning layer. In the planning layer, every game agent has an interpreter for the individual agent's planning. In addition, in the learning layer, there is a coordinator which is used for the whole team behaviour control. The coordinator interacts with all knowledge components in each learning cycle. At each learning cycle, the coordinator perceives each game agents' planning result I_k ($k \in [1, N]$), and groups them together as $\prod I_k$ in the process of learning.

The team learning process is an incremental process. It requires at least two revised stages to get the best result for the transfer learning. An important component of this incremental design involves different steps of reevaluating assumptions made about game agents.

- The first evaluation compares planning assumptions with game agents previous experience.

- In order to have a best team performance, even when assumptions about game agents can be successfully revised based on their past behaviour (*Experience*), the process still requires further revision regarding the total of team strategies intended to defeat the opponent. So, we define a second revision for the team based on *Strategies* defined in the adaptive mechanism.

8.4 The Transfer and Adaptive Learning Mechanism

In the learning layer, the transfer learning is based on the output from the previous planning level, and provides revised control over the whole team. Key aspects that contribute to the ability to achieve the desired transfer learning include recognizing the applicability of existing knowledge to novel situations and mapping from source knowledge to target knowledge.

Transfer learning is essentially an online machine learning algorithm. The source or target tasks implemented by the same group of game agents. So the process of task mapping from source to target is not the main concern of this research project. We focus mainly on the design of how to optimize existing experience in order to adapt it to the real-time game environment [SG09b]. In addition to *Beliefs*, *Desires*, *Intentions* and *Plans*, there is a knowledge component added in this level, the *Experience*.

For the transfer learning, we consider adding *Experience* as an additional factor in the decision making. *Experience* is the training data source which is a combination of a series of successful team performance. It is presented as a small database of experienced records. In general, the jth record in the experience database is represented as a tuple of three objects,

$$E_j \;=\; \left\{\gamma_j, B_e, \prod I_k\right\},$$

where j is the record ID of this experience record, k is the game agent ID in a team. The experience records are grouped based on different B_e values.

There are two stages to create *Experience* for a team.

1. The first stage focuses mainly on *Experience* accumulation.

   It first happens during early rounds of interaction between the team and the opponent. It is a period of establishing learning reference. During this period, the reward function calculates reward value γ. Records combining of the reward value, corresponding intentions and external beliefs are stored into the Experience *database.*

2. The second stage is a stage of *Experience* application.

   It is a process of replacing current game agents' intentions with the intentions in the experience database.

These two stages may overlap each other: in particular, the experience database is updated sometimes in order to keep records for best performance. Transfer learning does not provide an overall model for different target tasks; however, it trains target tasks based on real-time updated best models. The coordinator updates the experience when it gets a better experience record than the existing record in the same external knowledge B_e level.

8.4.1 The Reward Function

The reward function is the objective function for the transfer learning. It conducts a reward value γ which corresponds to a record in the experience database. The reward function counts in the internal and external beliefs changes. It specifies the reward from combats between the team and the PC. The reward value γ is used in the transfer learning which makes effort to maximize the team performance. It can be described as

$$R : B \times \prod I_k \mapsto \gamma.$$

So the transfer learning function can be represented as

$$T : \prod I_k \times B \times \prod E_j \to \prod I'_k$$

where j is the record ID of this experience record and k is the game agent ID in a team.

8.4.2 The Strategies

In the learning level, we adopt the idea of adaptive mechanism used in the control of team behaviour. We add another new knowledge module *Strategies* in this mechanism. The team behaviour changes in response to observed opponent's behaviour. In detail, *Strategies* is a number of production rules which steer the team of agents' behaviour. It is the highest level control over the team. These strategies treat the team as a whole; and the opponent of the team is the cause of the whole team's behaviour change.

Strategies can be represented as production rules with conditions and results. Conditions are different situations of the opponent B_e, and results are the team intention $\prod I_k^{''}$ corresponding to the opponent. In Section 8.4.4, we provide examples of strategies for a team of agents. The adaptive mechanism can be represented as

$$A : \prod I_k^{'} \times B \times S \rightarrow \prod I_k^{''}.$$

8.4.3 The Transfer and Adaptive Learning Algorithms

Figure 37 describes the coordinator's work flow during each learning cycle. The first revised process are steps 1 and 2. The second revised process are steps 3 and 4.

1. *The* $\prod I_k$ *sent from the planning layer; the* $\gamma_{current}$ *is calculated.*

2. *The* $\gamma_{current}$ *is compared with* γ_j *in* E_j *which has the same* B_e*. If* $\gamma_j > \gamma_{current}$*,* $\prod I_k^{'}$ *is retrieved from* E_j*, and replaced the current* $\prod I_k$*. Else, the current* $\prod I_k$ *is assigned to* $\prod I_k^{'}$*.*

3. *The* $\prod I_k^{'}$ *is revised based on team Strategies. Some game agents'* $I_k^{'}$ *might have to change in order to match the whole team behaviour.*

4. *If the second revised process requires change in the* $\prod I_k^{'}$*, related intentions is retrieved from knowledge modules and replace the existing* $I_k^{'}$*.*

5. *The second revised* $\prod I_k^{''}$ *is output from the learning layer.*

8.4.4 Testing of the Adaptive and Transfer Learning

In this section, we present testing results of transfer and adaptive learning process for two teams of game agents when they fight with a human player controlled monster. These testing results are both based on the basic procedural planning. Each team always tries to keep every member alive and to stay together when it fights with the monster. We first present the off-line design of game agents, and then present the real-time planning and learning for each team.

8.4.4.1 The Offline Design for Adaptive and Transfer Learning

We use the open source 3D graphics engine OGRE to simulate the game scenario. The 3D modeled robot is used to represent the monster. And, there are two teams of game agents which are 3D models called *ninjas* and *jaiquas*. For the monster, game players can manipulate the monster in three statuses: Secure, Fighting and Ease. Game agents in these two teams react to the monster. Monster and game agents gain power when they are not involving in fighting. They both lose power during fighting. In particular, the power level that the monster loses depends on the fighting intensity (Intense/Moderate) of the game agents.

Game agents have different goals based on their own power level and the visibility of the monster. Figure 38 indicates a priority queue (Intention) when the monster is invisible to the game agent. On the other hand, when the monster is visible, a game agent has different actions base on the fighting status of the monster. Table 1 indicates three priority queues; each row is a priority queue. Furthermore, in the adaptive learning, team strategies are used to control the whole team of agents. In this example, we defined three strategies.

