Current AI in Games: A Review

Abstract – As the graphics race subsides and gamers grow weary of predictable and deterministic game characters, game developers must put aside their "old faithful" finite state machines and look to more advanced techniques that give the users the gaming experience they crave. The next industry breakthrough will be with characters that behave realistically and that can learn and adapt, rather than more polygons, higher resolution textures and more frames-per-second. This paper explores the various artificial intelligence techniques that are currently being used by game developers, as well as techniques that are new to the industry. The techniques covered in this paper are finite state machines, scripting, agents, flocking, fuzzy logic and fuzzy state machines decision trees, neural networks, genetic algorithms and extensible AI. This paper introduces each of these technique, explains how they can be applied to games and how commercial games are currently making use of them. Finally, the effectiveness of these techniques and their future role in the industry are evaluated.

Keywords **– computer games, artificial intelligence**

Introduction

There are many different artificial intelligence (AI) techniques in use in modern computer games. The most prevalent techniques include finite state machines, scripting, agents and flocking. These techniques are well-established, simple and have been successfully employed by game developers for a number of years. Additionally, the use of fuzzy logic and fuzzy state machines as an alternative to finite state machines is starting to become widely accepted and commonplace, as is the addition of extensible AI for games. Finally, there are a few game developers that are venturing out to try new and interesting techniques, which have not been possible until recently, due to processor constraints. These techniques include decision trees, neural networks and genetic algorithms. This review will explore each of these techniques and their applications in industry and in computer games. Each technique will be introduced, the possibilities and boundaries of use of the technique in industry and games will be given and the use of the technique in current games will be explained. Additionally, an appendix is included that provides a summary table of the advantages, disadvantages and applications of these techniques.

Finite State Machines

Finite State Machines (FSMs) are used more frequently in computer games than any other AI technique. This is because they are simple to program, easy to understand and debug, and general enough to be used for any problem [37]. The idea of an FSM is to divide a game object's behaviour into logical states so that the object has one state for each different type of behaviour it exhibits [36]. An FSM can be any system that has a limited number of states

of operation. An FSM may not always provide the optimal solution, but it generally provides a simple solution that works. Furthermore, a game object that uses an FSM can also use other techniques such as neural networks or fuzzy logic [36].

In general, game AI focuses on creating the appearance of intelligence [21]. In many games it comes down to what the player perceives and whether they are convinced that the AI is behaving reasonably. Often, the use of more advanced algorithms and techniques is not possible, due to computation or other constraints, and in these circumstances a simple solution such as an FSM is desirable. Furthermore, if the simplest technique works for the problem, then the use of advanced techniques is not necessary, especially if it won't give better results.

Some problems with using FSMs are that they tend to be poorly structured with poor scaling, so that they increase in size uncontrollably as the development cycle progresses. These properties tend to make FSM maintenance very difficult. Furthermore, FSMs in games tend to include states within states, multiple state variables, randomness in state transitions and code executing every game tick within a state [37]. Consequently, game FSMs that are not well planned and structured can grow out-of-hand quickly and become very challenging to maintain.

When making an FSM for a game, the developer needs to anticipate, plan and test the elements on which the player's attention might possibly be focused. The more the developer can anticipate, the more immersive the environment will be for the player. For game AI, the possible ways in which to use an FSM are endless. It could be used to manage the game world or maintain the status of the game or game object. An example is modelling unit behaviour in a real-time strategy game. Alternatively, it could be used to parse input from the human player or even to simulate the emotion of a non-player character [13]. For example, an FSM could be used to represent a monster with emotional states such as berserk, rage, mad, annoyed and uncaring. In each of these states, the game AI would do something different to reflect the monster's changing attitude. In this case, an FSM would be used to manage the monster's attitude and the transitions between states based on the input from the game. Different inputs in this example could include information about the player's actions, such as whether they have come into view of the monster, attacked the monster or run away. Also, information about the monster would also be important, such as whether the monster has been hurt or healed. These variables form the input to the FSM and based on the input values and the monster's current attitude, the monster's attitude will change, or transition, to another state.

FSMs are used in most commercial computer games, some examples are Age of Empires, Enemy Nations, Half-Life, Doom and Quake. Specifically, Quake 2 uses nine different states for each character. These states are standing, walking, running, dodging, attacking, melee, seeing the enemy, idle and searching. In order to form an action, these states may be connected together. For example, in order to attack the player, the states could first

go from ‘idle’ to ‘run’ to allow the attacker to get closer to the player, then switch to ‘attack’ [17].

FSMs are by far the most popular type of artificial intelligence in modern games. This is due to how easily FSMs can be understood and programmed [33]. FSMs are amongst the simplest computational devices. Also, they have a low computational overhead and can be used as core modules of agents. Most importantly, they give a large amount of power relative to their complexity [10]. These attributes make FSMs ideal for the conditions of game AI development, which involves limited computational resources and, as the AI is usually one of the last things to be implemented, limited development and testing time. It will be a long time before game developers abandon the FSM in search of other, more advanced techniques. Most likely, this transition will include an FSM backbone with neural nets or fuzzy logic for specialised components of the AI.

Agents

Intelligent agents are software agents that perceive their environment and act in that environment in pursuit of their goals. Examples of intelligent agents include autonomous robots in a physical environment, software agents with the internet as their environment or synthetic characters in computer games and entertainment [28]. Agents usually integrate a range of competences, such as goals, reactive behaviour, emotional states and the consequent behaviours, natural language, memory and inference. Agents are central to the study of many problems in AI, such as modelling human mental capabilities and performing complex tasks [28].

