Maintenance of Game Character's AI by Players

Stephen Burman[1], Yang Sok Kim[1], Byeong Ho Kang[1], Gil-Cheol Park[2]
[1]School of Computing, University of Tasmania
e-mail : {sburman, yangsokk, bhkang}@se.ac.kr
[2] Dept. of Multimedia Engineering, Hannam University
e-mail : gcpark@mail.hannam.ac.kr

Abstract With the development of computer games, different game worlds and various game characters are found within them. Various Artificial Intelligence (AI) techniques are usually used to define behavior s of the characters within game worlds, which are controlled by AI algorithms in the computer as well as by the user. The AI techniques defined for these characters are generally developed by the game creators and cannot be changed without going to some effort, which means that if a user wished to control the behavior s of a character within a game, they could not easily do so. Being able to edit the behavior s of AI characters is beneficial as it gives the user extra control over their characters. Therefore a method for allowing a user to easily personalize the AI characters was needed. This goal was achieved by using an incremental knowledge acquisition method, called the MCRDR. The MCRDR allows the user to easily acquire new control knowledge of the AI characters by combining rule-based and case-based knowledge acquisition approach. Our experiment results showed that AI of a character could be personalized with this method of knowledge extraction.

Keyword: Computer Game, Game Character Personalization, Knowledge Acquisition, MCRDR

1. Introduction

Artificial Intelligence (AI) can be found within the various characters of a computer game. Being able to personalize the AI of a character would allow the user to gain higher levels of control over game characters that they can command. Some games such as Quake 2, Unreal Tournament and Half-Life provide an interface that allows anyone to write code to control a character within the game [1]. The problem with this approach is that one would need knowledge of programming to use such an interface. Similarly, these interfaces are generally only used to create purely computer-controlled character AI. It can be seen that it is hard to personalize the AI of a character for the average game player. To try and find an easy way to allow someone to edit the AI of a character, one must understand what goes into a game character's AI.

The aim of this research is to try and place the knowledge of the user into a game character such that the user can personalize AI of the game character to what they believe is appropriate behavior and therefore give the user higher levels of control. MCRDR is a proven way to extract knowledge from an expert in their respective field. Therefore it has been chosen as the method for extracting knowledge from the game player to increase the levels of control the game player can have within a game. For the purpose of our study, an RTS game was created as well as an AI editor to allow the user to create AI using MCRDR.

This paper consists of following contents: Section 2 summarizes various AI architectures that are employed in the computer game area and provide limitations of current approach. Section 3 provides a detailed explanation of our approach, called MCRDR. Section 4 explains implementation details, including game itself, MCRDR, and AI editor. Section 5 summaries what experiments are conducted and what results are obtained from them. The conclusions of this paper are in the Section 6.

2. AI Techniques in the Game Characters

Following are some different types of AI techniques that are applied in the computer game:

Finite State Machine (FSM). FSM works by pre-programming a set of states that a game character can be in. A state is chosen for the character depending on the information about the game world that the character is given, as well as the current state that the character is already in [2],[3],[4]. A state is basically a condition. The FSM can be seen as a 'black box' that takes inputs (information about the game world) and produces outputs (actions that the character can perform in the game world) [2]. The output is an abstraction of a set of lower level movement and actions [1]. The benefit of a FSM is that it is considered reliable and still has a good enough result. From the issues that were discussed in the AI roundtables at the 2000 Game Developers Conference, the general trend for game character AI tended to be for more traditional FSM methods than neural nets and genetic algorithms [5]. This is due to the fact that a FSM is useful for reducing complex behavior s into smaller and simpler behaviors that make it easier to debug and tune than neural nets and genetic algorithms [6]. The drawback of this approach is that the states are pre-programmed, and therefore the character's actions will generally be predictable. The programmer can try to add new rules to make the character more complex and unpredictable; however there will always be a limited level of unpredictability.

SOAR. SOAR architecture [7] consists of operators. Each operator can be primitive performable actions, internal actions such as remembering the last position of the enemy or more abstract higher-level actions that decompose into lower-level primitive actions. SOAR can be viewed as a set of user-written if-else rules [2]. The way in which SOAR works is by continually proposing, selecting, and applying operators to the current state with the if-else rules that match against the current state. There is the possibility that if an operator is selected it cannot be applied immediately. In this case a sub-state is created where additional operators are chosen until the original operator can be applied or the state changes so much that the original operator is no longer required [1].

Explanation-base Learning (EBL). EBL is a method for learning that increasing execution speed. By remembering actions that were selected for previously similar situations, the sequence of conditions that lead to the original conclusion are remembered and a new rule is proposed that gives the same conclusion given the same conditions. As the character's AI does not have to compute the same result given similar input, the speed of execution time is increased [1].

