CREATING REACTIVE NON PLAYER CHARACTER ARTIFICIAL INTELLIGENCE IN MODERN VIDEO GAMES

Leif Gruenwoldt, Michael Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada
E-mail: lwgruenw@gaul.csd.uwo.ca, katchab@csd.uwo.ca

Stephen Danton
Horseplay Studios
Seattle, Washington, USA
E-mail: stephen_m_danton@hotmail.com

KEYWORDS

Reactive artificial intelligence, relationship modeling in video games, reputation systems

ABSTRACT

Realistic and reactive non player characters that respond appropriately to activity in their game world would be welcomed by game developers and players alike. Unfortunately, despite significant progress made by in this area of research, this goal has still yet to be fully achieved.

In this paper, we present a new Realistic Reaction System, based on our previous work in this area, which provides reactive artificial intelligence through the use of an underlying relationship system that binds together players, non player characters, and objects in the game world. Through proper maintenance, manipulation, and querying of the relationship system, we can effectively and efficiently augment the decision processes used in artificial intelligence controllers in games to provide a richer and more immersive experience to players.

INTRODUCTION

Game developers have long sought to have meaningful and logical interactions between human players and non player characters driven by their games' artificial intelligence. Unfortunately, except for restricted circumstances, typically violent confrontations, this has not materialized successfully, leaving the players feeling isolated or disconnected from the world in which they are playing (Laramée 2002).

A key element to overcoming this problem is the development of artificial intelligence capable of dynamically reacting to players and player actions in reasonable and realistic fashions. Doing so would require a sense of relationship or social network binding the characters and objects in the game world to one another, a sentiment expressed in (Lawson 2003) and elsewhere. Without this, developers have to rely upon static or scripted methods of implementing behaviours and events to mimic realistic character reactions, which is ultimately quite limiting.

Work in developing reputation systems for games has drawn some attention recently in an attempt to address this problem. For instance, the games Ultima Online (Grond and Hanson 1998) and Neverwinter Nights (Brockington 2003) provide reputation systems, but do so with some restrictions and drawbacks; for example, in Ultima Online, reputation changes have unrealistic immediate global effects, regardless of who witnessed the actions precipitating the changes. The work in (Alt and King 2002) shows promise, but requires more flexibility and generality; for example, it supports only a limited relationship set, does not track relationships between non player characters, and has not been applied to game world objects and complex group situations.

In (Gruenwoldt 2005), we introduced a Realistic Reaction System (RRS) for modern video games to overcome these issues. This system models and maintains the relationships between players, non player characters, and objects in the game world over time dynamically, and provides methods by which characters can query the relationship network to formulate appropriate reactions in behaviour, dialogue, and so on. While this served as an important first step in providing reactive characters in games, more work was needed to fully address the problem at hand.

Our current work builds upon this previous work from (Gruenwoldt 2005) to provide a complete and integrated solution to the above problems. In particular, in this paper, we focus on the logic and mechanisms required to link the relationship system from our previous work with artificial intelligence controllers to create reactive non player characters. Doing so effectively poses significant challenges, as a well populated world with a rich collection of relationship types can produce a relationship network large enough to overwhelm both artificial intelligence programmers and scarce game resources at runtime. This was learned first hand in integrating our previous work into the action/adventure/role-playing game Neomancer (Danton 2004; Katchabaw 2005) that we are currently co-developing. Consequently, mechanisms must be in place to help manage, filter, and aggregate relationship information appropriately according to the needs of the game in question.

This paper presents a new extended architecture for RRS for creating reactive non player character artificial intelligence, and discusses our work in implementing and using it to date. We begin with a brief overview of relationships and augmenting artificial intelligence controllers with this information. Following this, we