Games are ideal environments for agents as they provide realistic environments in which only limited information is available and where decisions must be made under time and pressure constraints [28]. Generally, agents in games are sets of FSMs that work on their particular problems and send messages to each other. Alternatively, an agent could be a set of fuzzy state machines, neural networks, genetic algorithms or any combination of some or all of these techniques.

Important decisions that need to be made when designing an agent is the architecture and whether the agent is to be reactive, goal-directed or some combination of the two. A purely reactive agent is suited to highly dynamic environments where little information about previous actions and states is necessary. At the other extreme, a purely goal-directed agent is suited to a static environment where planning and considering previous moves are highly desirable. For example, a monster in a first-person shooter or a role-playing game would be more suited to simply reacting to what is currently happening in the game. However, an agent governing the strategy for the AI in a strategy game needs to carefully plan its moves depending on what has happened so far in the game.

A good architecture for a real-time strategy agent is necessary to ensure success. For example, in the game Empire Earth, the AI consists of several components called managers. Each manager is responsible for a specific area

in the management of the computer player. In Empire Earth, there are managers for the civilisation, building, units, resources, research and combat [40]. The civilisation manager is the highest level manager and is responsible for the development of the computer player's economy and the coordination between the other managers. The other managers have lower-level duties and send requests and reports to each other. This forms a well-structured agent that provides for maintenance and extensibility.

In summary, an agent's job in a computer game is to make decisions and perform tasks to achieve some set of goals, as does a human player. Every game that includes AI can be said to be using an agent of some form. However, the important question is whether the agents are well designed and structured, or put together ad hoc as development progresses. The latter case is true in most games that build FSM components as necessary, without any real prior planning. Unfortunately, these agents do not provide the power, extensibility and maintainability of a well-structured agent, such as the one used in Empire Earth. An agent that is structured and decomposed into communicating layers or modules has a large advantage over any agent that consists of one or two enormous state machines that grow uncontrollably throughout the development process.

Scripting

A scripting language is any programming language created to simplify any complex task for a particular program. It is a fourth generation language that is used to control the game engine from the outside. The scope of a scripting language can vary significantly depending on the problems it is designed to solve, ranging from a simple configuration script to a complete runtime interpreted language [35].

Scripting languages for games, such as Quake's QuakeC or Unreal's UnrealScript, allow game code to be programmed in a high-level, English-like language [25]. They are designed to simplify some set of tasks for a program and hide many complicated aspects of a game [5], thus allowing non-programmers, such as designers and artists, to write script for the game. During development, the designers use scripting to implement stories [35], while artists use scripting to automate repetitious tasks, do things that the computer can do better than humans and add new functionality [41]. After the game is shipped, mod groups and hobbyists write scripts if the scripting system has been exposed to the public [35]. However, as with FSMs, scripting languages are deterministic and they require the game developer to hard-code character behaviour and game scenarios. Therefore, the developer must anticipate and hard-code each of the situations the player might be in.

The uses of scripting languages in games vary from simple configuration files to entirely script-driven game engines. The common uses include creating events and opponent AI for the single-player mode of the game. Also, in single-player mode, they can be used to tell the story of the game and control the player's enemies [35]. A first-person shooter game could use scripting to create a monster's AI. Alternatively, a real-time strategy game might use scripting to define how spells function or to define a quest

or part of the game story. Also, scripting can be a very powerful tool in massively multiplayer online games (MMOGs). In MMOGs, scripting can be used for hiding the details of dealing with multiple servers in a server farm, simplifying sending network events to a client. Also, scripting languages can handle saving an object's state automatically. In role-playing games scripting can be used to define simple conversation trees for a non-player character. Finally, a scripting language could even be a complicated object-oriented language that controls every aspects of gameplay [5].

Many commercial games have used scripting for some, if not all, of the game AI. Games that have successfully used scripting, whether it was a custom-made scripting language or an off-the-shelf language, include Black & White, Unreal, Dark Reign and most of the games developed by BioWare.

The game Black & White used a custom scripting language to present the game's storyline through a set of 'Challenges', which served to advance the storyline, give the player an opportunity to practice their skills, and entertain the player [3]. The Challenge language was developed to implement the logic and cinematic sequences for the Challenges and allowed the game developers to experiment independently of the programmers. Also, as the script was independent of the data structures and game code, it was less likely that a bug in the script would cause the game to crash [3].

The games developed by Bioware using their Infinity Engine, including Baldur's Gate, Baldur's Gate II, Planescape: Torment and Icewind Dale, all used a custom scripting language called BGScript. BGScript implemented a very simple syntax in which the scripts consisted of stacked if/then blocks with no nesting, loops or other complicated structures. It was designed fundamentally as a simple combat scripting language. However, it was also used for simple, noncombat creature scripting, trap and trigger scripting, conversation and in-game movies [6]. In Baldur's Gate, players are able to directly edit scripts that control the actions of their non-player characters. All non-player characters have their own AI scripting, outlining their basic reactions to basic situations and at anytime the player can override what the non-player character is currently doing [45]. Bioware's most recent game, Neverwinter Nights, used a scripting language call NWScript. NWScript was designed to include the features from BGScript, as well as spells and pathfinding around doors. Both BGScript and NWScript were designed to be used by the end user. Also, Bioware's game MDK2 and the LucasArts game Escape From Monkey Island both used the Lua scripting language, which was heavily modified by the game developers to give the desired behaviour [6].