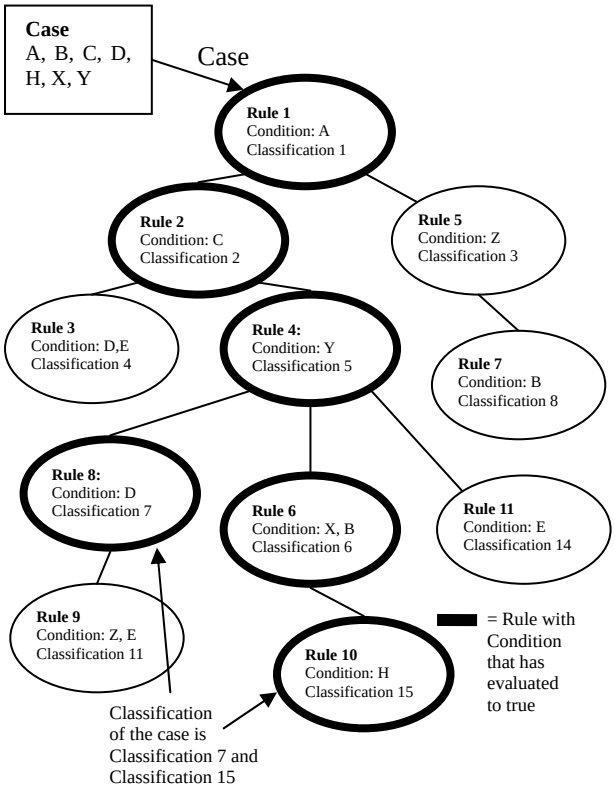
AI Scripts. A script is a way of defining AI outside of the actual program [4, 8, 9]. Scripts are used to specify an action or sequence of actions that should occur given a specific event. Therefore if the condition holds true, then the scripted action will fire. The AI script is read into the game and is therefore useful for allowing people to create their own AI and have it imported [3].

Many programmers consider a method from those described in the above to be good enough when creating AI. However, one needs to have a top-down view of the entire internal representation of the character in order to create the desired actions. This makes it harder for a non-programmer to implement such an AI, especially since they probably did not create it in the first place or have access to the AI now. In a similar respect, these methods make the assumption that knowledge can be successfully extracted from the user such that they can represent the knowledge in the intended way. Therefore, a method is needed that easily allows a user to place their knowledge into the game character. It must also be easy to change this knowledge without disrupting the previous performance of the rest of the system. This would allow the user to be able to change a small piece of a character's AI without having to consider the entire structure. Knowledge must also be quickly retrieved so that the system will cope in a game-playing environment. A solution that incorporates these areas is a method of knowledge acquisition known as Multiple Classification Ripple Down Rules (MCRDR).

3. MCRDR

MCRDR. MCRDR is an altered version of standard RDR, an approach to building a knowledge-based system without the aid of knowledge engineers to extract or maintain the knowledge [10]. As the name suggests, the main differences between RDR and MCRDR is that multiple classifications for a case can be returned with MCRDR [11]. There are, however, differences in how the inference processes work. MCRDR is represented in a tree structure. However, MCRDR is an n-ary tree rather than a binary tree and therefore each rule in the tree can have any number of children [11]. Each rule has a condition and a classification associated with it. When a case is to be classified in MCRDR, the case is given to the root rule. From here, the condition evaluates the case to be either true or false. If it is true, then the root rule passes the case onto its children. If the condition evaluates the case to false, then case is not passed. The case can follow multiple paths down the rule tree [12], unlike RDR where it can only go down one unique path. For every path that is taken down the rule tree, the classification associated with the last rule on every path that returns true for the evaluation of the case is considered to be a classification for that case. In this way, MCRDR can return multiple classifications for a given case [13].

Example. An example of the MCRDR classification process is shown in Fig.1 The case has attributes A, B, C, D, H, X, Y. This case is given to the root rule and is evaluated. It can be seen that the case is evaluated by the root rule to be true as the condition for the root rule requires that A holds for the case, which it does. The case is then passed to the children of the root rule. Rule 5 evaluates the case to be false and therefore the case is not passed down that path any further. Rule 2 evaluates the case to be true and therefore it is passed down to Rule 2's children. Each rule that is passed the case can pass it down to its children if they evaluate the case to be true. At the end of this process, Rule 10 and Rule 8 are the last rules on their paths to have evaluated the case to true and therefore whatever classification is associated with these cases is considered to be the classifications for the given case. Therefore this case is considered to be a Classification 7, and a Classification 15.



(Fig.1) MCRDR inference example. After the case is given to the root rule, the final classifications that are returned are Classification 7 and Classification 15

How the MCRDR Grows. There are several ways in which a new rule can be added to prevent misclassifications. The first way is to simply add a rule with the appropriate classification after the node that caused the misclassification [14]. Adding a rule in this way may be thought of as a refinement of the parent rule. The expert can, however, add a new rule higher in the rule tree anywhere along the path that lead to the misclassifying rule [15]. This can be done to add another classification path.