- *If more than half members of the team in low power, the whole team executes Escape*
- *If more than half members of the team in medium power, the whole team executes Moderate Fight*
- *If more than half members of the team in high power, the whole team executes Intense*

	High Power	Medium Power	Low Power
Monster is Secure	Intense Fight(2)	Intense Fight(2)	Moderate Fight (1)
Monster is Fight	Moderate Fight(2)	Escape (1)	Escape (1)
Monster is Ease	Intense Fight(2)	Intense Fight(2)	Moderate Fight (1)

Table 1: Three Intentions of a Game Agent when Monster is Visible

Fight

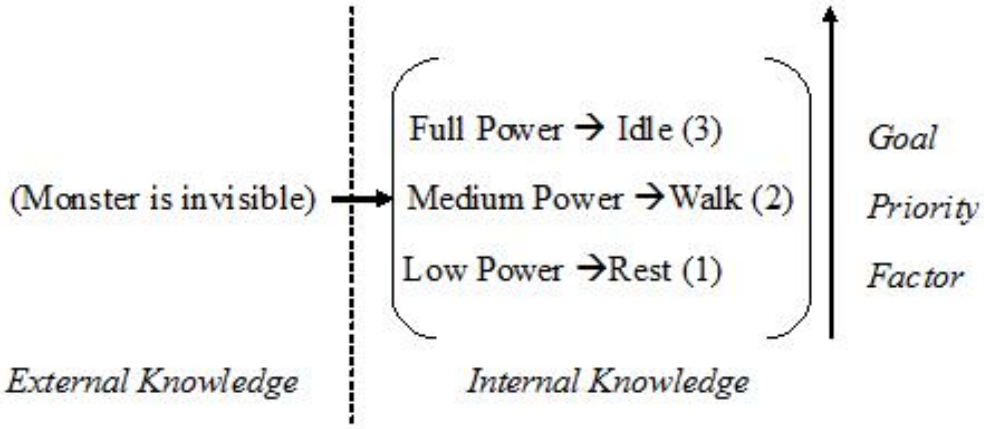

Figure 38: A Intention(Priority Queue) of "Monster is Invisible"

8.4.4.2 The Real-time Testing for Adaptive and Transfer Learning

In Figure 39 on page 110, we show the result of applying transfer learning to a battle between a monster and two teams. The game agents' performances are evaluated based on the monster's power. There are two curves in this figure; one is performance without transfer learning while the other is with transfer learning. In the no-transfer-learning case, a human player randomly controls the monster to fight with these two teams which is only in procedural planning status without transfer learning ability. We recorded the movement of the monster, and use it in the transfer-learning case. The blue curve indicates that after the experience accumulation (first ten game trials), the transfer-learning reduces the monster power more than the no-transfer-learning case.

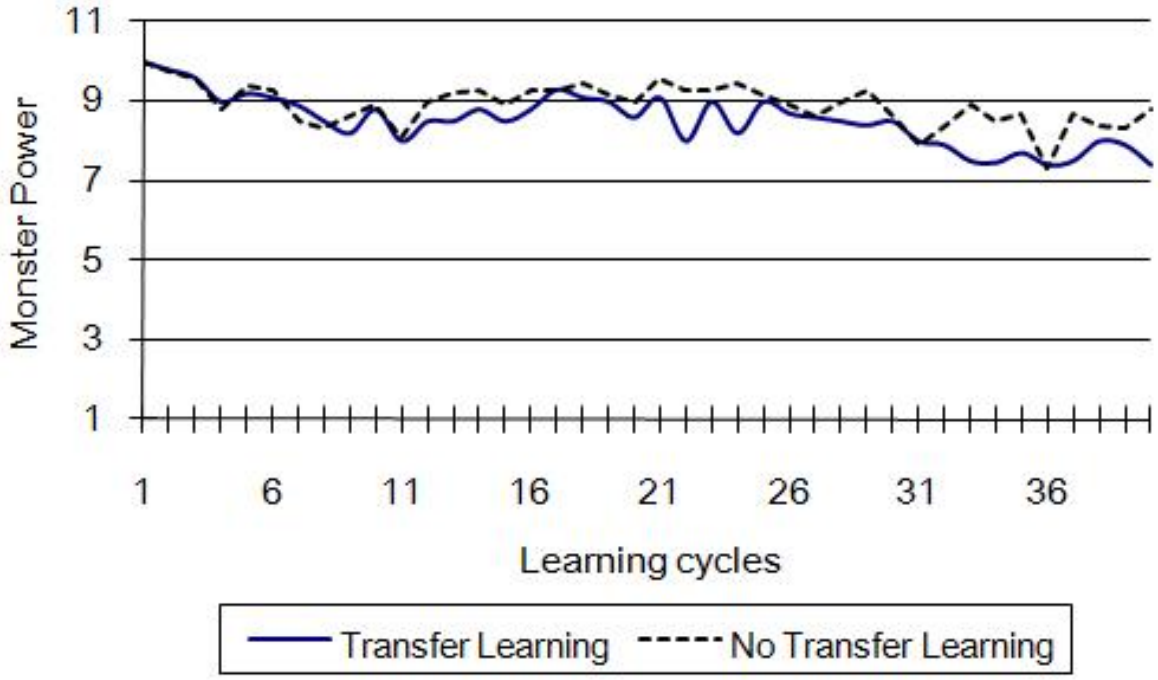

Figure 39: Transfer Learning and No Transfer Learning

In addition, we present the result of applying adaptive mechanism to these two teams in Figures 40 and 41 on page 111. Each team's overall reaction to the monster is also affected by those three team strategies defined in Section 8.4.4.1. When two members of the team jaiquas are in low power level, the team strategy making the whole team escape away from the monster even there is one jaiqua which is in medium power. The opposite situation happens with the team of Ninjas. They are all in high level, and so fight intensely.

Furthermore, the examples demonstrated in this section suggest a way of applying this multilayered agent processing system in game design. We point out that our approach provides a formal idea regarding a systematical game agents' procedural reasoning system. At the same time, the off-line game design still has very important impact on game agents' real-time performance. In order to have intelligent and autonomous game agents, we cannot underestimate either off-line design or real-time processing in games.

Figure 40: When the team of jaiquas has more than half member in low power, the whole team escape.

Figure 41: The team of ninjas is all in high power, they continue to fight with the robot intensely.