Scripting, similar to FSMs, is a favourite tool of game developers and will be a part of game development for a long time to come. Scripting languages are ideal for games as they are suitable for non-programmers, such as designers, artists and end users. Therefore, the designers and artists can implement sections of the game

independently of the game programmers and that end users can make their own mods for the game. Also, scripting languages are generally separate from the game's data structures and codebase and thus provide a safe environment for non-programmers and end users to make changes to the game, so that bugs in the script will not cause the game to crash. Many commercial games use scripting to some degree and most developers report success when they customise their own scripting tools.

Fuzzy Logic

Fuzzy Logic is unlike traditional Boolean logic in that it allows intermediate values to be defined between conventional values such as yes/no or true/false [4]. Consequently, "fuzzy" values such as 'rather hot' or 'very fast' that are used to describe continuous, overlapping states can be used in an exact mathematical way [29].

The benefit of Fuzzy logic is that decisions can be made based on incomplete or erroneous data that cannot be used in Boolean logic [25]. The power of fuzzy logic lies in the ability to represent a concept using a small number of fuzzy values [2], whereas in Boolean logic every state and transition needs to be hard coded. Fuzzy logic can be applied to the areas of decision making, behavioural selections and input/output filtering [25] and has been used in tools for controlling subway systems, industrial processes, household and entertainment electronics and diagnostic systems [29].

Fuzzy logic is applicable when there is no simple mathematical model that can solve the problem, when the processing of expert knowledge is required and for highly nonlinear problems. However, fuzzy logic is not ideal when conventional methods yield a satisfying result, when there is an existing mathematical model that already solves the problem or when the problem is not solvable [4]**.** In short, if there is already a simple solution that satisfies a problem then there is no need to complicate things. However, if the problem is non-linear or there is no simple solution, then fuzzy logic may be appropriate.

According to Zarozinski [47], fuzzy logic makes its way into most computer games. However, its role in games usually doesn't exceed complex *if-then-else* statements due to the complexity of creating a fuzzy logic system from scratch. A game AI engine can use fuzzy logic to fuzzify input from the game world, use fuzzy rules to make a decision and output fuzzy or crisp values to the game object being controlled [25]. Fuzzy logic can prove especially useful in decision-making and behaviour selection in game systems [1]. Also, it can be used for AI opponents to determine how frightened they are of the player, for non-player characters to decide how much they like the player, for flocking algorithms to see how close together the flock should stay or even for events such as how the clouds would move given the wind speed and direction [32].

Commercial computer games that have made use of this technology include BattleCruiser: 3000AD, Platoon Leader and SWAT 2. BattleCruiser: 3000AD, developed

by Derek Smart, mostly uses neural networks to control the non-player characters in the game. However, in situations where neural networks are not applicable, it uses fuzzy logic. Also, the game SWAT 2, developed by Yosemite Entertainment, makes extensive use of fuzzy logic to enable the non-player characters to behave spontaneously, based on their defined personalities and abilities [45].

In summary, fuzzy logic is a superset of traditional Boolean logic, with similar rules and operations. The main difference lies in the use of Fuzzy Linguistic Variables (FLVs) that define a range of values to be used in place of crisp values. Consequently, a small number of FLVs and rules can be used in place of extensive, hard-coded Boolean rule bases. Fuzzy logic has many commercial applications and can be successfully applied in games for decision-making and behaviour selection.

Fuzzy State Machines

A fuzzy state machine (FuSM) brings together fuzzy logic and FSMs. Instead of determining that a state has or has not been met, a FuSM assigns different degrees of membership to each state. Therefore, instead of the states on/off or black/white, a FuSM can be in the states 'slightly on' or 'almost off'. Furthermore, a FuSM can be in both the 'on' and 'off' states simultaneously to various degrees. Therefore, in a game situation, a non-player character doesn't have to simply be 'mad' at the player. Instead, they can be 'almost mad', 'very mad' or 'raging mad' at the player, behaving differently in each situation [14]. Thus, by using a FuSM, a character can have varying degrees of membership of a state assigned to it and these states do not have to be specific or discreet. The method for calculating these degrees of membership are determined by the programmer and plenty of game testing.

In games, it is important that behaviour is not predictable. However, in FSMs, the requirement of determinism prevents variable behaviour from being exhibited, as they are composed of a large set of predetermined states and transitions. On the other hand, FuSMS are composed of fewer, non-deterministic transitions [10], allowing greater flexibility and variability with far fewer fuzzy states and transitions.

A FuSM is an easy way to implement fuzzy logic, which can allow more depth in the representation of the concepts and relationships between objects in the game world. A FuSM can increase gameplay by allowing for more interesting and varied responses by non-player characters, which leads to less predictable non-player character behaviour. Therefore, the player can interact with non-player characters that can be various degrees of 'mad', 'wounded' or 'helpful'. This variability increases gameplay by adding to the level of responses that can be developed for the non-player character and seen by the human player. Also, a FuSM can increase replayability of a game by expanding the range of responses and conditions that the player may encounter in given situations during the game. Therefore, the player will be more likely to experience different outcomes in similar situations each time they play the game [14].

FuSMs can be used in varying forms in different types of computer games. For example, a FuSM could be used in a role-playing game or first-person shooter for the health or hit points of a non-player character or agent [14]. In this case, instead of the finite states healthy or dead, a range could be used for the hit points that would allow the agent to be in the fuzzy states 'totally healthy', 'almost healthy', 'slightly wounded', 'badly wounded', 'almost dead' or 'dead'. In a racing game, a FuSM could be used for the control process for accelerating or braking an AI-controlled car. So, the FuSM would allow various degrees of acceleration or braking to be calculated rather than the finite states of 'throttle-up', 'throttle-down', 'brake-on' and 'brake-off' [14]. Furthermore, a FuSM is ideal for representing non-player character emotional status and attitude toward the player or other non-player characters. That is, instead of simply 'liking' or 'disliking' the player, the non-player character could have a range of emotional states from 'really liking' or 'rather liking' to 'slightly disliking' or 'violently disliking' the player.