Since MCRDR allows for multiple classifications, it is possible for a case to be correctly classified by some of the rules, but wrongly classified by others. Therefore it is possible that the only change to the rule-base that is to be made is to stop the rules that are causing the misclassification. The user can do this by inserting a

stopping rule. This rule is inserted and used in the same way as a refinement rule, except that the classification is a null classification [11, 12]. This means that if the stopping rule is called upon to provide a classification, then there will be no classification from this rule, as it will give null. The condition, however, must still be carefully chosen so that the stopping rule does not break the previous performance of the system. Stopping rules are a major way for preventing wrong classifications [12].

In a similar way to RDR, MCRDR stores cornerstone cases with every rule in the rule tree. RDR only requires a single cornerstone case, whereas MCRDR requires one or more [12]. When a new rule is created, the case that prompted the creation of the new rule is stored with the new rule as its cornerstone case. This case also becomes a cornerstone case for all the rules that gave a correct evaluation (i.e. evaluate to true) for the case along the path to the new rule [11, 12]. It can be seen that the root rule will have a copy of all the cornerstone cases in the rule tree since it is the beginning of every path. Similarly, rules that are lower down the tree have less cornerstone cases associated with them [12].

MCRDR requires validation for the same reasons that RDR does, however MCRDR must validate against more cornerstone cases. All rules share the same parent. When the parent passes down the cornerstone case of any of its children, all of its children will receive this case. Therefore a rule can be reached by the cornerstone cases that are associated with any of its sibling rules. Similarly, a rule can also be reached by the cornerstone cases associated with the children of sibling rules. For this reason, a rule must make sure that its condition does not evaluate to true for these cases [12]. Otherwise, the addition of the new rule may give a new and unwanted classification to a previously stored cornerstone case. When these cornerstones cases are being used to create a new condition, the expert is presented with a difference list of all the cornerstone cases that are to be considered in the creation of the condition for the new rule. The difference list is used to determine which features of the current case make it different from the stored cornerstone cases [15].

Benefits of MCRDR. Using MCRDR/RDR has following benefits: From the method just described, it can be seen that the MCRDR rule tree can grow without the help of a knowledge engineer. This means that the knowledge engineer is not relied upon for the acquisition of knowledge [16]. This is beneficial as the expert may not be able to convey their ideas correctly to the knowledge engineer. MCRDR/RDR also operates on a case-by-case basis. Knowledge acquisition can be improved with MCRDR/RDR because experts generally tend to be good at judging between cases rather than giving knowledge in abstract form. Similarly, the explanation of certain data can be situation dependant and the justification may vary with context [12]. Therefore, if such situations occur, the expert may be able to give more accurate information dealing with that specific case. Since MCRDR/RDR grows from fixing errors as they occur, it does not require any modeling or analysis of the domain for knowledge acquisition and maintenance. Studies have shown that knowledge bases produced by correcting errors as they occur are similarly as compact and accurate as those produced by induction [11]. On average, a knowledge base created with MCRDR will only be about 10% to 15% larger than the same knowledge base created using standard machine learning methods. As well as size, a benefit of MCRDR is that information is retrieved quickly.

In addition common benefits of MCRDR/RDR, MCRDR gives another benefit because it obviously supports multiple classifications. A result of this is that the rule tree for an MCRDR system can be considerably smaller than the rule tree for an RDR system. The reason for this is because in an RDR rule tree, there tends to be more duplication of information. The duplication occurs because only a single path through the rule tree can be made for a classification. Therefore if a classification was to be made for a certain case, and another classification was to be made for a similar case with a slightly different property, then a new rule would have to be created with a combined classification [17].

4. Implementation

4.1 AI and Game Considerations

Game World Considerations. The character's AI will depend on the type of game world that is created. From a movement point of view, the game world has been segmented into logical squares. Any object within the game has to occupy a square. The reason that this approach was taken rather than a more continuous game world was to simplify the game. The AI creation process was simplified with the square system due to the fact that users could determine a position on the game area simply with an x y coordinate. The game area was limited to a 40 x 30 square grid size. There are two main reasons for this. Firstly, because the game was designed more for AI creation rather than advanced game features, it was more beneficial to have the entire game area visible at the one time so that an AI creator can see exactly what is going. Therefore the game area is not large and fits on the entire screen. Secondly, the path-finding algorithm is recommended to work on a maximum area of 40 x 40 squares [18]. To make the game more realistic, a fog of war was implemented to stop the user (and game characters) from knowing the positions of enemy units that are not within sight range of any friendly units. Therefore, the game world that was created was made inaccessible. To help make the game more non-deterministic, the random action selection was introduced. Similarly, the user can use the mouse and keyboard to control the user team's characters, making the environment more random. The game area itself is static, however, the characters themselves can move, making the general environment dynamic. The game area was designed to be discrete as there are only a limited number of positions on the game area that can be reached.