8.5 Summary

We present an approach of achieving team behaviour control by applying transfer and adaptive learning in real-time. Our idea of the planning and learning mechanism allows NPCs to react from game environment, learn from their experience and communicate with their opponent. Most importantly, this approach can reduce off-line NPCs' adaptability design. Our experiment demonstrates online transfer and adaptive learning in a scenario of a combat game. We show that our agents are capable of improving their team performance when they face unpredictable PCs.

Chapter 9

Emergent Learning

Intelligent goal-oriented game agents reasoning about their behaviours in relation to goals must consider not only the current situation but also previous and future situations. The players' actions spread throughout the whole game world, affecting not only the immediate target but nearby elements of the game world as well [Swe07]. The uncertainty of PCs' behaviours directly affect NPCs' behaviours. Intelligent game design requires NPCs to have flexibility in order to accommodate different game players. Emergent learning happens in the learning layer, as shown in Figure 8 on page 39. This learning technique can be the complementary learning ability to NPCs which intend to perform flexibility for different PCs. This chapter introduce an approach of emergent learning for game agents' control. Parts of this chapter were originally published as [SG09a].

9.1 Determined vs. Adaptive

We can distinguish between determined and adaptive systems by checking their input and output. Johns points out that a feature of determined systems is that the relationship between the inputs and the outputs is linear [Jon03]. However, complex adaptive systems (CASs) are not determined systems. CAS do not have to be complicated. The fascinating feature of CASs is "simple input, complex output".

The off-line game design is a determined design process which defines atomic or composite goals for game agents. It is a partially complete design which provides the basic knowledge for game agents. Game designers have to provide an explicit goal directed procedure for each goal in GOBTs. The capability of adaptation does not mean that game agents have to learn and adapt to everything from scratch. In complex situations, game agents cannot afford to learn every fact in a realistic environment. The knowledge design gives game agents some initial built-in knowledge which determines the basic reaction of game agents. The planning and learning processes are biased towards deterministic knowledge which is relevant to their goals.

For the purpose of presenting adaptive abilities, we can add some specific adaptive features in knowledge representation of game agents. These features include multi-dimensions, modularity and prioritization for the arrangement of goals. They serve as the static foundation for the adaptive learning mechanism in the layered architecture. The multidimensional goals refers to the tree structure of GOBTs. For each goal, there can be different paths (Plans) to achieve it. Also, the modularity feature of GOBTs adds the flexibility in goal combination and decomposition. In addition, the prioritization process in *intentions* allows agents to have a global scale for decision making in real-time.

In most cases, meticulous off-line knowledge design cannot guarantee satisfying real-time NPCs' intelligent adaptation to PCs. It is not easy to predict every possible situation, especially with different human players. Both adaptive features in off-line design and adaptive mechanisms in real-time agent control are important. In this chapter, we introduce a learning mechanism which has adaptive functionality for dynamically generating emergent behaviours for NPCs. As long as game agents have the ability of dynamic behaviour control, they are able to have adaptation ability for different PCs.

9.2 Introduction of the Emergent Learning

The emergent leaning means adjusting or combining agents' basic behaviours to exhibit adaptive strategies to the game player. Emergent learning for game agents is an adaptive

learning strategy that mainly focuses on the interaction between game agents and human players, and results in rational and acceptable unplanned behaviour of game agents. The emergent learning is used to enhance the replayability of games by *Behaviour Patterns*.

In general game play, human players play games by controlling PCs that interact with NPCs. The NPCs' action is followed by the PCs' action which decides the next states that the NPCs face. The uncertainty of PCs is the cause of the dynamic environment that NPCs should interact with. So, it is not reasonable to provide game agents with either a full model of different tasks or an explicit relation mapping from a source task to a target task as general transfer learning mechanisms do. For example, when a team of NPCs fight with a PC, since different human players have different tactics in manipulating the PC, the real-time transfer learning is much more adaptable than off-line training for NPCs.

Emergent leaning has features such as iterative, incremental and sub-optimal. It requires a number of revised iterations in order to achieve satisfactory reactions. After a period of incremental learning, NPCs' adaptabilities can have certain levels of enhancement. The emergent learning leads to the effectiveness of suboptimality for agents' performance. NPCs do not have to be perfect all the time. Within each iteration of learning, they only have to present their slight improvement with respect to their adversaries in the system. In addition, NPCs should have the ability to present different in-game strategies when they interact with different PCs. The goal of using the emergent learning is to provide NPCs capabilities of optimizing their behaviours when they face diverse human players.

The learning process of game agents has its own characteristics compared to conventional AI learning mechanisms. If game agents only have capabilities of action selection based on current PC's behaviour, their adaptability will be very restricted in real-time. However, designing a complicated learning process is not suitable for game agents. We have to be concerned about the real-time efficiency of behaviour control since the core AI module has to output game agents' next actions to the graphics and sound module for visual rendering. The emergent learning algorithm we discuss here considers *Behaviour Patterns* as a long term factor in learning algorithms, and selects the most suitable actions for NPCs.

9.3 Emergent Learning Processes

The emergent learning includes two interdependent processes, which are *behaviour pattern emergence* and *adaptive behaviour feedback*. They run in cycles, and improve NPCs' behaviour step by step. For example, when a human player starts to play a game, it is not easy to abstract the playing habits right away. Usually, it takes several rounds of playing to emerge PC's behaviour pattern. Then, the behaviour pattern can be fed back to the NPC to obtain the next behaviour. After several rounds of interactions, new behaviour patterns may be abstracted for new adaptive learning. The adaptive learning can be represented as the function

