A Realistic Reaction System for Modern Video Games

Leif Gruenwoldt, Michael Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada
Tel: +1 519-661-4059
lwgruenw@gaul.csd.uwo.ca, katchab@csd.uwo.ca

Stephen Danton
Horseplay Studios
Seattle, Washington, USA
Tel: +1 206-956-4689
stephen_m_danton@hotmail.com

ABSTRACT

The substantial growth of the video game industry has fueled a search for new technologies and methodologies for providing rich and rewarding experiences for players of modern video games. Many of the most popular games offer visually rich and compelling environments to support a higher level of believability and immersion for the players. Recent generations of games have also offered great advancements in areas like realistic physics, engaging audio, and believable artificial intelligence.

Our current work, however, focuses on oft-overlooked and neglected area of development– providing societal-like relationships between the characters and objects of the game world. A dynamic and reactive relationship system opens up new directions for interaction within a game world to be explored. In this paper, we discuss our work on the development of a realistic reaction system to support relationship modeling and representation in modern video games, and outline our experiences in using it to date.

Keywords

Relationship modeling, relationship networks, relationships in video games

INTRODUCTION

With the explosive growth in the game industry, game developers are constantly seeking new methods of creating a more immersive and believable gaming experience for players of their games, both to remain competitive and to provide more satisfaction and enjoyment to their players. This has led to the development of more life-like graphics and audio systems, improved individual artificial intelligence, realistic physics engines, and a variety of other technologies that have greatly advanced the state-of-the-art in video games.

One thing that modern video games still lack, however, is a sense of relationship or social network binding the characters and objects in the game world to one another. This sentiment is expressed eloquently in [5], and elsewhere. Without this, game developers have to largely rely upon scripted behaviours and events to mimic realistic character reactions to events that occur in the game world. Since a developer can only script so much, and a game is stuck with whatever scripts it ships with, this method is ultimately limited. Consequently, players often sense a disconnection in the game world that leads to a break in immersion and a loss of believability.

To address this issue, our current work investigates the development of a Realistic Reaction System (RRS) for modern video games. This system models and maintains the relationships between the various 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. In the end, RRS provides game characters the information they need to respond appropriately to the situations with which they are presented.

This paper presents our initial work in developing and using RRS. We begin with a general overview of relationships and relationship networks in general. Following this, we provide architectural details of RRS, and outline its implementation using Epic's Unreal Engine [3]. We then discuss our experiences with using RRS in an Unreal game mod [1], and its use in our Neomancer project [2,4], currently under development. We finally conclude the paper with a summary, and a discussion of directions for future work in this area.

RELATIONSHIPS AND THE RELATIONSHIP NETWORK

The relationship network forms the infrastructure for RRS. This 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).

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.

With the relationship network, relationship types, and relationship attributes, RRS has a great deal of expressive power at its disposal. Consider the example relationship network shown in Figure 1 below.