Character Movement Considerations. Since the individual character takes care of its own movement and path-finding, then a path-finding algorithm must be implemented. The A* algorithm is a commonly used and effective path-finding algorithm for computer games [5, 19, 20]. To try and increase the speed and realism of path-finding, some alterations were made to the base A* algorithm. When considering if a square is taken or not taken, other game characters are only considered if they are within sight range of the character. Though this varies with the character type, it is roughly 4 squares. Otherwise the character would implicitly know about characters that it could not see. If the destination square is occupied by a game object (i.e. terrain, a character, a mine or a factory), the closest non-taken square is selected as the destination. This is to stop the character from trying to reach an unattainable goal. The game area is split up into two regions, one for each side of the river. If the destination

square is over the river, then the path-finding algorithm first finds a path to the bridge and then from the bridge to the destination. This saves some path-finding time. Finally, if a destination square is surrounded (i.e. unattainable), then the A* algorithm searches the entire game area to try and find a way in. To stop this, a check is done to see if the destination is within sight range. If it is, then a path from the destination to the character is attempted. If a path is found, then there is no problem. If the path is not found but leads outside the sight range of the character, then this is acceptable as the character itself may be surrounded and when the real path-finding starts it will terminate very quickly. If no path is found, then the destination is surrounded and therefore a path cannot be found to that location.

4.2 Implementation of MCRDR

Cases. The AI's decision making process depends on information regarding the game world in its current state and from previous states of the game world. Information is examined by an MCRDR engine via cases. The case must include all the information regarding the game world's current state at the point in time when the decision is being made. If more complex AI is to be created, then the case should also contain information regarding the previous events and game states that have already occurred. For this study, the only information used regarding the previous state was the previous five actions.

It is preferred to have computer-controlled characters to only have access to information that a human controlled character would have in the same position [1]. This allows for more realistic behavior, as well as making the game fairer. Therefore, the only information that a case for a character will contain is information regarding itself (eg the character's hit points, position etc), its fellow team members (eg a team member is being attacked, number of team members left etc), its team's factory (eg the number of units produced etc) as well as any enemies that are seen by the character or its team members. Regarding actions, the character can only know what action it is currently performing or has performed. The character does not know what actions its team members are performing, or what actions the enemy is performing. The exception to this is if a character or a character's team member is being attacked. This would indicate to the character that the enemy is attacking since it is attacking a friendly unit. This, however, is the only time when a character can specifically know what action an enemy is performing. When an enemy is seen, the only information that a character can know about that enemy is its position, hit points, and unit type. The location of the enemy's factory is always known, as it does not move. However, information about the enemy's factory such as the number of units produced cannot be known.

Classifications. The next major consideration when implementing MCRDR is how to represent classifications in the game environment. In the computer game context, AI aims to allow computer controlled characters to show some kind of cognitive process when taking actions or reacting to human players. The cognitive process that must be shown can be seen by the action the computer controlled character decides to perform. Therefore, at the end of the MCRDR inference process, the classifications that are found are actions that the character can perform.

Number of MCRDR Engines. It had to be decided whether there was to be only one MCRDR engine for the entire game, an MCRDR engine for each team, an engine for each character type or whether each character was to have its own individual MCRDR engine. The choice of having only one engine would be useful in terms of saving memory space as some rules may be reused for several characters. The problem with this approach is that there would need to be some easy way of determining which character is which and from what team.

If there would be separate sub-trees for each character types, then there is no real need to put them all together into one tree. There may as well be just a separate tree for each character type. This has a benefit as it means that there does not need to be rules deciding which team or which character a case is for. After these considerations it was decided that the best approach for this implementation was to use either a separate MCRDR engine for each character or a separate engine for each character type of a team.

MCRDR Structure Considerations. MCRDR deals with multiple classifications. The problem with using a multiple classification system in the context of AI for a game character is that the game character can only apply one action at a time. To address the problems resulting from using traditional MCRDR, an alternative was chosen. This alternative was to use the MCRDR engine but to remove the multiple classification aspect. From doing this, the problem of having to handle many cornerstone cases when creating a new rule is addressed. This simplifies the process of validation and the problem of deciding which action to apply from the list of actions is also no longer an issue as there is only one action returned. Since pure MCRDR returns multiple classifications, there are also multiple contexts for which these rules were created. Therefore the user would only have to consider one context for a given case when the multiple classification aspects were removed. The one draw back of using MCRDR in this manner is that, like RDR, rules may need to be repeated. Similarly, the knowledge base will be slightly larger than a traditional MCRDR knowledge base. Despite this problem, using MCRDR in this way was considered more beneficial, as it is less complex for the AI creator.