$$H : I_{planning} \times Be \times \omega \mapsto I_{learning}.$$

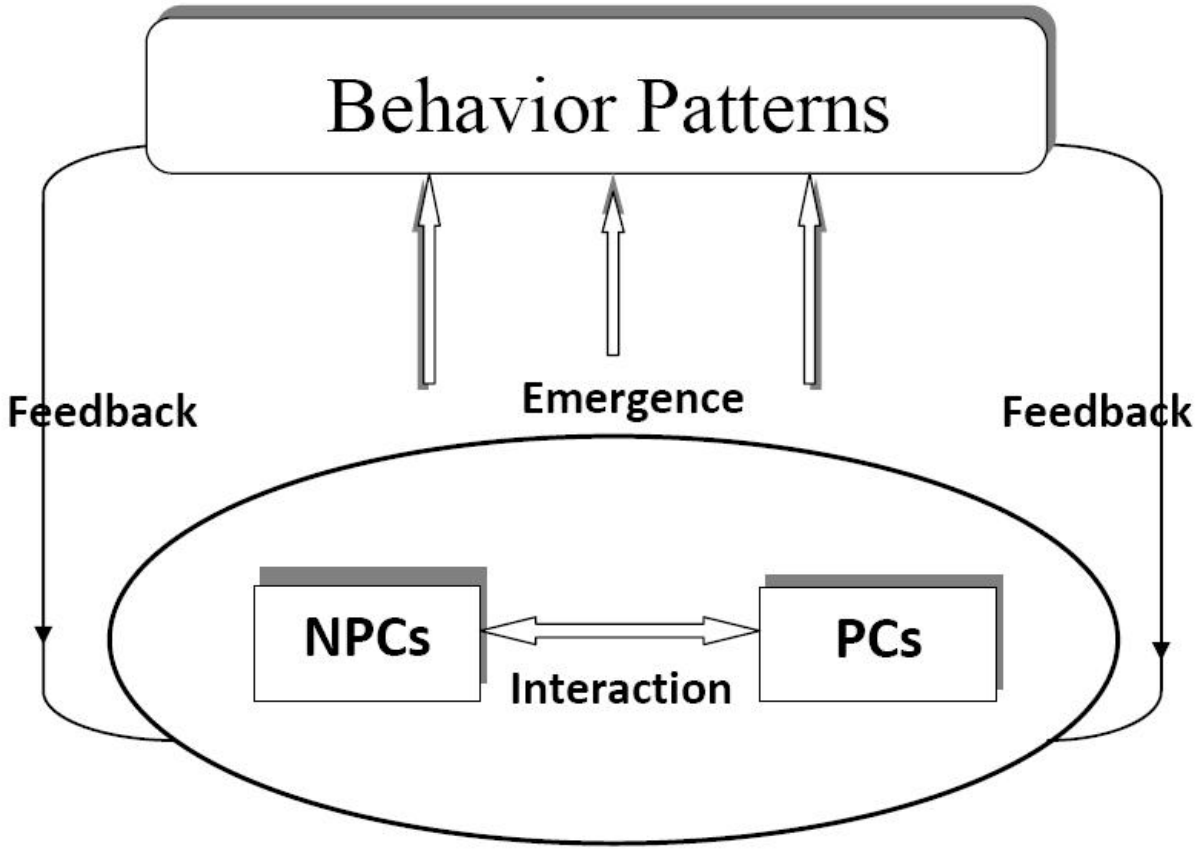

Figure 42: The Emergent Learning Mechanism

9.3.1 Behaviour Pattern

The *behaviour pattern* is the regularized model of a PC's behaviour. The pattern here focuses on characteristics of the ways in which game players can participate in the game. It describes features of how the game player manipulates the PC and thereby affects NPCs.

The pattern is abstracted on the basis of consecutive amounts of interactions between the PC and NPC. It is mutable. It can be updated over time since the PC does not always keep the same strategy all the time. In general, game players attempt to make some decisions regarding their future actions. They can learn from trial-and-error cases in certain rounds of interactions with NPCs, and change their further strategies. Because the PC's behaviour is regular and predicable, the pattern can reflect the playing style of the game player.

On the other hand, game designers cannot embed all possible PC's behaviour pattern into NPCs' knowledge. Since PC's strategies changes over time, it is possible to lessen the possibility for adaptive learning by introducing the behaviour pattern to NPCs, thereby giving NPCs adaptive learning abilities.

9.4 Behaviour Pattern Emergence

The operation of emergence is the process of abstracting and classifying behaviour patterns. For example, in Kung-Fu fighting games, the behaviour pattern is discerned from the most used action permutation of the game player. NPCs develop their fighting actions based on the pattern. They can then choose the most effective fighting reaction in terms of PC's behaviour pattern.

The potential for emergence is compounded when the elements of a system have some capacity for adaptation and learning [Swe07]. PCs and NPCs do not merely coexist in games; they communicate and interact with each other in a relationship of interdependent. Adaptive learning is the key aspect to present capacity to deal with unpredictable PCs' behaviour. We explore the possibilities of applying emergence in the interaction of PCs and NPCs.

The emergence phenomena follow a different set of dynamics. The PCs' behaviour

changes over time. However, in some specific games, it is possible to detect the regularity in game players' habits in controlling PCs. In detail, we design an emergent learning process for game agents' control. Figure 42 on page 116 shows the whole learning process. The PC's behaviour pattern emerges from interactions between PC and NPC in a number of continuous interaction cycles. The NPC learns a mapping from the PC's behaviour pattern to its next action, so that it follows the next action selection policy to optimize its performance overtime.

Behaviour patterns emerge from PC's real-time states in certain period. The PC has a total of n possible states S_i which are in the set S ($i \in [1, n]$ and $n > 1$). The PC selects one state S_i each time to interact with the NPC in real-time. In real-time, the S_i is the B_e which affect NPC's real-time action. The behaviour pattern is abstracted from real-time PC's states which contains m continuous states. We consider these states as a permutation σ written in one-line notation. The permutation σ is said to contain the behaviour pattern ω if there exits subsequence of entries of σ that has the same relative order as ω. The function E maps PC's actions (in consecutive times t) to a behaviour pattern.

$$E : \prod Be_t \mapsto \omega.$$

9.5 Behaviour Pattern Feedback

The *behaviour pattern* feedback is a process of mapping PC's state to corresponding NPC's next *intention* based on the behaviour pattern ω. As interactions between PC and NPC, the NPC have to develop a policy

$$\pi : B_e \xrightarrow{\omega} I_{learning},$$

which optimize the NPC's next *intention* for the adaptation purpose. At each time t, the NPC perceives the PC's state S_i (B_e) and its planning result $I_{planning}$ sent from the planning level. The NPC either chooses $I_{learning} \in \varpi$ based on the behaviour pattern ω or keeps $I_{planning}$ from the planning result.