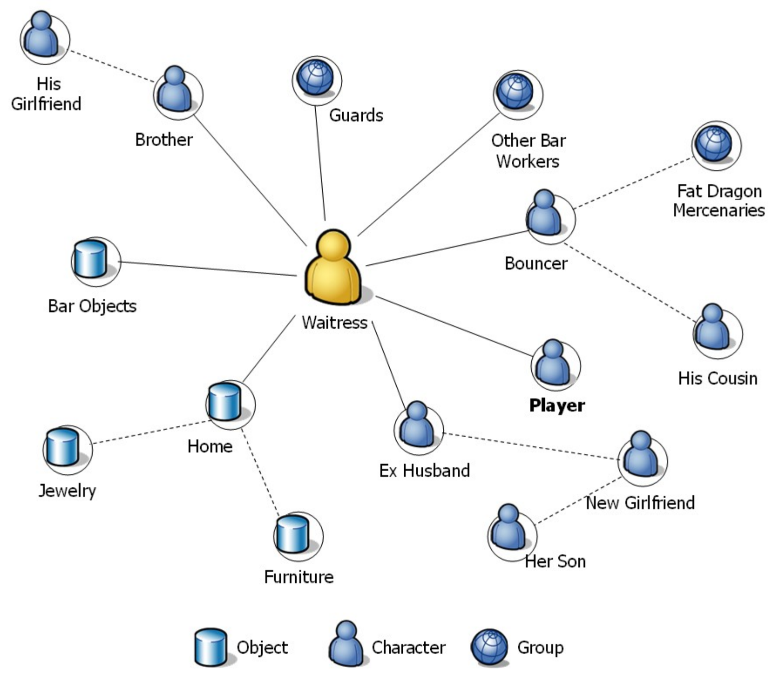


Figure 1: Example relationship network

In this example, the focal point of the network is a waitress working in a bar. She has direct relationships established between various objects, characters, and groups in the game world, such as objects in the bar, other workers in the bar, her home, her brother, and the game player. Through these entities, she has indirect relationships with other entities, such as her brother's girlfriend. Each of these relationships can have types and values. As one example, she might find the jewelry and furniture in her home to be more valuable than bar objects, as she is in an ownership relationship with the objects in her house, but not with those in the bar. As another example, she might have emotional relationships with both the bouncer in the bar and her ex-husband, although she might feel affection for the bouncer and not feel very fondly toward her ex-husband.

Affecting Relationships

Relationships in the relationship network can be affected in numerous ways. The most direct method is by the filtered input of game events into the network. Once they are passed on to the relationship network, events cause either the creation of new relationships between entities, the replacement of one relationship by another, the modification of existing relationship attributes, or

the removal of a relationship from a system. To maintain history, however, it is likely best to mark a relationship as invalid instead of completely removing it from the network.
Revisiting the example scenario depicted in Figure 1, a patron could become disruptive in the bar and damaging the tables and other objects in the bar. This would create a rather negative relationship with the waitress, due to her standing relationship with the bar objects. If her ex-husband came to her rescue and expelled the unruly patron, this could change or replace the waitress's relationship with him to a more positive relationship, and might even affect the budding relationship she was building with the bouncer.

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. How this indirect influence affects other entities depends on the entities in question and the relationships that exist between them. Propagation can occur when two entities directly interact with one another, for example in two characters having a dialog. Propagation can also spread along relationships without direct contact over time depending on the initiating event, much in the same way that reputation and notoriety spread throughout a community.

As an example of propagation, we revisit the example scenario in Figure 1 once more. If we suppose that the Fat Dragon Mercenaries, a group with which the bouncer is associated, begin to terrorize the residents of the town in which the bar is situated, news of this could propagate back to the waitress. Since the bouncer is associated with this group, her relationship with him could be changed for the worse, or replaced with a different one entirely, if we suppose that the mercenaries killed her brother in the process.

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. Relationships, in essence, must be fed and nurtured for them to endure.

Querying Relationships and Formulating Reactions

In order to produce effective responses within a game, the relationship network within RRS must be queried. In other words, game entities such as characters must be able to search the relationship network, uncover relevant relationships and relationship attributes, and use this information to formulate the correct behaviour, dialogue, and so on for the current situation. Other game entities may query the relationship network to formulate context appropriate content, such as missions or quests that reflect the current state of the game and the social network that exists within it.

To support this ability, a simple querying and matching facility has been defined. This provides the game ready and efficient access to relationship information whenever it is required. This is discussed further in the next section.

RRS ARCHITECTURE

The architecture required to facilitate the RRS consists of persistent descriptions of relations on disk, loading and internally representing the data in a useful manner, and lastly associating the data with existing game world objects or actors. This broad architecture is depicted below in Figure 2.