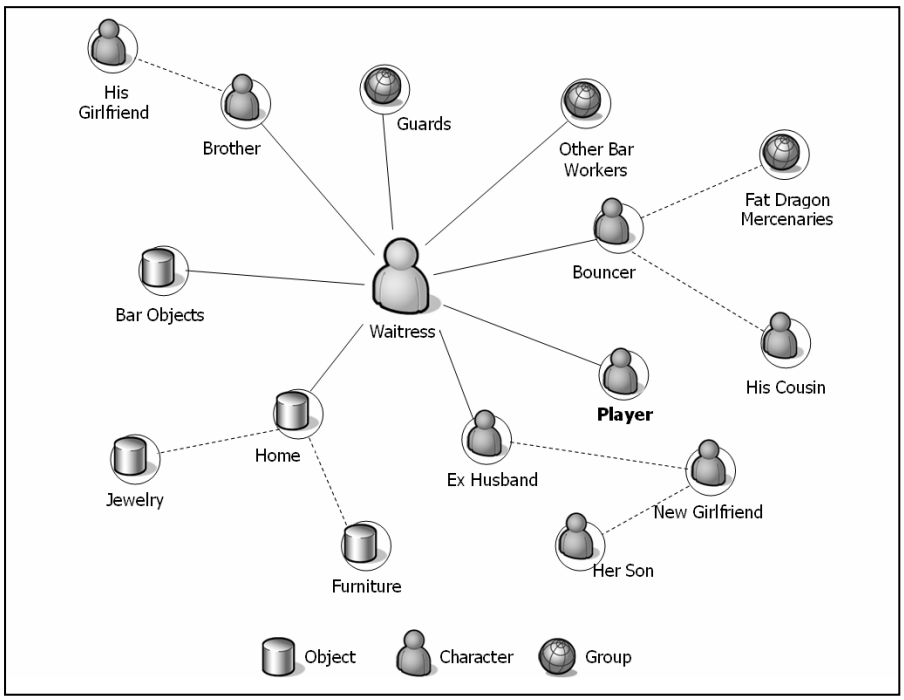

Figure 1: Example Relationship Network from Neomancer

provide architectural details of our new approach, and outline its implementation using Epic's Unreal Engine (Epic Games 2004). We then discuss our experiences with using this new system both in the context of an Unreal game mod (Castaneda 2005), and our Neomancer project (Danton 2004; Katchabaw 2005), currently under development. We finally conclude the paper with a summary, and a discussion of directions for future work in this area.

RELATIONSHIP MODELING AND USE IN GAME ARTIFICIAL INTELLIGENCE

Before examining the details of the newly extended RRS, we first provide background on modeling and manipulating relationship data for use in games, and how we can augment artificial intelligence controllers using this information.

Relationships for Games

A relationship network models all of the relationships between all of the characters, groups of characters, and objects of interest in the game world. One can envision this network as a graph-like structure, with the characters, groups, and objects as nodes in the graph, and the various relationships that exist between them as edges (directed or undirected, depending on the relationship). An example of this kind of relationship network from the Neomancer game project (Danton 2004; Katchabaw 2005) is presented in Figure 1.

There are numerous possible types of relationships that exist between entities in the relationship network. Each of these types can have subtypes, and so on, resulting in a hierarchical tree of relationship types. For example, main types of relationships can include: emotional, familial, business, leadership, ownership, membership, and so on. If we were to expand the membership branch, for example, there exist relationships to denote belonging to groups in the game, such as ethnicity, social caste, profession, community residence, and so on. This hierarchy can be easily expanded with additional types and sub-types as necessary.

Furthermore, each relationship has several attributes. These attributes include origin, history, regularity, strength, polarity, and validity. Relationship-specific attributes can also be assigned where appropriate.

Relationships can be affected in numerous ways. The most direct method is by filtered and processed game events. In other words, when one entity in the game world observes the actions of another, those actions can directly impact the relationships between those two entities, and the appropriate relationships must be added, updated, or removed. Relationships are also affected by game events indirectly, by their propagation through the relationship network. Depending on the nature of the event and how it affects entities in the network directly, the event can be felt by other related entities. Time also affects relationships. Given enough time, relationships drift towards a neutral state, in the absence of events or interactions that would otherwise act to strengthen them.

Augmenting Artificial Intelligence Controllers with Relationship Data

We now examine how to examine how to augment artificial intelligence controller using relationship data from a relationship network. Since scripting, state machines, and rule-based systems are the most widely used techniques in implementing game artificial intelligence (Champandard 2004), and scripting tends to be too static to be truly reactive, we will focus our attention on these last two techniques in this paper.

Generally, an artificial intelligence controller can be augmented to use relationship data in its decision process by querying the relationship network and using this data as additional game state to regulate state transitions or rule firings. In a state machine, this will require additional specialized transitions and/or specialized states; in a rule-based system, this will require additional rules with specialized firing conditions.