In detail, a PC's behaviour pattern is $\omega = \{S_1, S_2, ..., Sk\}$, and the corresponding NPC's reactive *intentions* is $\varpi = \{I_1, I_2, ..., I_k\}$. The feedback is designed as these steps,

1. Accept the NPC's planning result $I_{planning}$.
2. Check whether the behaviour pattern ω is empty or not. If $\omega = \emptyset$, go to step 4 case 1.
3. If PC's current state $S_{current} = S_1$, schedule next $k-1$ intentions of the NPC as $\{I_2, I_3, ..., I_k\}$. Go to step 4 case 2.
4. case 1: Enqueue the next intention $I_{planning}$ the 3D engine rendering buffer;
   case 2: Enqueue intentions $\{I_2, I_3, ..., I_k\}$ to the 3D engine rendering buffer;

9.6 Testing for the Emergence Learning

In this section, we present an example of procedural planning and emergent learning process for integrations between a PC and a NPC. The PC is controlled by a human player; the NPC can learn from the PC's behaviours and present the adaptability to fight with the PC. Initially, the PC has a number of actions; each action has its own action ID, shown in Table 2. The human player can select one action or a combination of several actions to defeat the NPC. In particular, a combination of actions can have twice the attack power of a single action. The human player can define their own action combination, such as { Jump → HighKick → Kick }.

9.6.1 The Offline Design for Emergent Learning

ID	1	2	3	4	5	6
Name	Idle	Jump	Spin	SideKick	Kick	HighKick
Power	0	1	1	2	3	4

Table 2: PC's action and power level

PC's Action	NPC's Reaction		
	High Power	Medium Power	Low Power
Idle	Attack	Attack	Attack
Jump	HighKick	Attack	Block
Spin	Attack	Block	Stealth
SideKick	Kick	Jump	Escape
Kick	High Kick	Stealth	Escape
HighKick	Crouch	Stealth	Escape

Table 3: NPC's Reaction vs. PC's Action

The NPC has its predefined knowledge, such as reactions regarding the PC's different actions. Also, these reactions may be different depending on the NPC's power level. For example, in Table 3, if PC is "Idle", the NPC's intention is "Attack" when the NPC is in high power level. These rules give the planning process ideas about action selection in the planning level. Furthermore, with the emergence of NPC's behaviour pattern, the adaptive learning process can refine the NPC's reaction in order to have adaptability to this PC's fighting strategy. For example, if the learning process abstracts PC's behaviour pattern

$$\omega = \{\text{Jump} \rightarrow \text{HighKick} \rightarrow \text{Kick}\},$$

the NPC can schedule its next defense reactions based on this pattern, such as

$$\varpi = \{\text{HighKick} \rightarrow \text{Crouch} \rightarrow \text{HighKick}\}.$$

This learning result intends to provide the best solution to defeat the PC regardless the internal belief Bi (Power level) of the NPC. It is an emergent learning process which enhance the adaptive and dynamic features of NPCs.

9.6.2 The Real-time Testing for Emergent Learning

In Figure 43, we use the 3D Game engine OGRE to simulate the interactions between PC and NPC. There are two "Ninja" 3D models. One is controlled by a human player(PC) and another one is controlled by the emergent learning mechanism (NPC). In order to demonstrate the benefit of using emergent learning for the NPC, we test our approach with same

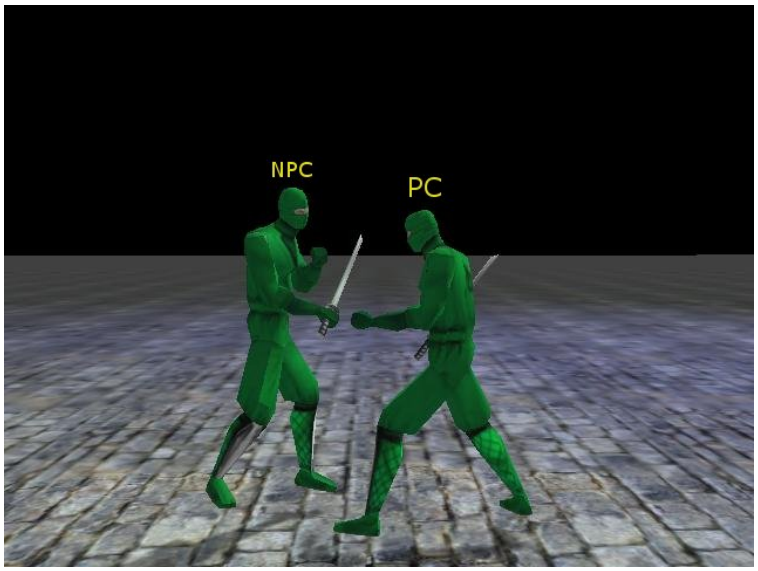

Figure 43: The Testing Case Rendering in ORGE3D

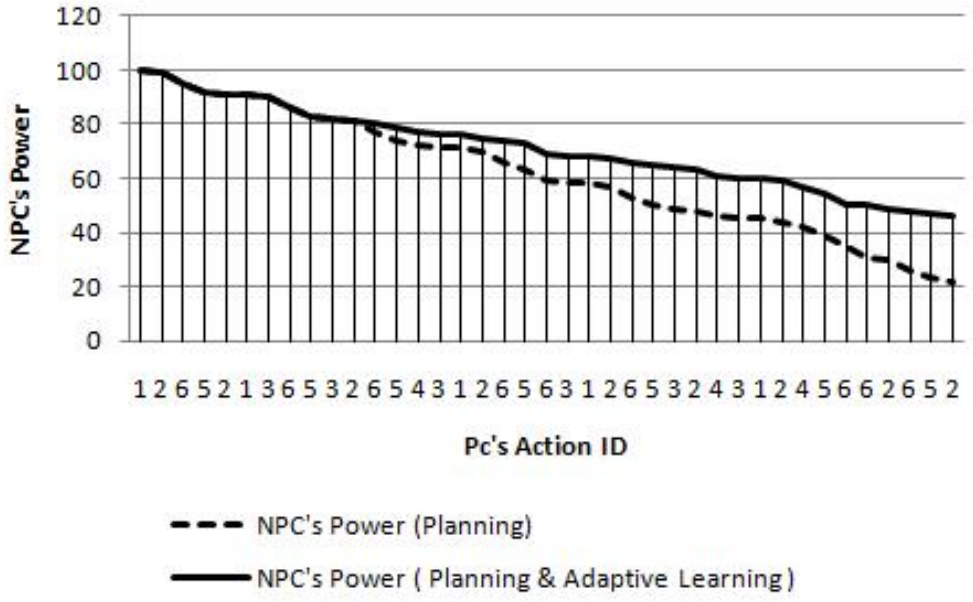

Figure 44: The NPC's Power Change Based on Planning or Planning&Emergent Learning

set of PC's actions. In the first case, the NPC only has the basic procedural planning ability; in the second case, the NPC has procedural planning and emergent learning abilities. Initially, the NPC's power level is 100. It was reduced as the PC keep attacking the NPC. The PC has a behaviour pattern { Jump → HighKick → Kick }. The testing results shown in Figure 44 indicate that the planning and emergent learning process makes the NPC lose less power than with the planning process alone.