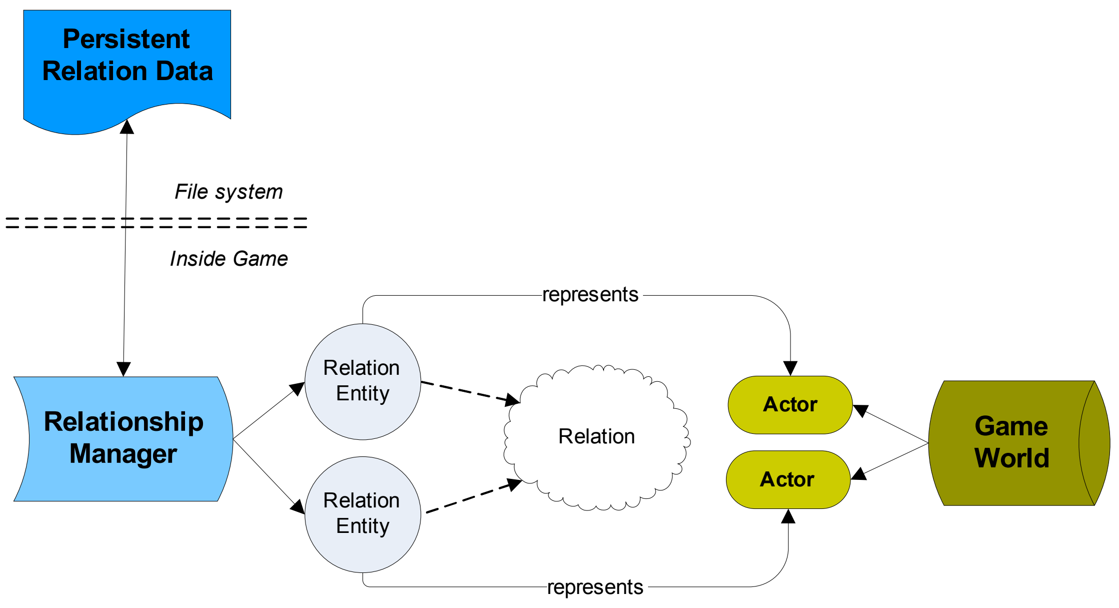


Figure 2: Architectural overview of RRS and its connection with the game world

The persistent relation data residing on disk contains all of the core relationship data in detail; this data can either be stored in a flat file, or some kind of database. The relationships themselves are maintained and controlled by a relationship manager. The relationship manager provides the ability to load relationship data from persistent storage, and synchronize relationship data throughout the game as necessary. The relationship manager also provides the facilities to create, update, delete, modify, and query individual relationships or relationship attributes.

At the core of each relationship are relation entities. These entities can be characters, objects, or groups from the game world. Consequently, each relation entity is linked back to the corresponding actor from the game world. Relation entities can be created dynamically to reflect new or previously unencountered actors in the game world, as the relationship manager deems appropriate. As the game is initialized and progresses, the relationship manager constructs the required relationships using these relation entities, as discussed below.

Modeling of Relations

Relations have been modeled to recreate various relation types in a manner similar to the real world, capturing a variety of relationship types and subtypes as discussed earlier. A relation is used to represent the association of one entity with another. A relation is described by both a perception and a description. The perception is used to represent all relationship attributes pertaining to how a particular entity views the relation in question. The perception is, in essence, the relation entity's opinion of their relationship with the other entity in question. For example, continuing the scenario depicted in Figure 1, the waitress initially might have perceived that her relationship with the bouncer at the bar was one of love, when in fact it may not have been. The description of the relationship includes only the hard, solid facts surrounding the relationship in

question. For example, the description of the waitress's relationship with the bouncer could indicate that they have known each other for 3 years and that they meet regularly on a daily basis at work.
Figure 3 illustrates the scenario in which one entity is aware of the existence of another entity but not necessarily vice-versa, also referred to as a semi-relation in this work. Following the example from Figure 1, if the waitress was to observe the player doing something heroic, she would establish a semi-relation of admiration with the player, even if the player was unaware of the existence of the waitress in this case.