Care must be taken in using this relationship data, however. Using raw relationship information will result in an explosive increase in the size and complexity of state machines or rule systems using it, because the number of possible relationships and relationship attribute values could be quite large. This would make programming artificial intelligence for games quite difficult and tedious. If relationship information, however, was filtered and aggregated, much of this complexity can be removed, resulting in state machines and rule systems that are more manageable and easy to use.

Augmenting State Machines with Relationship Data

Consider, for example, the fragment of a state machine shown in Figure 2 for a guard in the guards group depicted in the relationship network in Figure 1.

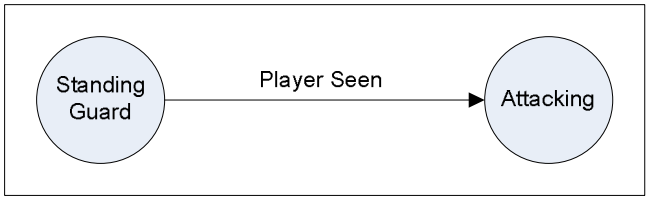

Figure 2: State Machine Fragment without Relationship Data

When the player is observed, this event causes the guard to switch to an attacking state to attack the player, regardless of the player's prior activities or relationship to the guard in question. While this reaction might appear realistic if the player's behaviour warranted an attack, it would appear oddly out of place if not. Such out of place behaviour can break the player out of immersion, and have a detrimental effect on the player's enjoyment of the game (Bates 2004; Rouse 2004).

Now consider the state machine fragment that is shown in Figure 3, based on the previous state machine. In this case, the relationship data between the player and guard in question has been distilled and aggregated for simplicity into one of three possibilities, negative, positive, and neutral, resulting in three possible transitions from a new state in the machine. (This new state can be avoided if an extended state machine is used that can have transitions triggered by multiple events or pieces of state information.) This allows player behaviour to influence the relationship with the guard, and the guard's reaction to the player as a result.

Suppose that guards in the guards group that was shown in Figure 1 are friends with the waitress that is the focal point of that relationship network. If the player hurts the waitress, the guards would have a negative view of the player and react with hostility towards the player. If the player, on the other hand, was helpful to the waitress, the guards would have a positive view of the player and help the player in return. With no prior contact with the waitress, the guards would have a neutral view of the player, and act accordingly.

Consequently, using relationship data, we can now have artificial intelligence that reacts in a realistic fashion determined by player behaviour. This results in a better overall experience to the player, as the player is left with the impression that his or her actions actually have an impact on the game world and its inhabitants (Bates 2004; Rouse 2004).

Augmenting Rule-Based Systems with Relationship Data

In this section, we examine augmenting rule-based systems with relationship data, much like state machines were in the previous section. Using the situation calculus notation of (Russell and Norvig 2003), we can express the original state machine fragment given in Figure 2 using a

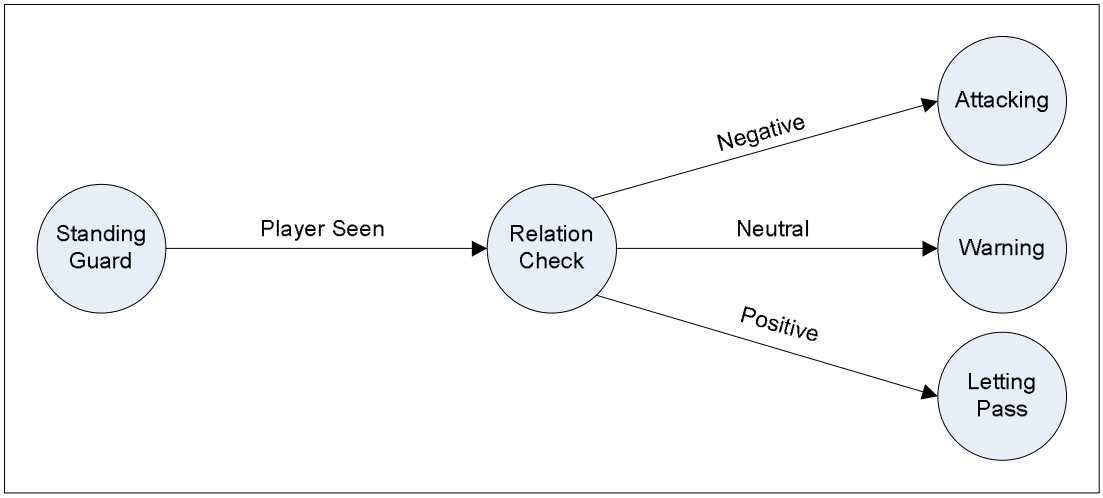

Figure 3: State Machine Fragment with Relationship Data Considered

rule system quite easily. Note that for simplicity, we are omitting effect axioms from the rule systems given in this paper. In the end, we are left with the rule system presented in Figure 4.