9.7 Conclusion

Games have become more complicated in presenting dynamic game playing. Game designers intend to simulate reality worlds in game in order to attract more players. We cannot define games as fully complex systems since it is still simpler than CASs such as human immune systems. Also, game playing cannot be fully unpredictable or nonlinear as general CASs. We still have to set some restriction in games. However, if we can adopt some features from CASs to games, games can become more attractive in the way of presenting virtual reality worlds.

The emergent learning is a high-level behaviour control which consider long-term efficiency for game agents' performance. Our testing result shows that the emergent learning mechanism is suitable for real-time NPC's behaviour control. Emergent learning has huge potential use for game agents' behaviour control in various games. Especially, the algorithm of abstracting behaviour patterns can be different based on different game types. Also, game designers may have to make balance between adaptive and determined behaviour for NPCs. Predicable NPCs' behaviour is still an important aspect in the game play. At the same time, the off-line game design still has very important impact on game agents' real-time performance. In order to have intelligent and autonomous game agents, we cannot underestimate the effectiveness of either off-line design or real-time processing in games.

Chapter 10

Conclusions and Future Work

In this thesis, we have presented the design for a game development system—Gameme. This chapter consists of two main sections, summarizing the main contributions and open directions for future research from two different perspectives: that of the design for Gameme and that of game AI. Within each section, our contributions will be discussed in the context of a number of open research topics. A final section summarizes and discuss future research directions.

10.1 The Gameme Design Perspective

The development of Gameme is a project that includes building different modules from different computer science subjects. For example, the AI core module is based on artificial intelligent theories; the Graphics & Sound module is composed of multimedia engines; the UI module is related to user interface design technologies. We prototype relationships among modules in Chapter 3. These modules could be developed and tested separately, and combined as needed. In experiments that we presented in this thesis, the core AI module is connected with the ORGE 3D engine for the purpose of providing visual game scenarios.

We focus mainly on the development of the core AI module in Gameme since it is the logical control module in Gameme. We designed and customized AI methodologies to build

the AI core module. Goal oriented is the common feature of most game agents. So, GOBD becomes our main idea in describing game agents' behaviours. GOBT is the data structure we designed in order to characterize goals for game agents. Also, game agents are not independent of their environment; they have to interact with other in-game entities. We extend the GOBD and GOBT to model game agents as BDI-like agents, and give them the basic sense-reaction abilities. After design the off-line game agents' modeling, we embedded planning and learning mechanisms in a layered architecture for real-time game agents' behaviour control. The planning mechanism is about creating game agents' basic personality. And, the transfer, adaptive and emergent learning mechanisms provide advanced behaviour control for game agents. The real-time agents' behaviour processing extends the original procedural knowledge processing from simple planning to advanced learning. We test planning and learning mechanisms in fighting scenarios. NPCs could cooperate as a team, or learn from their previous experience in order to have better strategies to fight with PCs. Furthermore, NPCs could analyse different PCs' behaviours, and find the best action combinations to defeat PCs.

10.2 The Game AI Perspective

We develop the core AI module in Gameme as our research project. However, all methodologies in Gameme can still be separately applied in computer games development. The four potential contributions to game AI that can be found in this thesis are: the methods for goal oriented design; procedural planning; transfer and adaptive learning; and emergent learning.

- We have combined goal oriented design and procedural knowledge representation to model game agents. In game AI, the GOBD can not only apply in describing game agents' behaviours, but also use in game story telling. For example, in adventure games, game players have to fulfill different in-game task in order to proceed. These tasks could also be describe by as goal oriented procedures. Game designers could benefit from the intuitive and clear design methodology of GOBD.

- Procedural planning enables game agents to have basic autonomous characteristics. The planning mechanism has the feature of fast reaction to the environment since it selects the next action from off-line design knowledge instead of generating new action. The planning outcome might not be suitable for the creation of intelligent game agents. However, there is still demand for game agents with simple behaviours in games. Using this planning mechanism can reduce the time spent in logical control of game agents, and leave more system resources to be used by multimedia rendering.

- We provided a framework for transfer and adaptive learning for team cooperation in this thesis. The key for transfer learning is to set up reasonable source knowledge, and apply it to the target knowledge. In our experiments, *experience* is accumulated and selected as source knowledge. The adaptive learning provides a coordination over a team instead of individuals in the team. The transfer and adaptive learning has huge potential in game AI. Game developers could use these learning idea by choosing different source knowledge or target beneficiaries.

- Emergence is a fascinating phenomena which attracts game designers' attention. The emergent learning we discuss in this thesis is one of the aspects that emergent learning could devote to games. We use emergent learning to let NPCs have flexible behaviours when they face various PCs. *Emergence* and *feedback* are two important steps in this learning. NPCs could have the ability to use or combine their basic actions to exhibit new behaviours. For the application of emergence is not limited to the behaviour control over game agents. Game designers could extend the emergence effects to other areas, such as story line, narrative and physics interactions, in games.

10.3 Future Work

Our future work would be the continuing development of Gameme. Gameme is an open-ended research project. There are many aspects that we could add to Gameme. In addition, we could build new theories and experiment with them in Gameme for the purpose

of proposing new concepts into other game development. Here, we present a short term research proposal for Gameme.

- For the core AI module, more AI methodologies will be added into this module. Indeed, there are lots of new AI features that we could add into this module. We plan to provide Gameme with the ability to build particular types of games, such as fighting games. By using the same idea regarding goal oriented design, we would like to have a methodology to describe story lines in games. Then, more multi-agent control theory would be added to control NPCs. In addition, we would like to extend our learning mechanisms to support PCs. Games created by Gameme could provide certain information to help PCs in battle with NPCs.

- For the UI module, an intuitive user interface would be added to Gameme. Our approach is to have training processes as simple as possible for Gameme's users. The interface should be simple and clear. Non-professional game designers could master the UI in a very short time.