The games that have made use of FuSMs include Civilisation: Call to Power, Close Combat 2, Enemy Nations, Petz and The Sims. In Call to Power, FuSMs are used to set priorities for the strategic level AI, allowing the creation of new unit types and civilisations. Close Combat 2 uses a FuSM that weights hundreds of variables through many formulas to determine a probability of a particular action [45].

In summary, FuSMs are a combination of FSMs and fuzzy logic, meaning that they consist of fuzzy states and fuzzy transitions, rather than the usual finite set of crisp states and transitions. Consequently, FuSMs can represent a greater variation in states and transitions with far fewer variables and rules than in an FSM, where everything must be hard-coded. Most games that make use of FuSMs do so in combination with other techniques such as flocking, FSMs or neural networks. FuSMs are ideal for controlling the behaviour of game characters, giving greater variation in actions and reactions.

Flocking

Flocking is an AI technique for simulating natural behaviours for a group of entities, such as a herd of sheep or a school of fish [18]. Flocking, also known as swarming or herding, was developed by Craig Reynolds in 1987 [42] as an alternative to the conventional method of scripting the paths of each bird individually. Scripting, for a large number of individual objects, was tedious, error-prone and hard to edit. In flocking, each bird in the flock is an individual that navigates according to its local perception of its environment, the laws of physics that govern this environment and a set of programmed behaviours. Flocking assumes that a flock is simply the result of the interaction between the behaviours of individual birds. Also, flocking is a stateless algorithm, which means that no information is maintained from update to update [42]. Each member revaluates its environment at every update cycle. This reduces the memory requirements and allows

the flock to be purely reactive, responding to the changing environment in real time.

Flocking has been used with great success in a variety of commercial titles. It can provide a powerful tool for unit motion and for creating realistic environments the player can explore [42]. For example, in a real-time strategy or role-playing game, flocking can be used to allow groups of animals to wander the terrain more naturally and for realistic unit formations or crowd behaviours [43]. For example, groups of swordsmen can be made to move realistically across bridges or around obstacles, such as boulders. Alternatively, in first person shooter games, monsters can wander the dungeons in a more believable fashion, avoiding players and waiting until their flock grows large enough to launch an attack.

Apart from games, the possible applications of flocking include the visual simulation of bird flocks or fish schools in computer animation or the simulation of crowds of extras for feature films. For example, the movie Batman Returns made use of flocking algorithms to simulate bat swarms and penguin flocks [39]. Also, flocking could aid in predicting traffic patterns, such as the flow of cars on a freeway, or be used in the scientific investigation of flocks, herds or schools [38].

Many games have successfully used flocking to simulate the group behaviours of monsters and animals. Games that have used flocking include Half-Life, Unreal and Enemy Nations [45]. Half-Life uses flocking to simulate the squad behaviour of the marines, who run for reinforcements when wounded, lob grenades from a distance and attack the player with dynamic group tactics. Unreal used flocking for many of the monsters as well as the other creatures such as birds and fish. Enemy Nations used a modified flocking algorithm to control unit formations and movement across a 3D environment [42].

In summary, flocking is currently used widely in games where there are groups of animals or monsters that need to simulate life-like flock behaviour. It is a relatively simple algorithm and only composes a small component of a game engine. However, flocking makes a significant contribution to games by making an attack by a group of monsters or marines realistic and coordinated. It therefore adds to the suspension of disbelief of the game and is ideal for real-time strategy or first-person shooter games that include flocks, swarms or herds.

Decision Trees

Decision tree learning is a method for approximating discrete-valued target functions. It is one of the most widely used and practical methods for inductive inference [19]. Additionally, decision trees are robust to noisy data and missing values. Consequently, they are a standard tool in data mining. Decision trees are generally preferred over other nonlinear techniques due to the readability of their learned rules and the efficiency of their training and evaluation [7]. A decision tree acts as a predictor or classifier for classifying a particular example into one of a given set of classes. Each example is a description of an instance composed of a set of attribute-value pairs. Similar

to biological trees, decision trees have a single root, which branches out into various subtrees, which in turn have subtrees, until terminating in leaves.

Decision trees are widely used in data mining, to find relationships in large sets of data and to predict future outcomes. They have been successfully applied to industrial applications in marketing, finance, manufacturing and health care. However, their use in commercial computer games so far has been limited. Decision trees are applicable in games where classification or prediction is required. For example, a character could use a decision tree to learn which of a set of actions will most likely have the best result in different situations. This learning could be achieved by using the example situations during play to build up a tree and then using the tree to estimate the best action to take. Alternatively, the tree could be pre-built before shipping and simply used for prediction, rather than learning. Another example would be to allow a character to learn about objects or other characters in its environment. The tree would be built of attributes of objects the character has encountered and their classification, or type. Then, given a new object, the character could predict what the object is and what to do with it.

Decision trees are appropriate for problems in which the instances can be represented as attribute-value pairs. That is, the instances are described by a fixed set of attributes and their values. The easiest situation for decision tree learning is when each attribute has a small number of possible values. Also, decision trees can only be used when the target function has discrete output values. This allows the decision tree to assign a classification to each example, chosen from two or more possible classes [19]. Decision trees are robust in the presence of errors, missing data and large numbers of attributes. They do not require long training time to estimate and are easier to understand than other types of models, as the derived rules have a straightforward interpretation.