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \Rightarrow Poss(Attacks(Guard,Player),s) \quad (1)$$

Figure 4. Rule System Fragment without Relationship Data

The rule system in Figure 4 consists of one very simple rule. In essence, this rule states that if the guard is standing guard in a given situation and sees the player, then it becomes possible for the guard to attack the player from that situation. This is a fairly direct translation of the state machine from Figure 2.

Adding distilled and aggregated relationship data to the rule system in Figure 4 is quite straightforward. Following the example in the previous section, we augment this rule system with negative, positive, and neutral relationship data, resulting in a system with more specialized rules, as shown below in Figure 5.

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Negative) \Rightarrow Poss(Attacks(Guard,Player),s) \quad (1)$$

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Neutral) \Rightarrow Poss(Warns(Guard,Player),s) \quad (2)$$

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Positive) \Rightarrow Poss(LetsPass(Guard,Player),s) \quad (3)$$

Figure 5: Rule System Fragment with Relationship Data Considered

With three possible aggregated relationship states, the rule system in Figure 5 now must have three rules, each with a specialized firing condition corresponding to one of the possible states. Rule 1 is fired from a situation in which the guard is standing guard, sees the player, and has a negative relationship with the player. In this case, it is now possible for the guard to attack the player. Rule 2 is fired from similar circumstances, except that the guard has a neutral relationship with the player. In this case, the player is only given a warning. Lastly, Rule 3 is fired when the guard has a positive relationship with the player, and lets the player pass as a result. With these three rules in place, the artificial intelligence can be more reactive to player behaviour and produce results that are more in line with player expectations.

More Advanced Use of Relationship Data

As can be seen from the examples in previous sections, augmenting state machines and rule-based systems with relationship data is not difficult when aggregated relationship data is used. If raw data were to be used instead, however, both techniques could become significantly more complex, as discussed earlier.

That said, the use of raw relationship data can result in more advanced artificial intelligence controllers for non player characters that are able to fine tune their response to particular situations using more specific relationship data. This would result in even more realistic reactions and a better overall player experience.

Continuing the previous examples, if the player has a lucrative financial arrangement with the guard in question, the guard may still let the player pass, even if the guard has an overall negative or neutral opinion of the player. If we were to add this logic to the rule system from Figure 5, we can do so by adding the new specialized rules in Figure 6. (Other specialized rules may need to be added or used to replace existing rules for completeness, but the rules provided in Figure 6 suitably illustrate what would be required.)

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Negative) \wedge RelationshipExists(Guard,Player,Financial) \wedge FinancialRelationshipValue(Guard,Player,High) \Rightarrow Poss(LetsPass(Guard,Player),s) \quad (4)$$

$$StandingGuard(Guard,s) \wedge SeesEntity(Guard,Player) \wedge RelationshipAggregate(Guard,Player,Neutral) \wedge RelationshipExists(Guard,Player,Financial) \wedge FinancialRelationshipValue(Guard,Player,High) \Rightarrow Poss(LetsPass(Guard,Player),s) \quad (5)$$

Figure 6: New Rule System Fragment with More Advanced Relationship-Based Decisions

Naturally, similar extensions can be made to state machines, such as the one shown in Figure 3, to capture such behaviour as well. This will again result in additional states and/or transitions, but would result in a more sophisticated and realistic controller.

In the end, we have the ability to construct both simple and specialized artificial intelligence controllers using relationship data as shown in the examples given above. This allows us to trade off complexity for expressiveness when required to suit the needs of the game in question.