- For the Graphics & Sound module, we would like to import more 3D models along with their animation data to Gameme. Currently, our visual testing cases are limited to a couple of 3D models provided by the ORGE 3D engine. With more 3D models, we could have more in-game characters to be chosen by game designers.

Bibliography

[ABB+04] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *PSYCHOLOGICAL REVIEW*, 111:1036–1060, 2004.

[Ber02] Lee Berger. Scripting: Overview and code generation. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 505–510. Charles River Media, 2002.

[Ber06] Bryan Bergeron. *Developing Serious Games*. Charles River Media, 2006.

[Bjo07] Dines Bjorner. *Software Engineering 3: Domains, Requirements, and Software Design*, page 106. Springer Verlag, 2007.

[Bra99] Michael E. Bratman. *Intention, Plans, and Practical Reason*. CSLI Publications, 1999.

[Bro91] Rodney A. Brooks. Intelligence without reason. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595. Morgan Kaufmann, 1991.

[Bry01] Joanna Joy Bryson. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, Massachusetts Institute of Technology, 2001.

[Bry03] Joanna Joy Bryson. The behavior-oriented design of modular agent intelligence. *Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, 2003.

[BS04] David M. Bourg and Glenn Seemann. *AI for Game Developers*. O'Reilly, first edition, July 2004.

[BS07] Bikramjit Banerjee and Peter Stone. General game learning using knowledge transfer. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 672–677, January 2007.

[Buc05] Mat Buckland. *Programming Game AI by Example*. Wordware Publishing, 2005.

[Cha07] Alex J. Champandard. 10 reasons the age of finite state machines is over. Game AI For Developers
(`http://aigamedev.com/`), 2007.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communication of the ACM*, 19(3):547–554, 1976.

[CNM83] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc, 1983.

[CSS+06] Maria Cutumisu, Duane Szafron, Jonathan Schaeffer, Matthew McNaughton, Thomas Roy, Curtis Onuczko, and Mike Carbonaro. Generating ambient behaviors in computer role-playing games. *IEEE Intelligent Systems*, 21:19–27, 2006.

[dB04] Penny Baillie de Byl. *Programming Believable Characters for Computer Games*. Charles River Media, 2004.

[dKLW97] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV. Lecture Notes in Artificial Intelligence*, volume 1365, pages 155–176. Springer Verlag, 1997.

[Dro07] R. Geoff Dromey. Principles for engineering large-scale software-intensive systems. Invited talk for ASWEC, 2007.

[DSAS03] Pat Langley Daniel, Daniel Shapiro, Meg Aycinena, and Michael Siliski. A value-driven architecture for intelligent behavior. In *Proceedings of the IJCAI-2003 Workshop on Cognitive Modeling of Agents and Multi-Agent Interactions*, 2003.

[Dyb03] Eric Dybsand. AI middleware: Getting into character. part 1 biographic technologies' ai.implant.
`http://www.gamasutra.com/`, July 2003.

[Eng07] Andries P. Engelbrecht. *Computational Intelligence An Introduction*, page 83. John Wiley & Sons, Ltd, 2007.

[FG96] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the third International Workshop on Agent Theories,Architectures and Languages*, pages 21–35. Springer-Verlag, 1996.

[FL02] Kenneth D. Forbus and John Laird. AI and the entertainment industry. *IEEE Intelligent Systems*, 17:15–16, July 2002.

[FP99] Gerhard Fischer and Leysia Palen. Design = the sciences of the artificial. Center for Lifelong Learning and Design, University of Colorado, Boulder,
`http://l3d.cs.colorado.edu/courses/csci7212-99/pdf/science-of-the-art.pdf\`, 1999.

[GI90] M. P. Georgeff and F. F. Ingrand. Real-time reasoning: the monitoring and control of spacecraft systems. In *Proceedings of the Sixth Conference on Artificial Intelligence Applications*, pages 198–204. IEEE Press, 1990.

[GL87] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *AAAI*, pages 677–682, 1987.

[Gou88] Ronald J. Gould. *Graph Theory*. Benjamin/Cummings Publishing Inc., 1988.

[Hol95] John H. Holland. *Hidden Order: How Adaptation Builds Complexity*, pages 5,7. Addison-Wesley Publishing Company, 1995.

[Hol00] John H. Holland. *Emergence: From Chaos To Order*, page 2. Oxford University Press, 2000.

[Hor00] Ian Douglas Horswill. Functional programming of behavior-based systems. *Autonomous Robots*, 9:83–93, 2000.

[HR85] Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28:921–932, 1985.

[Joh01] Steven Johnson. *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*. Scribner, 2001.

[Jon03] Wendell Jones. Complex adaptive systems. Beyond Intractability, Eds.Guy Burgess and Heidi Burgess, Conflict Research Consortium, University of Colorado, Boulder, `http://www.beyondintractability.org/essay/complex_adaptive_systems/`, 2003.

[Jon08] M. Tim Jones. *Artificial Intelligence: A System Approach*. Infinity Science Press LLC, 2008.

[JW01] D. Johnson and J. Wiles. Computer games with intelligence. In *Proceedings of the 10th IEEE International Conference on Fuzzy Systems,*, pages 1355–1358, 2001.

[Koz92] John R. Koza. *Genetic Programming I : On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Koz07] John R. Koza. Introduction to genetic programming. In *Proceedings of Genetic and Evolutionary Computation Conference(Companion)*, pages 3323–3365, 2007.

[LK05] Manfred Lau and James J. Kuffner. Behavior planning for character animation. In *Eurographics ACM SIGGRAPH Symposium on Computer Animation*, 2005.

[Lou08] Ronald Loui. In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41, July 2008.

[LRN86] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1(1), November 1986.

[LRN87] John E. Laird, Paul S. Rosenbloom, and Allen Newell. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33, September 1987.

[LvL01] John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. *AI Magazine*, 22(2):15–26, 2001.

[Mae94] Pattie Maes. Modeling adaptive autonomous agents. *Artificial Life*, 1:135–162, 1994.

[Mil06] Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.

[MPG87] Marcel J. Schoppers Michael P. Georgeff, Amy L. Lansky. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical report, Technical Note 380, Artificial Intelligence Center, SRI International, 1987.

[MS01] Ivan Moscovich and Ian Stewart. *1000 Play Thinks: Puzzles, Paradoxes, Illusions and Games*. Workman Publishing Company, 2001.

[Mye01] K. L. Myers. Procedural reasoning system user's guide. Technical report, Artificial Intelligence Center, Technical Report, SRI International, 2001.