Rule Considerations. As with the typical MCRDR structure, a rule would need to incorporate a condition and a classification. Because the case is a game situation taken from the point of view of the character, the condition of a rule must specify what aspect of the game situation must hold true for that rule to fire. Therefore the AI editor allows the user to select conditions regarding the game situation that would return true or false. The case is only the game situation from the point of view of the character. Therefore conditions that can be selected in the AI editor cannot check for information that is not allowed to be known for that character. Apart from the condition, a rule must contain a classification that is an action that the character can perform. MCRDR only stores one action per rule. Therefore it can be seen that if the same case is given to the MCRDR engine, then the same action will always be returned. A desirable game feature is to have a character that is unpredictable [2, 21]. It can be seen that MCRDR does not support this feature. Therefore the MCRDR engine was modified to try and add an element of randomness. The way it works is that a rule does not have to store a single action as its classification, it can store several. When that rule is reached, then one of its actions is chosen at random. Returning more than one action from a rule can be thought of as similar to traditional MCRDR.

The main difference is that with this method, the same set of actions is always given for a case as only the one rule is ever returned. This method of action selection is random; however it is not completely emergent. This is because the actions of a rule are all relevant to the situation for which the rule was created and if multiple actions are not desired, they are simply not added. It was considered that the random actions could be weighted, so that certain actions have a higher chance of getting picked than others. This would be useful; however, it was discarded for this experiment as it was considered to add an extra level of complexity.

4.3 The Game and AI Editor

The Game. A game called "Forsaken Malice" was developed in C++, and was created for the purpose of this paper. The game allows for the ability to import and use the artificial intelligence created by the AI Editor. While the game is being played, the AI that has been imported does not change at all. The game is purely used to test and explore the AI from the editor. The game is a basic RTS game that allows the user to control their team either by selecting characters with the mouse and manually commanding them, and/or by creating AI for the characters with the AI editor. There are two teams, where each starts with ten characters and a factory that is a production plant for more characters. The only restriction on unit production is how long it takes to make a unit. The characters within this game environment can be one of five classes.

- Infantry – This unit has no special abilities and is the quickest to build at the factory.
- Mine Layer – Abilities such as firepower etc of the Mine Layer are reduced, however it can lay mines.
- Mine Sensor – The mines laid by enemy Mine Layers can be found with this unit.
- Bazooka – This unit runs slower, however it has greater firepower.
- Malice Warrior – Being the strongest unit in the game, the Malice Warrior has twice the abilities of the other units. The weakness of this unit is that it can be killed with one shot of the Bazooka unit.

Each character can perform actions. Following is a list of all the actions that can be performed by a character.

- Move – Simply move from one place to another.
- Attack – Attack a specified target.
- Follow – Follow a specified target.
- Move Attack – Move from one location to another. If an enemy is seen along the way, then attack them. Once the enemy is lost or killed, then resume to the specified location.
- Scout – The character progressively sweeps across the game area. This can be done aggressively (attack enemies if seen) or passively (ignore enemies).
- Patrol – A character continually Move Attacks between a specified source and destination.
- Nothing – The character does nothing and just stands idle.
- Lay Mine – If the character is a Mine Layer, then they possess the ability to lay a mine onto the game area.
-

The game is won when a team destroys all characters and the factory of the other team. The terrain is randomly generated, however, the elements are roughly the same and there is always a river that divides the game area into two halves. Each team starts on a different side of the river. There is also an indestructible bridge that is the only way to cross the river. A user can only see enemy units that are within the sight range of their units. An AI editor was also created that allowed the user to edit and create AI for the game. The user had to load or create a character's AI engine, then give this engine a case (which is a game situation). The AI engine could then evaluate this case. The user could then create a new rule for this case if they believed the currently returned action or actions were unsatisfactory. If a rule were to be created, then the user would have to create a condition and an action (or list of actions if desired) for the rule before it was added. The rule's condition was only excepted if it evaluated to true for the current case. Cases could be saved while the game was being played. This allowed the user to view and use cases in the AI editor later. A case in the AI editor could either be constructed manually or imported from the ones saved during the game. A case in the AI editor is represented textually. The AI created in the AI editor is team independent. This means that the AI can be used for the user's team, or it can be used for the computer's team. In this way, the user can play against their own or other people's AI if they wish.