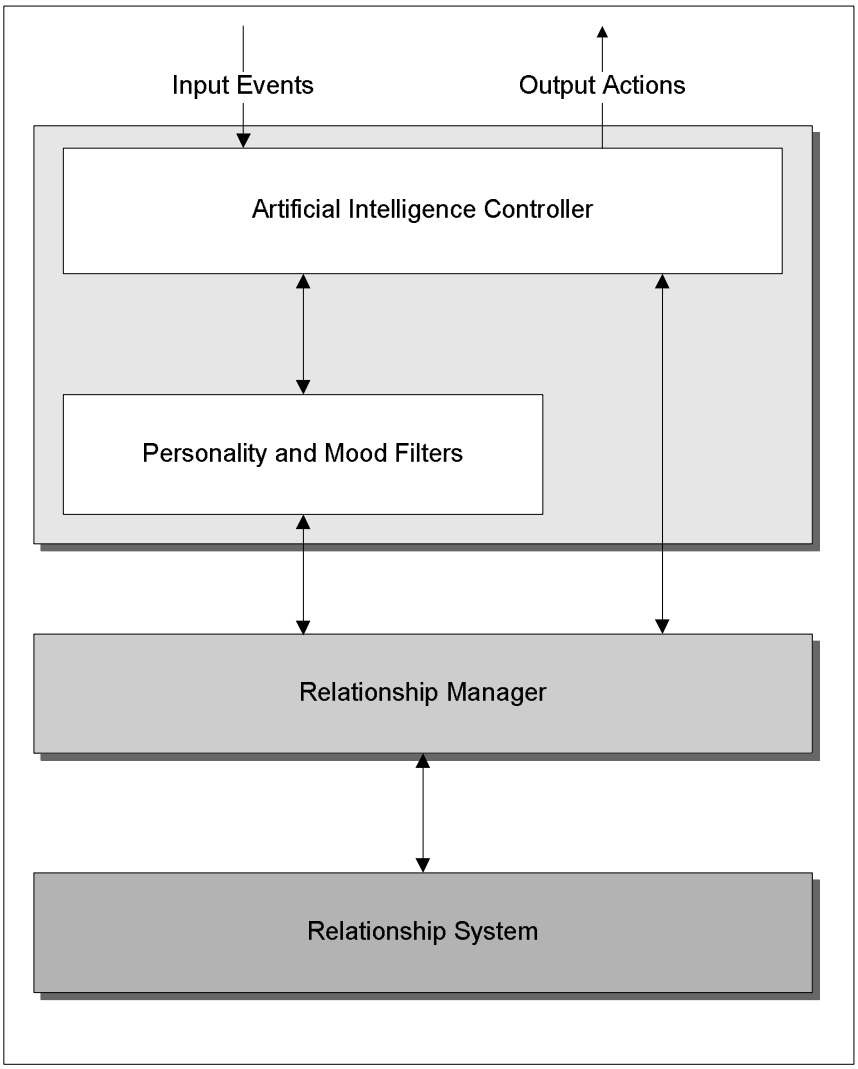

Figure 7: Reactive Artificial Intelligence Architecture

ARCHITECTING REACTIVE ARTIFICIAL INTELLIGENCE

To create reactive non player character artificial intelligence, we have developed the new architecture for RRS shown in Figure 7. The various elements of this architecture will be discussed in detail in the subsections below.

Relationship System

The relationship system is used to maintain and provide access to the relationship network constructed in the game. It provides raw access to this relationship data to the relationship manager, which can then provide a more specialized interface to simplify data access. For details on the internal design of the relationship system and its relationships and relationship network, the reader is urged to consult (Gruenwoldt 2005) for more information.

Relationship Manager

A relationship manager provides an interface to relationship data from the relationship system to one or more non player characters in a game. Typically, a relationship manager provides filtered or aggregated access to this data to simplify accessing and dealing with relationships in the artificial intelligence controllers of the corresponding characters. For example, the augmented artificial intelligence controller logic that was presented earlier in this paper made use of relationship data aggregated into three possible values: positive, neutral, and negative. If the controllers had to make use of raw relationship data, they would be orders of magnitude more complex to deal with the large number of possible relationships and relationship attributes as was discussed. This was learned first hand in directly using relationship data from the first prototype of RRS from our previous work (Gruenwoldt 2005) in constructing new artificial intelligence controllers; as the relationship system grew more robust and more expressive, the controllers grew in complexity quickly to the point where they became exceedingly difficult to program.