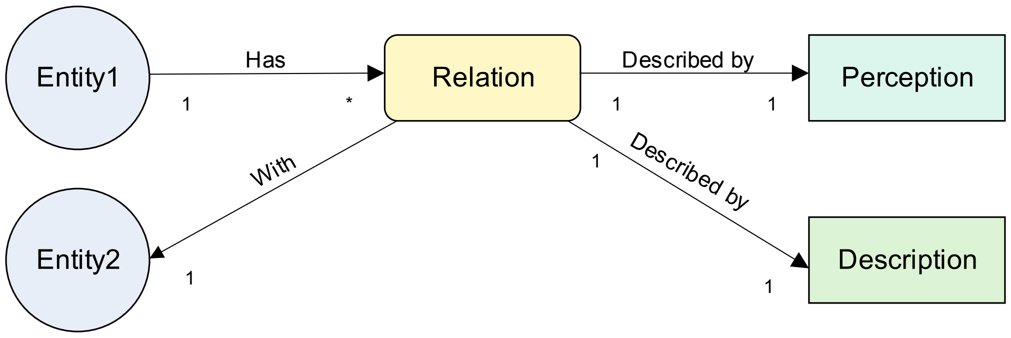


Figure 3: A one way relationship, also known as a semi-relation

Figure 4 illustrates the scenario in which the two entities are both aware of the existence of each other. This is called a full-relation in this work. This requires the use of two relation objects, two perceptions, but just a single description. There are two perceptions in this model because each entity is entitled to its own perception of the other entity in the relation. As you will notice there is a common description though because facts contained, like the duration of the relationship and historical events in the relationship, are unambiguous. (The perception of these facts might differ between the entities in question, but the description contains the facts independent of these perceptions.) Continuing the above example from Figure 1, the waitress might now have a perception that she dislikes the bouncer due to his association with the Fat Dragon Mercenaries, even though he has a different perception and still loves her. The factual description of the relationship would be the same, however.

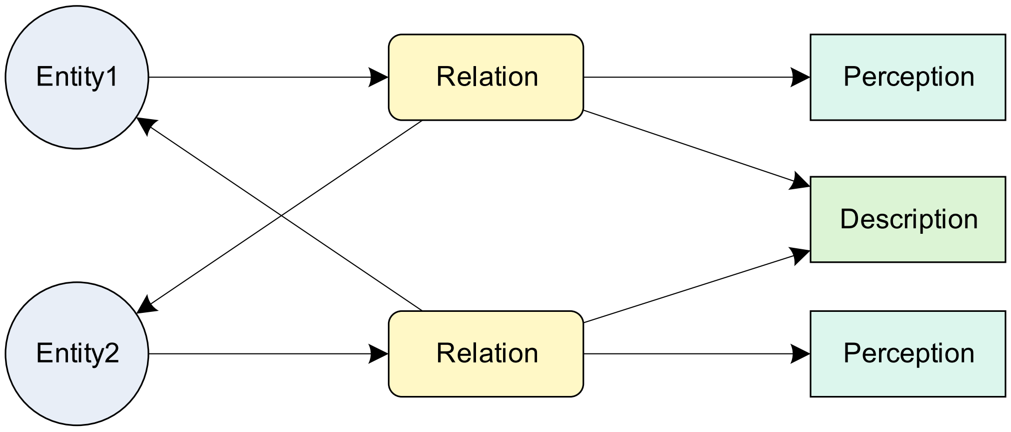


Figure 4: A full-relation with shared description and unique perceptions

An advantage to modeling the relation as two separate relation objects is that each entity's perception can be hidden from the point of view of the other side of the relation. Consider the case in which the artificial intelligence controller for Entity1 is reacting to an event involving Entity2. Entity 1 should only act based on its own perception and the common description attributes of the relation.