The MCRDR process within the Game. Each character in the game receives a new action from their engine every time an event happens within the game. These events include when a character is produced at the factory, when a character is killed and when a character moves position. In the event that the engine does not return an action, then the character simply does nothing. Otherwise, the action retrieved from the engine is examined before it is applied to the character. If there are several actions stored within a rule, then a different one could be chosen each time that rule is reached. If a rule is reached several times in a row (eg when the game situation has not changed), then a different action will probably be returned each time. However, if the same situation occurs later in the game then any of the actions listed for a rule could be returned. After that check, the new action is compared to the previous action, where parameters are considered as well. For example, "Move to location x, y" is different from "Move to location x, z". In the previously mentioned situation where the new action is the previous action, it is still compared to itself as the parameter it points to may have changed (eg the "Any Attacker" parameter of "Attack Any Attacker" may not be the same attacker as when it was first called). The comparison is to see if the new action gives the same result as the current action. If the actions are different, then the new action is applied. If the new action gives the same result as the previous action, then the character's current state is checked. If the current state of the player is the same as the state the character would be in if they applied the new action, then the player is currently performing that action and it does not need to be reapplied. If the current state is different, then the previous action has either finished or could not be completed for some reason (eg the character's path is blocked etc), therefore the new action is applied again.

4.4 Experiments

The experiment consisted of recruiting subjects to use the AI editor in order to create AI for the game. All but two

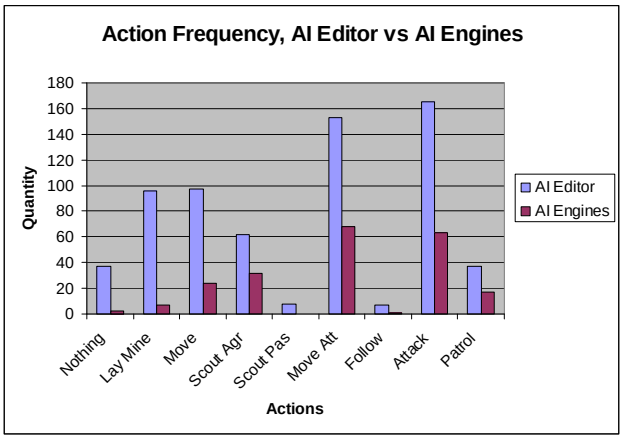
of the participants were university students, where most were doing degrees in computing or engineering. The subjects where given the user manual and offered a tutorial on how to use the system, although not all participants opted for the tutorial. The subjects were asked to return relevant files that were generated while the software was being used. From this, the returned files were examined. There were several files that were collected from the participants of the experiment when they returned their results. The first was the log file from the AI editor. This log file contained all information about anything that was done while the AI editor was being used.

5. Results

5.1 Selected Actions

From the results that were collected, the AI editor log files and the final AI engines were examined first. The actions that were selected by the participants were then inspected, followed by the selected conditions. When an action for a rule was being created in the AI editor, the selected action was written to the log file.
(Fig.2) Number and type of actions created with the AI Editor vs. the number and type of actions found in the final AI Engines

shows all the action types graphed against the number of times each one was created in the AI editor and the number of times each action type actually appeared in the final AI engines.

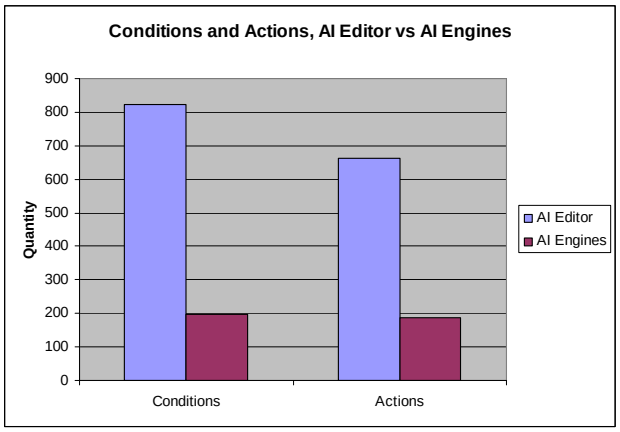


(Fig.2) Number and type of actions created with the AI Editor vs. the number and type of actions found in the final AI Engines

It can be seen from Fig. 2 that the most used action in the final AI engines was Move Attack. After this, the next most used action was Attack. If one considers the basic fundamental actions that can occur within the game, they would be to either move to a location or attack an object. It can be seen that the users preferred to use these basic commands than to use the more abstract higher-level commands such as Patrol or Scout. This could be because Patrol and Scout were the only higher-level commands and participants would have used the higher-level commands more often if there were more of them. Alternatively, people may feel more comfortable with the basic commands as they are the fundamental building blocks for strategy. If the participants were given more time to become more comfortable with the process, they may have ventured further and used the higher level commands.