Depending on the needs of the game and its non player characters, relationship managers can aggregate data in a number of ways. A manager can provide a single relationship measure, or multiple relationship measures, and these measures can of a variety of types. For example, instead of providing positive, neutral, or negative values to the controllers presented earlier in this paper, a relationship manager could provide a numeric hostility score with

different values resulting in different transitions or rule firings. While providing a filtered or aggregated view of relationship data, a relationship manager must also still provide access to raw relationship data to allow the construction of more robust artificial intelligence controllers able to respond to more specific situations, as discussed in the previous section.

No constraints are imposed on how filtering and aggregating are to take place in this architecture; the selection of algorithms and heuristics is up to the implementer, as this process can be game specific. Filtering and aggregating in a relationship manager can occur when relationship changes are submitted to the relationship system, to have this data pre-computed for when queries are made, or in an on demand basis, when queries for data are actually received. Which of these approaches performs better likely depends on the game in question and the mix of relationship changes versus queries; fortunately, since this process is encapsulated within each relationship manager, a relationship manager can tune its own behaviour at run-time to provide the best overall results and performance without affecting the rest of the system.

In the end, we take an object-oriented approach to relationship managers, providing a base framework from which game-specific managers can be derived. In fact, game-specific managers can be further derived to allow variations from one class of non player characters to another, or even to the point of having specific managers for specific characters within the game, if required. This allows for a great deal of flexibility in filtering and aggregating data for use in a game.

Artificial Intelligence Controller

The artificial intelligence controller provides the core decision making functionality for a particular non player character in the game, accepting input events and formulating appropriate actions in return. As discussed earlier in this paper, in a gaming environment, this module will likely be driven by a state machine or rule-based system of some kind. For more details on controller design and implementation, (Champandard 2004) serves as a good reference.

Personality and Mood Filters

Personality and mood filters are used to provide further customization and specialization to artificial intelligence controllers to allow for more varied non player characters. These filters work by modifying the way input events are processed into relationship changes or by modifying results from relationship queries before they are processed by the controllers. This lets different characters record and retrieve relationship data differently, allowing different behaviours even when the characters are driven by fundamentally the same controller. This also allows character behaviour to be tuned over time as their personality develops throughout the game, or as their moods shift. As necessary, however, these filters can also be bypassed by the artificial intelligence controller, to provide direct contact with the relationship manager for the character.

Different personality traits can be parameterized and used in these filters, such as intelligence, aggressiveness, attentiveness, disposition, prejudices, and so on. For example, a character that is generally pessimistic could have its relationship changes adjusted more negatively than a character would have otherwise. Different moods and emotional states can also be used in filters for similar effects. For example, a character that is in an exceptionally good mood at a given time could have results of its relationship queries modified in a positive fashion, causing it to act better towards other characters it might not have otherwise.

IMPLEMENTATION AND EXPERIENCE

A prototype of the new RRS architecture from the previous section has been developed for Epic's Unreal Engine (Epic Games 2004) in UnrealScript. UnrealScript has many of the features of a traditional object-oriented language, providing excellent support for extensibility for the future. Games built on the Unreal Engine can take advantage of this system by either extending a new game type and new pawn and controller classes, or by embedding the appropriate hooks into existing game code. In addition, our earlier work with the Unreal Engine provided additional console commands to support manipulation of relationships manually from within the game (Gruenwoldt 2005). This allows game developers and designers to add relationship information during production from within the game itself, allowing easy debugging and initialization of content.

Artificial intelligence controllers in the Unreal Engine are in essence state machines, allowing them to be augmented with relationship data as described earlier in this paper. Using the existing relationship system, relationships, and relationship network from (Gruenwoldt 2005) as a foundation, a new relationship manager was derived from its base class to aggregate this relationship data together using a collection of simple heuristics. A small number of personality and mood filters were also developed to provide more varied tuning of relationship changes and query responses. As simple examples, optimistic and pessimistic filters were implemented to adjust the positivity and negativity of relationship changes respectively when required within a game.

After development, initial validation of the new RRS took the form of individual test cases. More extensive validation took the form of modifying the existing LawDogs game modification to Unreal Tournament 2003/2004 (Castaneda 2005). LawDogs was chosen primarily because its setting included a bar scene, which follows in line closely to the relationship network example presented in Figure 1, and introduced originally in (Danton 2004). LawDogs also had reasonably simple gameplay with straightforward and traditional artificial intelligence, making it a suitable first deployment for our work. Our experience with LawDogs demonstrated that the non player characters in the game were able to react according to player interactions quite well.