The game Black & White allows the player to have a creature that could learn from the player and other creatures as the game progresses. Each creature has a set of beliefs based on a Belief-Desire-Intention architecture. A creature's beliefs about objects are represented symbolically as a list of attribute-value pairs and its beliefs about types of objects are represented as decision trees. The creature has opinions about what types of objects are most suitable for satisfying different desires [16]. The creature can learn opinions by dynamically building decision trees. The creature remembers the learning episodes and uses the attributes that best divide the learning episodes into groups. The algorithm used is based on Quinlan's ID3 algorithm [15]. For example, a creature learns what sorts of objects are good to eat by looking back at its experience of eating different types of things and the feedback it received in each case, such as how nice it tasted. The creatures tries to makes sense of this data by building a decision tree that minimises entropy, a measure of the degree of disorder of the feedback [16].

In short, decision trees are straightforward tree-like structures that are used for learning, classification and prediction. Although decision trees have not been used

widely in games, they are much simpler to implement, tune and understand than other learning and classification techniques, such as neural networks. Therefore, they should be one of the first nonlinear techniques to be trialled in future games and become used more widely. They are ideal for allowing a character to explore and learn about concepts and objects during the game. Alternatively, the decision tree could be built prior to shipping and used for a character's decision making.

Neural Networks

An Artificial Neural Network (NN) is an electronic simulation based on a simplified human brain. In an NN, knowledge is acquired from the environment through a learning process and the network's connection strengths are used to store the acquired knowledge [20].

Choosing the variables from the game environment that will be used as inputs is the most labour intensive part of developing an NN [30]. This difficulty is due to the fact that there is wealth of information that can be extracted from the game world and choosing a good combination of relevant variables can be difficult. Also, the number inputs needs to be kept to a minimum to prevent the search space from becoming too large [9]. Therefore, it is a good idea to start with the essential variables and add more if required. Choosing inputs that are poor representations of the game environment is the primary reason for failed applications.

NNs are techniques that can be used in a wide variety of applications. Some common uses include memory, pattern recognition, learning and prediction. There are many commercial applications of NNs across various industries, including business, food, financial, medical and health care, science and engineering [24]. Prominent companies that are using NNs include Microsoft, Sharp Corporation, Mars, Intel, John Deere, Mastercard, Fujitsu and Siemens [24]. Some examples of applications that NNs are being used for are predicting sales, handwritten character recognition for PDAs and faxes, odour analysis via electronic nose, stock market forecasting, credit card fraud detection, Pap smear diagnosis, protein analysis for drug development and weather forecasting. This list illustrates the wide variety of applications that can make successful use of NNs, and how their usefulness is only limited to what can be imagined.

The computer game industry is no different from the industries mentioned above in terms of the variety of applications of NNs. A few applications are described by LaMothe [26], including environmental scanning and classification, memory and behavioural control. The first application, environmental scanning and classification, involves teaching the NN how to interpret various visual and auditory information from the environment, and to possibly choose a response. The second application, memory, involves allowing the AI to learn a set of responses through experience and then respond with the best approximation in a new situation. Finally, behavioural control relates to the output of the NN controlling the actions of the AI, with the inputs being various game engine variables. Also, the NN can be taught to imitate the human player of the game [31].

Basically, an NN can be used to make decisions or interpret data based on previous input and output it has been given. The input can be seen as various games states, similar to that used by a state machine, and the output could be the action to be performed. The important difference is that the current state doesn't need to have been hard-coded. Instead, the NN makes the best approximation that it can, based on the states that it already knows about. This means that the NN will choose an action that would have been performed in a *similar* state.

So far, game developers have been reluctant to allow a game to ship with the learning in NNs and other techniques "switched on", in case the AI were to learn something stupid [46]. Therefore, the developers that have used NNs in their games have not used them for learning, but rather trained them during development and locked the settings before shipping. Some examples of games that include NNs for various tasks include BattleCruiser: 3000AD, Black & White, Creatures, Dirt Track Racing and Heavy Gear.

In BattleCruiser: 3000AD (BC3K) the AI uses NNs to control the non-player characters as well as to guide negotiations, trading and combat [45]. It uses the development language AILOG (Artificial Intelligence & Logistics), which was created by the developer of BC3K, Derek Smart, and uses an NN for very basic goal oriented decision making and route finding, with a combination of supervised and unsupervised learning. In Black & White the player has a creature that learns from the player and other creatures. The creature's mind includes a combination of symbolic and connectionist representations, with their desires being represented as NNs [15]. Finally, the Creatures series of games makes heavy use of Artificial Life techniques, including heterogeneous NNs, in which the neurons are divided into lobes that have individual sets of parameters. In combination with genetic algorithms, the creatures use the NN to learn behaviour and preferences over time.

In short, NNs are techniques that can be used for a wide range of applications in many different environments. Several commercial games have used this technique successfully, with the most recent and prominent being the game Black & White. This technique's flexibility means that it has the potential to be applied in a wide range of situations in future games. Therefore, it is likely that NNs will play a bigger role in commercial games in the near future.

Genetic Algorithms

A Genetic Algorithm (GA) is an AI technique for optimisation and machine learning that uses ideas from evolution and natural selection to evolve a solution to a problem [8]. A GA works by starting with a small number of initial strategies, using these to create an entire population of candidate solutions and evaluating each candidate's ability to solve the problem. Gradually, more effective candidates are evolved over several generations until a specified level of performance is reached [27].