6. Selected Conditions

In the Section 6.1, it could be seen that the number of actions created in the AI editor were greater than the number found within the actual AI engines. From Fig. 3 it can be seen that the same applies to conditions. Also from this figure, it can be seen that the ratio of AI editor to AI engine Boolean values (roughly 4:1) is higher than the ratio of AI editor to AI engine actions (roughly 3:1). Therefore the user created more Boolean values than actions in the AI editor compared to the final AI engines. This implies that the user may have had more trouble creating a condition for a rule than simply deciding on the action. The condition is used to determine what feature of a case must hold true for this rule to be activated. Therefore it would be naturally harder to create a condition than to create an action. Since this area is one of trouble for the user, the user-interface must be considered carefully as to not add confusion.

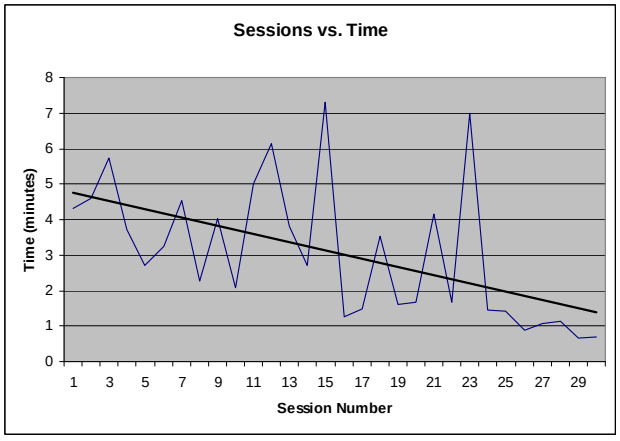


(Fig.3) The number of actions and conditions that were created in the AI Editor vs. the number of actions and conditions found in the final AI Engines
Knowledge Acquisition Activities

From all the AI engines that were gathered from the participants, there were 60 in total. From all of these engines, there were 251 rules. On average, each participant created 14 to 15 rules for their team, where the maximum number of rules for a participant was 40. Given more time, participants would have started to create more complex AI. If the participants were asked to swap their AI engines and use them for the computer team's AI, then the participants could then try and make their AI beat the other participant's AI. This would increase the complexity of the opponent, which would then lead to an increase of the number of rules. The reason that this would occur is because the participants would then be playing against a higher level of intelligence and therefore they would have to adapt their AI to compete against it.

To get an indication of the ease of AI creation, the session times were recorded and graphed. A session refers to when a user opens the AI editor, does something with it, and closed the AI editor. Fig. 4 shows the average time of the first 30 sessions from all the participants, where the left hand side is the first session. The data that was

used had the largest session time removed from each session. This was done because some users tended to have the occasional large session time. This could be attributed to reasons such as the user leaving the computer and doing something else while the AI editor was still running. Therefore the largest session time out of each session was removed to help prevent this from affecting the results. From Fig. 4, it can be seen that the session times are randomly distributed. However, when a trend line was introduced, it showed that the general trend was that the time taken to use the AI editor was dropping as the number of sessions increased. This can be attributed to several reasons. The user would start becoming used to the AI editor, and therefore become faster at the AI creation process. It can be seen from the graph that as the AI size increases (as it would when people are adding rules from one session to the next), the time it takes to create AI reduces. The benefit behind this is that it shows that the size of the AI does not affect the time it takes to create the AI. This shows that using MCRDR segments the AI into manageable pieces that do not need to be considered in context with the rest of the system.



(Fig.1) The number of actions and conditions that were created in the AI Editor vs. the number of actions and conditions found in the final AI Engines.

7. Conclusions

It can be seen that AI can be personalize d using the MCRDR method used in this study. The user-interface must be enhanced to emphasize the case-based approach and to make the AI creation process more intuitive. In similar respects, a more graphical representation of a case would help to increase a user's understanding of the game situation. This would show the user what was happening when that case was created. A programmer implementing this method into a game must decide whether they wish to allow the user to create AI from nothing, or only allow them to extend the pre-programmed base AI. If the user is allowed to create his/her own base AI, then it is probably a good idea to give the user an opportunity to create it with some other general form of AI creation. This would allow the user to create a base AI and then allow them to refine it with the MCRDR system. If a base AI was included rather than allowing the user to create their own, it would be useful to have some way to find out what that base AI was. It can be seen that the participants were finding the important aspects of the game by focusing on similar areas when creating their AI. The strategies were, however, different and this shows that people were personalizing the AI to what they believed to be useful. This showed that MCRDR could be used to create AI and personalize it to what a user wanted. Participants mentioned that this system for AI personalization allowed them to segment the system and only focus on was important. This was one of the reasons why MCRDR was chosen. They said that this project was interesting, allowed them to explore various strategies and was fun. This indicates that there is definitely an interest for personalizing AI.

References

[1] Laird, J.E., Using a Computer Game to Develop Advanced AI. Computer, 2001. 34(7): p. 70-75.

[2] Watt, A. and F. Policarpo, 3D Games: Real-Time Rendering and Software Technology. Vol. 1. 2000, NY: Addison Wesley.