Extending the System

The architecture described above is quite flexible and allows complex relationships to be modeled quite effectively within a game. RRS can easily be used to represent semi and full-relations, relations between characters and objects, and relations between groups. This latter ability can be quite useful for modeling the relationships between collections of entities. Following the example from Figure 1, the guards group can have an on-going hostility with the Fat Dragon Mercenaries, providing a default relationship for all members of the group until specific relationships are added for individual members of those groups.

The architecture is also object-oriented, allowing classes of entities, relations, perceptions, descriptions, and relationship attributes to be created and specialized for the needs of different games. This allows RRS to be implemented for use in a particular game engine with a basic library of classes provided for modeling common types of relationships. At the same time, this also provides the ability to specialize and extend the system for use in particular games that make use of that engine. This was the case when we developed a prototype of RRS, as discussed in detail in the next section.

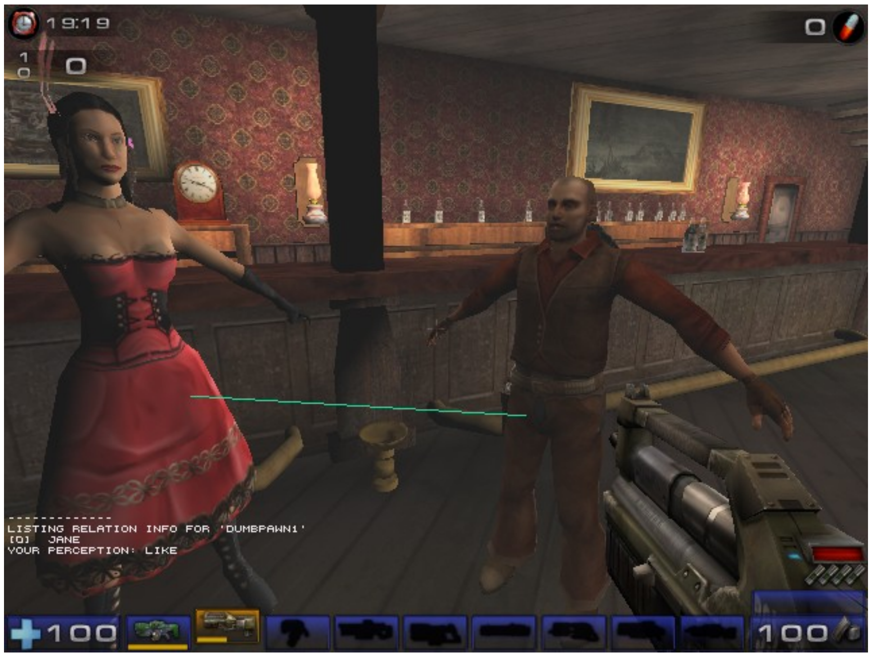
IMPLEMENTATION AND EXPERIENCE WITH RRS

A prototype of RRS has been developed for Epic's Unreal Engine [3] 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 RRS by either extending a new game type added to support RRS, or by embedding the appropriate RRS initialization hooks into an existing game type.

In addition, the RRS implementation in the Unreal Engine provides additional console commands to support manipulation of relationships manually from within the game. This allows game developers and designers to add relationship information during production from within the game itself, allowing easy debugging and creation of content.