The possible applications of GAs are immense. Any problem that has a large enough search domain could be suitable [22]. Traditional methods of search and optimisation are too slow in finding a solution in a very complex search space. However, a GA is a robust search method requiring little information to search effectively in a large, complex or poorly-understood search space. GAs are also useful in nonlinear problems [11]. There are many applications that can benefit from the use of a GA, once an appropriate representation and fitness function has been devised. An effective GA representation and meaningful fitness evaluation are the keys to the success of GA applications. The appeal of GAs comes from their simplicity and elegance as robust search algorithms, as well as from their power to discover good solutions rapidly for difficult high-dimensional problems. GAs are useful and efficient when domain knowledge is scarce or expert knowledge if difficult to encode to narrow the search space, when no mathematical analysis is available and when traditional search methods fail [12].

GAs have been used for problem solving and modelling, and applied to many scientific, engineering, business and entertainment problems [12]. Also, GAs have been extensively explored by academics. However, they are yet to become accepted in game development. They offer opportunities for developing interesting game strategies in areas where traditional game AI is weak. For example, a GA could be used in a real-time strategy game to adapt the computer's strategy to exploit the human player's weaknesses. This GA would need to consider things like how the player's base is set up, how well they can cope with multiple engagements, unit mobility and combined force flexibility. A GA could also be used in a real-time strategy to define the behaviour of individual units rather than groups of units or the overall strategy [23]. Additionally, a GA could be used in a role-playing game or first-person shooter to evolve behaviours of characters and events [34]. For example, a GA could take the creatures in the game that have survived the longest and evolve them to produce future generations. This would only need to be done when a new creature is needed [25]. Furthermore, GAs could be used in games for pathfinding, in which the chromosome could represent a series of vectors and the fitness function could be the distance the sum of vectors is away from a target point [8].

The downside is that in game development AI has to fight with graphics and sound for scarce CPU time and resources. GAs are computationally expensive and the more resources they can access the better. Also, large populations and more generations give better solutions. Therefore, GAs are better used offline. One solution is that the GA could work on the user's computer while the game is not being played, utilising the computer's down time. Alternatively, all the work could be done in-house before shipping and then released with the parameters locked.

Computer games that have used GAs include Cloak, Dagger, and DNA, the Creatures series, Return Fire II and Sigma. Cloak, Dagger, and DNA uses GAs to guide the computer opponent's play. It starts with four DNA strands, which are rules governing the behaviour of the computer

opponents. As each DNA strand plays it tracks how well it performed in every battle. Between battles the user can allow the DNA strands to compete against each other in a series of tournaments, which allows each DNA strand to evolve. There are a number of governing rules for DNA strand mutation, success and so on, and the player is able to edit a strand's DNA ruleset. The Creatures series of games makes more use of Artificial Life technology, such as GAs and NNs, than any other series of games. It uses a combination of heterogeneous NNs and a GA-like winnowing process to push evolution of the creatures. It is effectively a self-training NN that allows the creatures to learn over time what they like, what they're not supposed to do and so on.

In summary, GAs are based on evolution and natural selection and are used for learning and optimisation. They are resource intensive and require much time in development and tuning, which does not make them ideal for in-game learning. Generally, the most difficult part in GA development is determining a suitable representation for the solutions. Also, parameters such as population size, mutation and recombination operators and the number of solutions to erase, make parents or keep unchanged can take a long time to tune. Basically, a GA is not a good algorithm to incorporate into a game where time and resources are limited. Unfortunately, this describes most commercial games. However, GAs also have many advantages, in that they are a robust search method for large, complex or poorly-understood search spaces and nonlinear problems. In short, if GAs are to be used in games, they will most likely be evolved before shipping or between games, and it will be a long time before they become widespread in games.

Extensible AI

Some game developers have built various degrees of extensibility into their game AI and made it accessible to the user community. These games provide some functionality through which the user can modify or develop customised AI for the game [44]. Games with extensible AI provide greater flexibility to the players. Also, customising the actions and reactions of game characters increases the replayability of the game and gives the player a greater "stake" in their characters [45]. Furthermore, games that have successfully implemented extensible AI usually have large online user communities dedicated to providing tutorials and documentation, swapping their creations and showing off their prowess in writing their own AI. Games such as Quake and Unreal have cult followings of people who enjoy coding and trading their own game bots.

There are many different games using a range of methods that allow the player to customise or create their own AI. Most of these methods are based on either scripting or some kind of toolkit. Scripting is where the player can actually write their own AI code for the game characters and usually accesses the game data files rather than the actual game codebase. These scripting languages are usually customised, English-like languages that are used by game designers in development. The other method of providing extensible AI, toolkits, includes a wide range of

different interfaces and varying control over the game AI. Some toolkits merely allow the player to tweak parameters and others allow them to have total control of all character and scenario facets, almost to the point of scripting it themselves.

There are many different games that allow the player to edit the game's monsters, characters and scenarios. Some well-known games that include extensible AI are Age of Empires, Baldur's Gate, Civilisation: Call to Power, Dark Reign, Halflife, Quake, Unreal, Warzone and most recently, Neverwinter Nights. Following is a brief description of how these games make use of extensible AI.