[3] DeLoura, M., Game Programming Gems 2 (Game Programming Gems Series). 2001, Rockland, Massachusetts, USA: Charles River Media, inc.

[4] Sweetser, P. and J. Wiles, Scripting versus Emergence: Issues for Game Developers and Players in Game Environment Design. International Journal of Intelligent Games and Simulations, 2005. 4(1): p. 1-9.

[5] Woodcock, S., Game AI: the state of the industry, in Game Developer. 2000.

[6] Treglia, D., Game Programming Gems 3. Game Programming Gems Series, ed. D. Treglia. 2002, Hingham, Massachusetts: Charles River Media.

[7] Laird, J.E. and P.S. Rosenbloom, The Evolution of the Soar Cognitive Architecture, in Mind Matters: A Tribute to Allen Newell, D.M. Steier and T.M. Mitchell, Editors. 1996: Erlbaum, Mahwah, NJ. p. 1-50.

[8] Kendall, G. and K. Spoerer. Scripting the Game of Lemmings with a Genetic Algorithm. in Congress on Evolutionary Computation 2004 (CEC'04). 2004. Portland, Oregon.

[9] MacNaughton, M., et al. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. in 19th IEEE International Conference on Automated Software Engineering (ASE). 2004. Linz, Austria.

[10] Richards, D. and P. Compton. Combining formal concept analysis and ripple down rules to support reuse. in Software Engineering Knowledge Engineering SEKE'97. Madrid: Springer-Verlag.

[11] Preston, P., et al. An implementation of multiple classification ripple down rules. in 10th AAAI-

Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. 1996. Banff, Canada, University of Calgary.

[12] Kang, B., P. Compton, and P. Preston. Simulated Expert Evaluation of Multiple Classification Ripple Down Rules. in 11th Banff knowledge acqusition for knowledge-based systems workshop. 1998. Banff, Canada: SRDG Publications, University of Calgary.

[13]Kang, B., P. Compton, and P. Preston. Multiple Classification Ripple Down Rules : Evaluation and Possibilities. in 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. 1995. Banff, Canada, University of Calgary.

[14]Kang, B.H., W. Gambetta, and P. Compton, Verification and validation with ripple-down rules. International Journal of Human-Computer Studies, 1996. vol.44, no.2: p. 257-269.

[15]Compton, P., et al., Knowledge acquisition without analysis. Knowledge Acquisition for Knowledge-Based Systems. 7th European Workshop, EKAW '93 Proceedings, 1993: p. 277-299.

[16]Richards, D., Combining cases and rules to provide contextualised knowledge based systems. Modeling and Using Context. Third International and Interdisciplinary Conference, CONTEXT 2001. Proceedings (Lecture Notes in Artificial Intelligence Vol.2116), 2001: p. 465-469.

[17]Compton, P. and R. D. Extending Ripple-Down Rules. in 12th International Conference on Knowledge Engineering and Knowledge Managements (EKAW'2000). 2000. Juan-les-Pins, France.

[18]Pinter, M., Toward More Realistic Pathfinding, in Game Developer Magazine. 2001.

[19]Rabin, S., AI Game Programming Wisdom. 1 ed. Vol. 2. 2004: Delmar Learning - ITP.

[20]Patel, A.J., Amit's Thoughts on Path-Finding and A-Star. 2006.

[21]Rousa, R., Game Design: Theory & Practice Second Edition. 2001, Texas, America: Wordware Publishing.

Authors

Stephen Burman

2000 – 2002 University of Tasmania, Australia (BA)

Yang Sok Kim

1987-1995 University of Seoul
1994-2001 Hyundai Information Technology Co., Ltd
2001-2002 E -2 Corporation
2002-2004 University of Tasmania (Master of Computing)
2005. 2- Now University of Tasmania, Australia, PhD Student

Byeong Ho Kang

1982-1988 Pusan National University
1988-1990 University of Tasmania, Australia (Master of Computing)
1990-1995 New South Wales Univ., Australia (PhD)
1995-1996 Hitachi Advance Research Lab, Japan, Research Fellow

1996-1999 Hoseo University, Assistant Professor
2000-Now University of Tasmania, Australia, Senior Lecturer

Gil-Cheol Park

1979-1983 HanNam Univ.(BA)
1983-1985 SungSil Univ., Graduate School(MA)
1994-1998 SungKunKwan Univ., Graduate School(Ph.D)
1985-1990 SamSung Advanced Institute of Technology

1991-1996 DaeKyo Computer Co., LTD.
1996-1998 HanSeo University, Professor
2005 Visiting Professor of Tasmania State Univ., Australia
1998- Present, HanNam Univ., Professor.
Concerning and Interesting Recent Research Area
-Mobile & Ubiquitous Web Service platform
-Real-time Multimedia Communication.
-Security Engineering