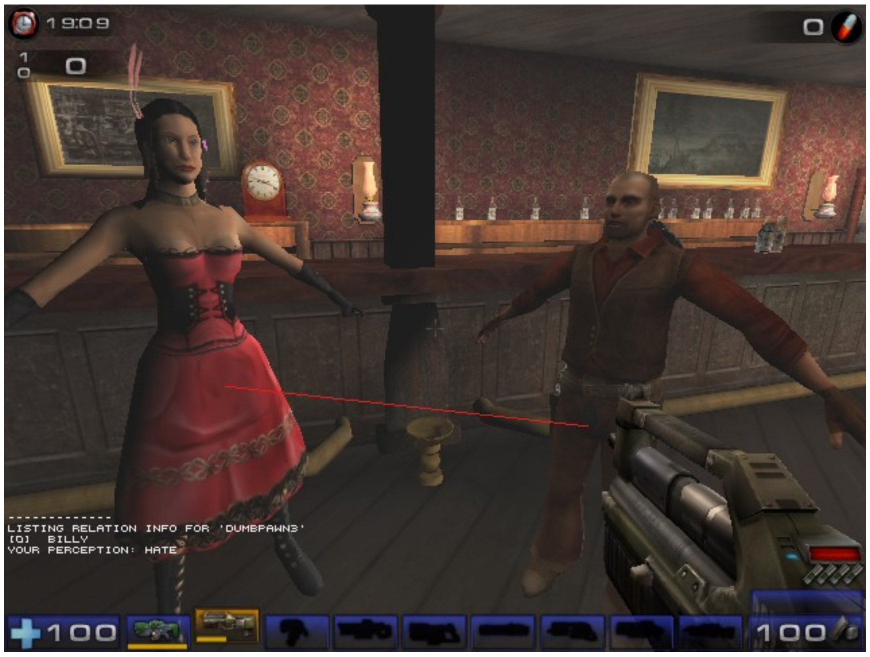
After development, initial validation of RRS took the form of individual test cases. More extensive validation took the form of modifying the existing LawDogs game modification to Unreal Tournament 2003/2004 [1] to support RRS. LawDogs was chosen primarily because its setting included a bar scene, which follows in line closely to the example used in our example relationship scenario discussed throughout this paper, and introduced originally in [2].

Figures 5 and 6 demonstrate the relationship facilities in RRS in action. To highlight the relationships present in the system, coloured lines were drawn to represent relationships, and text was output at the bottom of the game display. This information would not be displayed during an actual game, however, and would only be tracked and maintained internally. In Figure 5, for example, we can observe that the cowboy patron has an emotional relationship established with the waitress in the scene, indicating that he likes her. Figure 6, on the other hand, indicates that the waitress has a different perception of their relationship. In fact, she intensely dislikes the cowboy patron.



```
-------------
LISTING RELATION INFO FOR 'DUMBPAWN1'
[0]   JANE
YOUR PERCEPTION: LIKE
```

Figure 5: A demonstration of a "like" relationship facilitated by RRS.



```
-------------
LISTING RELATION INFO FOR 'DUMBPAWN3'
[0]   BILLY
YOUR PERCEPTION: HATE
```

Figure 6: A demonstration of a "hate" relationship facilitated by RRS.

RRS is currently being deployed for use in the Neomancer project [2,4], a joint development effort between the University of Western Ontario and Seneca College. Efforts in this project are currently being directed towards integrating RRS functionality, providing RRS with the contextual information required to build and maintain relationships, and using relationship information provided by RRS to drive character behaviours.

CONCLUDING REMARKS

By capturing game relationships and facilitating more appropriate character responses, our Realistic Reaction System can provide more immersive and compelling gameplay in modern video games. Experimentation with an Unreal-based implementation of RRS to date has proven successful, and demonstrates promise for future development efforts.

In the future, we plan to complete our current integration efforts and port 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. Finally, we also intend to extend our library of pre-defined relationships and relationship attributes to allow RRS to express a wider range of relationships by default.

REFERENCES

1. G. Castaneda, et al. LawDogs UT2003-UT2004 Modification. *Available from project home page online at http://www.planetunreal.com/lawdogs.* February 2005.
2. S. Danton. *Neomancer Game Design Document*. Unpublished manuscript. November 2004.
3. Epic Games. *Unreal Engine 2, Patch-level 3339*. November 2004.
4. 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.
5. G. Lawson. "Stop Relying on Cognitive Science In Game Design - Use Social Science". *In Gamasutra Letter to the Editor (available at http://www.gamasutra.com/php-bin/letter_display.php?letter_id=647)*. December 2003.