[Nar04] Alexander Nareyek. Artificial intelligence in computer games. *Game Development*, 1(10), February 2004.

[Nil01] N. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electronics Transactions in Artificial Intelligence*, pages 99–110, 2001.

[Nwa96] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11:205–244, 1996.

[Ode00] James Odell. Agents (part 2): Complex systems. *Executive Report, Cutter Consortium*, 3(6), 2000.

[OR08] Santi Ontanon and Ashwin Ram. Adaptive computer games: Easing the authorial burden. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 617–631. Charles River Media, 2008.

[Ork05] Jeff Orkin. Agent architecture considerations for real-time planning in games. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2005.

[Pil04] Nelishia Pillay. A first course in genetic programming. *The SIGCSE Bulletin*, 36(4), 2004.

[Poi02] Falko Poiker. Creating scripting languages for nonprogrammers. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 520–529. Charles River Media, 2002.

[PS92] David N. Perkins and Gavriel Salomon. Transfer of learning. In *International Encyclopedia of Education*. Pergamon Press, 1992.

[PSSTT06] David Pardoe, Peter Stone, Maytal Saar-Tsechansky, and Kerem Tomak. Adaptive mechanism design: A metalearning approach. In *The Eighth International Conference on Electronic Commerce*, pages 92–102, August 2006.

[RB96] Christopher D. Rosin and Richard K. Belew. A competitive approach to game learning. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, pages 292–302. ACM Press, 1996.

[RG91a] Anand S. Rao and Michael P. Georgeff. Intelligent real-time network management. Technical report, Australian Artificial Intelligence Institute. Technical Note 15, 1991.

[RG91b] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture, 1991.

[RG95] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multi-agent Systems*, pages 312–319, 1995.

[RM02] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers—a survey. *IEEE Transactions on System, Man and Cybernetics: Part C*, 1(11), November 2002.

[Roc99] Luis M Rocha. Complex systems modeling: Using metaphors from nature in simulation and scientific models. Technical report, Los Alamos National Laboratory, Novemeber 1999.

[SA04] Matthias Scheutz and Virgil Andronache. The apoc framework for the comparison and evaluation of agent architecture. In *Proceedings of AAAI Workshop on Intelligent Agent architecture*, pages 66–73. AAAI Press, 2004.

[San03] Thomas Sandholm. Making markets and democracy work: A story of incentives and computing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1649–1671, 2003.

[Sch04] Brian Schwab. *AI Game Engineering Programming*. Thomson Delmar Learning, 2004.

[SG08a] Yingying She and Peter Grogono. Goal oriented behavior trees: A new strategy for controlling agents in games. In *Proceedings of the 4th International North*

American Conference on Intelligent Games and Simulation, pages 108–112. EUROSIS, August 2008.

[SG08b] Yingying She and Peter Grogono. The procedural planning system used in the agent architecture of games. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pages 256–257. ACM, November 2008.

[SG09a] Yingying She and Peter Grogono. The adaptive learning mechanism design for game agents's real-time behavior control. In *Proceedings 2009 IEEE International Conference on Intelligent Computing and Intelligent Systems*, pages 792–795. IEEE Press, November 2009.

[SG09b] Yingying She and Peter Grogono. An approach of real-time team behavior control in games. In *Proceedings of the 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 546–550. IEEE Computer Society, November 2009.

[SG09c] Yingying She and Peter Grogono. A procedural planning system for goal oriented agents in games. In *Advanced in Artificial Intelligence, 22nd Canadian Conference on Artificial Intelligence*, pages 245–248. Springer, May 2009.

[SG09d] Yingying She and Peter Grogono. A real-time transfer and adaptive learning approach for game agents in a layered architecture. In *Lecture Notes in Computer Science, Proceedings of the 9th International Conference on Intelligent Virtual Agents*, pages 545–546. Springer-Verlag, September 2009.

[SHS+07] Manu Sharma, Michael Holmes, Juan Santamaria, Arya Irani, Charles Isbell, and Ashwin Ram. Transfer learning in real-time strategy games using hybrid CBR/RL. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1041–1046, January 2007.

[SL98] Aaron Sloman and Brian Logan. Architectures and tools for human-like agents. In *Proceedings of the Second European Conference on Cognitive Modeling*, pages 58–65. University Press, 1998.

[Sow00] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pacific Grove, 2000.

[SP06] Ingo Schnabel and Markus Pizka. Goal-driven software development. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 59–65. IEEE Computer Society, 2006.

[Ste94] Luc Steels. A case study in the behavior-oriented design of autonomous agents. In *Proceedings of the third international conference on Simulation of adaptive behavior : from animals to animats 3*, pages 445–452. MIT Press, 1994.

[Sto07] Peter Stone. Learning and multiagent reasoning for autonomous agents. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 13–30, January 2007.

[Sut88] R. S. Sutton. Learning to predict by the method of temporal difference. *Machine Learning*, 3:9–44, 1988.

[SV98] Peter Stone and Manuela Veloso. Using decision tree confidence factors for multiagent control. In *RoboCup-97: Robot Soccer World Cup I*, pages 99–111. Springer Verlag, 1998.

[Swe07] Penny Sweetser. *Emergence in Games*, pages 19, 36, 39, 40, 79, 132. Charles River Media, 2007.

[SZ03] Katie Salen and Eric Zimmerman. *Rules of Play, Game Design Fundamentals*. The MIT Press, 2003.

[Szi07] Nicolas Szilas. An implementation of real-time 3D interactive drama. *ACM Computers in Entertainment*, April 2007.

[TA98] E. Turban and J. Aronson. *Decision Support Systems and Intelligent Systems*. Prentice Hall, 1998.

[Tet87] Linda Tetzlaff. Transfer of learning: Beyond common elements. *ACM SIGCHI Bulletin*, 17:205–210, 1987.

[TKS08] Matthew E. Taylor, Gregory Kuhlmann, and Peter Stone. Autonomous transfer for reinforcement learning. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 283–290, 2008.

[Tog07] Julian Togelius. *Optimization,Imitation and Innovation: Computational Intelligence and Games*. PhD thesis, University of Essex, September 2007.

[Toz02a] Paul Tozour. How not to implement a basic scripting language. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 548–554. Charles River Media, 2002.

[Toz02b] Paul Tozour. The perils of ai scripting. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 541–547. Charles River Media, 2002.

[VCP+95] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7:81–120, 1995.