Age of Empires includes a scripting capability through the game's data files, which gives the user some ability to design and customise the game AI. In Baldur's Gate, players are able to directly edit scripts that control the actions of their non-player characters. Each non-player character has its own AI scripting, outlining its basic reactions to basic situations [45]. The game Civilisation: Call to Power allows the players to modify unit attributes and access the fuzzy logic rule sets used by the AI to set priorities for the strategic AI. This allows the creation of new unit types and civilisations. In Dark Reign, the user can design their own missions and adjust the computer AI within that mission. It also allows the player to tailor the behaviour of individual units for the game. Halflife provides the player with a toolkit to allow the development of customised AI bot code. Quake 1 includes a kit that allows the player to write code and modify the behaviour of enemies and weapons. Quake 2 extends the engine's capabilities with a fully java scripting language interface [44]. Unreal allows the player to write their own mods and game types through a custom scripting language called Unrealscript. Warzone 2100 features an extensible AI that uses a basic C-like language in external scripts, which provides a fair amount of flexibility. Neverwinter Nights provides a graphical interface to allow the user to create their own scenarios, monsters, non-player characters and tell them how to behave in different situations through scripting in C++. Finally, a not so popular game that utilises extensible AI that is worth a mention is Cloak, Dagger, and DNA, as it uses GAs. This game offers the user a "lab" for breeding new and better AIs. The user has some control over the breeding and evolution of the AIs and can tailor them to be more aggressive, risky and so on [44].

Conclusion

It is easy to see why current games are employing techniques such as finite state machines, scripting, agents and flocking. It is because these techniques are simple, require little tuning compared to the alternatives and adhere to the game developer's constraints of time and resources. However, users are growing weary of predictable and deterministic game characters, and so developers must look to new techniques for solutions. Also, as the graphics race in games diminishes, it is likely the next wave of games will require superior AI as a selling point, rather than more polygons, higher resolution textures and more frames-per-second. This will drive the need to create better AI, using more advanced techniques

such as fuzzy logic, decision trees, NNs and GAs. As more development and processor time is allocated to AI, these advanced techniques will allow games to include in-game learning, adaptive strategies and a wide range of non-linear behaviour from game characters and events. Ultimately, games will converge to like-like behaviour rather than the typical "there's no point doing it if the user can't see it" that resounds from game developers today.

Appendix. Summary Table of AI Techniques in Games

Technique	Advantages	Disadvantages	Applications	Games
Finite State Machine	- simple - general - use in conjunction with other techniques - computationally inexpensive - lots of power relative to complexity	- can be poorly structured - poor scaling - need to anticipate all situations - deterministic	- manage game world - manage objects / characters	- Age of Empires - Half-Life - Doom - Quake
Scripting	- simple - can be used by non-programmers - safe environment	- deterministic - need to anticipate all situations	- events - opponent AI - tell the story - automate tasks - conversation trees	- Black & White - Unreal - Dark Reign - Baldur's Gate
Fuzzy Logic	- when no simple solution - when expert knowledge is needed - non-linear problems - more flexible, variable	- not good when there is a simple solution - complicated to build from scratch	- decision making - behavioural selections - input/output filtering - health of NPC - emotional status of NPC	- SWAT 2 - Call to Power - Close Combat - Petz - The Sims
Flocking	- purely reactive - memory requirements - realistic / lifelike	- limited applications	- unit motion - groups of animals / monsters	- Half-Life - Unreal - Enemy Nations
Decision Trees	- robust to noise / missing values - readable - efficient training / evaluation - simpler than NNs	- need tuning	- prediction - classification - learning	- Black & White
Neural Networks	- flexible - non-deterministic - non-linear	- need tuning - choosing variables is difficult - complicated - resource intensive	- memory - pattern recognition - learning - prediction - classification - behavioural control	- Black & White - BC3K - Creatures - Heavy Gear
Genetic Algorithms	- robust search method - effective in large, complex, poorly understood search spaces - non-linear - non-deterministic	- resource intensive - slow - need a lot of tuning - complicated	- optimisation - learning - developing game strategies - evolve behaviour - pathfinding	- Cloak, Dagger & DNA - Creatures - Return Fire II

References

[1] Alexander, T. “An Optimised Fuzzy Logic Architecture for Decision-Making.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002.

[2] Attar Software Limited. “White Paper: Fuzzy Logic in Knowledge Builder.” Retrieved June 20, 2002, from http://www.attar.com/pages/fuzzy.htm, 2002.

[3] Barnes, J. “Scripting for Undefined Circumstances.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 530-540.

[4] Bauer, P., Nouak, S. and Winkler, R. “A Brief Course in Fuzzy Logic and Fuzzy Control.” Retrieved March 7, 2002, from http://www.flll.uni-linz.ac.at/pdw/fuzzy/index.html, 1996.

[5] Berger, L. “Scripting: Overview and Code Generation.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 505-510.

[6] Brockington, M. & Darrah, M. “How Not to Implement a Basic Scripting Language.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 548-554.

[7] Brown, M. “Decision trees.” Retrieved July 18, 2002, from http://www.cse.ucsc.edu/research/compbio/genex/genexTR2html/node10.html, 1999.

[8] Buckland, M. “Genetic Algorithms in Plain English.” Retrieved March 7, 2002, from http://www.btinternet.com/~fup/ga_tutorial.html, 2002.

[9] Champandard, A. J. “The Dark Art of Neural Networks.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 640-651.

[10] Collins, E. “Evaluating the Performance of AI Techniques in the Domain of Computer Games.” Retrieved July 16, 2002, from http://www.dcs.shef.ac.uk/_u8aec/com301, 2001.

[11] Dulay, N. “Application of Genetic Algorithm.” Retrieved July 18, 2002, from http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/tcw2/article1.html, 1996.

[12] Dulay, N. Genetic Algorithms. Retrieved July 18, 2002, from http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/tcw2/report.html, 1996.

[13] Dybsand, E. “A Finite-State Machine Class.” In M. Deloura (Ed.), Game Programming Gems. Hingham, MA: Charles River Media, Inc., 2000, pp. 237-248.

[14] Dybsand, E. “A Generic Fuzzy State Machine in C++.” In M. Deloura (Ed.), Game Programming Gems 2. Hingham, MA: Charles River Media, Inc., 2001, pp. 337-341.