Figure 8: Screenshot from Neomancer with Test Character

Based on this success, we are currently augmenting the artificial intelligence developed for the Neomancer project (Danton 2004; Katchabaw 2005), an action/adventure/role-playing game being co-developed by the University of Western Ontario and Seneca College. It features richer characters and gameplay than LawDogs, and is better suited towards larger scale deployment and testing of our current work.

We have encountered similar success with test non player characters in Neomancer using the new RRS, and are currently expanding use of the system as the artificial intelligence controllers and characters in the game continue to be developed, refined, and enhanced. (The Neomancer project is expected to take up to three years to complete, and we have only completed the initial year of the project.) A screenshot of a test character in the Neomancer world is shown in Figure 8.

CONCLUDING REMARKS

By capturing game relationships and facilitating more appropriate character responses through linking relationship data to artificial intelligence controllers for non player characters, our new Realistic Reaction System can provide more immersive and compelling gameplay in modern video games efficiently and effectively. In the end, non player characters will be able to react to player behaviour in a more realistic fashion, leading to a better overall gameplay experience for the player.

Experimentation with an Unreal-based implementation of this system to date has proven successful, both in small and mid-size deployments of the system. Larger scale deployment in a commercial-grade game is currently underway and progress to date has been excellent. This new Realistic Reaction System demonstrates great promise for future development efforts.

In the future, there are many possible directions for research to take. This includes the following:

- We plan to complete our current development and deployments efforts with Neomancer and port the new RRS to other games and platforms for further research and development.

- To meet stringent performance constraints we further plan to investigate techniques to optimize RRS and minimize run-time overhead in manipulating and querying relationships in the system. While we have found that proper filtering and aggregation in relation managers can greatly improve performance, additional performance improvements would always be quite beneficial.

- Finally, we also intend to extend our library of relationships, relationship managers, and personality and mood filters to allow RRS to support a wider variety of artificial intelligence behaviours in games by default.

REFERENCES

G. Alt and K. King. "A Dynamic Reputation System Based on Event Knowledge". *Appeared in AI Game Programming Wisdom.* Charles River Media. 2002.

B. Bates. *Game Design.* Second Edition. Thomson Course Technology. 2004.

M. Brockington. "Building a Reputation System: Hatred, Forgiveness and Surrender in Neverwinter Nights." *Appeared in Massively Multiplayer Game Development*. Charles River Media. 2003.

G. Castaneda, et al. LawDogs UT2003-UT2004 Modification. *Available from project home page online at http://www.planetunreal.com/lawdogs.* February 2005.

A. Champandard. *AI Game Development*. New Riders Publishing. 2004.

S. Danton. *Neomancer Game Design Document*. Horseplay Studios Technical Report. November 2004.

Epic Games. *Unreal Engine 2, Patch-level 3339.* November 2004.

G. Grond and B. Hanson. "Ultima Online Reputation System FAQ". *Origin Systems Technical Document (available at http://www.uo.com/repfaq).* 1998.

L. Gruenwoldt, M. Katchabaw, and S. Danton. "A Realistic Reaction System for Modern Video Games". In *the Proceedings of the DiGRA 2005 Conference: Changing Views – Worlds in Play*. Vancouver, Canada, June 2005.

M. Katchabaw, D. Elliott, and S. Danton. "Neomancer: An Exercise in Interdisciplinary Academic Game Development". *In the Proceedings of the DiGRA 2005 Conference: Changing Views – Worlds in Play*. Vancouver, Canada, June 2005.

F. Laramée. "Nine Trade-Offs of Game Design". *Appeared in Game Design Perspectives*. Charles River Media. 2002.

G. Lawson. "Stop Relying on Cognitive Science In Game Design - Use Social Science". *In Gamasutra Letter to the Editor (available online at http://www.gamasutra.com/php-bin/letter_display.php?letter_id=647).* December 2003.

R. Rouse III. *Game Design: Theory and Practice*. Second Edition. Wordware Publishing, Inc. 2004.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Second Edition. Prentice Hall. 2003.