[15] Evans, R. “AI in Games: A Personal View.” Retrieved March 7, 2002, from http://www.feedmag.com/templates/default.php3?a_id=1694, 2001.

[16] Evans, R. “AI in Computer Games: The Use of AI Techniques in Black & White.” Retrieved July 23, 2002, from http://www.dcs.qmul.ac.uk/seminars/theory/abstract/EvansR01.html, 2001.

[17] Generation5. “AI in Gaming.” Retrieved July 16, 2002, from http://www.generation5.org/app_game.shtml, 2002.

[18] Grub, T. “Flocking.” Retrieved March 7, 2002, from http://www.riversoftavg.com/ flocking.htm, 2001.

[19] Hamilton, H. “Overview of Decision Trees.” Retrieved July 18, 2002, from http://www.cs.uregina.ca/%7Ehamilton/courses/831/notes/ml/dtrees/4_dtrees1.html, 2002.

[20] Haykin, Simon S. “Neural Networks: A Comprehensive Foundation.” New York: Maxwell Macmillan International, 1994.

[21] Howland, G. “A Practical Guide to Building a Complete Game AI.” Retrieved March 7, 2002, from www.lupinegames.com/articiles/prac_ai.html, 1999.

[22] Hsiung, S., and Matthews, J. “An Introduction to Genetic Algorithm and Genetic Programming.” Retrieved July 16, 2002, from http://www.generation5.org/ga.shtml, 2000.

[23] James, G. “Using Genetic Algorithms for Game AI.” Retrieved July 18, 2002, from http://www.gignews.com/gregjames1.htm, 2002.

[24] Keller, P. Pacific Northwestern Laboratory. “Commercial Applications, Artificial Neural Networks.” Retrieved June 5, 2002, from http://www.emsl.pnl.gov:2080/proj/neuron/neural/products, 1997.

[25] LaMothe, A. “Tricks of the Windows Game Programming Gurus.” Indianapolis, Indiana: SAMS, 1999.

[26] LaMothe, A. “A Neural-Net Primer.” In M. Deloura (Ed.), Game Programming Gems. Hingham, MA: Charles River Media, Inc, 2000, pp. 330-350.

[27] Laramee, F. D. “Genetic Algorithms: Evolving the Perfect Troll.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc, 2002, pp. 629-639.

[28] Logan, B. “Intelligent Agents.” Retrieved July 16, 2002, from http://www.cs.nott.ac.uk/IPI/agents.html, 2001.

[29] Logic Programming Associates, Ltd. “About Fuzzy Logic”. Retrieved 8 March, 2002, from http://www.lpa.co.uk/ind_pro.htm, 2002.

[30] Manslow, J. “Using a Neural Network in a Game: A Concrete Example.” In M. Deloura (Ed.), Game Programming Gems 2. Hingham, MA: Charles River Media, Inc., 2001, pp. 351-357.

[31] Manslow, J. “Imitating Random Variations in Behaviour Using a Neural Network.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 624-628.

[32] McCuskey, M. “Fuzzy Logic for Video Games.” In M. Deloura (Ed.), Game Programming Gems. Hingham, MA: Charles River Media, Inc., 2000, pp. 319-329.

[33] Neitz, J. & Lima, H. “Game Playing: Modern Games.” Retrieved July 16, 2002, from http://sern.ucalgary.ca/courses/CPSC/533/W99/presentations/L2_5B_Lima_Neitz/modern.html, 1999.

[34] NeuroDimension, Inc. “Genetic Algorithms: Common Applications.” Retrieved July 18, 2002, from http://www.nd.com/products/genetic/apps.htm, 2002.

[35] Poiker, F. “Creating Scripting Languages for Nonprogrammers.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 520-529.

[36] Rabin, S. “Designing a General Robust AI Engine.” In M. Deloura (Ed.), Game Programming Gems. Hingham, MA: Charles River Media, Inc., 2000, pp221-236.

[37] Rabin, S. “Implementing a State Machine Language.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 314-320.

[38] Reynolds, C. “Flocks, Herds, and Schools: A Distributed Behavioural Model.” Computer Graphics 21(4), pp. 25-34, 1987.

[39] Reynolds, C. “Boids.” Retrieved 10 July, 2002 from http://www.red3d.com/cwr/boids/, 2002.

[40] Scott, B. “Architecting an RTS AI.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc., 2002, pp. 397-401.

[41] Stripinis, D. “The (Not So) Dark Art of Scripting for Artists.” Game Developer Magazine 8(9), pp. 40-45, 2001.

[42] Woodcock, S. “Flocking: A Simple Technique for Simulating Group Behaviour.” In M. Deloura (Ed.), Game Programming Gems. Hingham, MA: Charles River Media, Inc., 2000, pp305-318.

[43] Woodcock, S. “Flocking with Teeth: Predators and Prey.” In M. Deloura (Ed.), Game Programming Gems 2. Hingham, MA: Charles River Media, Inc., 2001, pp. 330-336.

[44] Woodcock, S. “Games with Extensible AIs.” Retrieved July 20, 2002, from http://www.gameai.com/exaigames.html, 2002.

[45] Woodcock, S. “Games Making Interesting Use of Artificial Intelligence Techniques.” Retrieved March 7, 2002, from http://www.gameai.com, 2002.

[46] Woodcock, S. “AI Roundtable Moderator’s Report.” Retrieved April 9, 2002, from http://www.gameai.com, 2002.

[47] Zarozinski, M. “An Open-Source Fuzzy Logic Library.” In S. Rabin (Ed.), AI Game Programming Wisdom. Hingham, MA: Charles River Media, Inc. 2002.