A Relationship Model for Believable Social Dynamics of Characters in Games

by

Brendan O'Connor, BBIT. (Hons.)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2015

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Brendan O'Connor

August 31, 2015

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Brendan O'Connor

August 31, 2015

Acknowledgments

First and foremost I would like to thank my supervisor Dr. Mads Haahr for his advice and support throughout the duration of this dissertation. Dr. Haahr offered valuable insights and direction which helped to improve the quality of work produced here.

I would also like to give thanks to Sarah Noonan and Tony Cullen for their advice, feedback, ideas and overall camaraderie during the course of the project.

Finally, I would like to thank my family and friends for their incredible support, encouragement and belief in me. I would not have found the courage to continue charging forward without all of you.

Brendan O'Connor

University of Dublin, Trinity College
August 2015

A Relationship Model for Believable Social Dynamics of Characters in Games

Brendan O'Connor
University of Dublin, Trinity College, 2015

Supervisor: Dr. Mads Haahr

This dissertation investigates believable social dynamics based on a relationship model between Non-Player Characters (NPCs) in games. The majority of interactions between characters in games is rigidly defined; with limited scope for character relationships that change as a result of interactions with each other or other factors. This limited ability hinders player immersion and believability in their interactions with the characters. This dissertation presents a model which can be easily applied to NPCs to provide them with relationship awareness. The model supports characters that store their own "assumed knowledge" of other characters relationships, in addition to their own direct relationships with other characters in their environment. This allows the characters awareness of not just their own relationships to other characters (e.g. to a sibling or child character), but also awareness of the relationships between other characters (e.g. that two other characters may be enemies). This knowledge can be treated

as an assumption; allowing characters to be provided with misleading information - the results of which are left up to extended implementations of the model. A messaging system is also implemented that allows the characters to communicate with one another directly as well as through their relationships (e.g. passing a message to all members of a group). The model has been implemented with a generic interface that allows other types of information to be tracked between NPCs as extensions to the basic relationships between characters. This provides for advanced implementations such as characters that "remember" previous relationship states with other NPCs. The model is tested and evaluated within a prototype environment within the Unity game engine. The implementation showcases a model with promising potential where characters act upon, learn and communicate about relationships they are aware of.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 3

1.3 Roadmap 3

Chapter 2 State of the Art **4**

2.1 Social Dynamics 4

2.1.1 Generating a concrete model 4

2.2 Social Psychology 6

2.2.1 Individual Characteristics and the Social Situation 6

2.2.2 Self-Concern 7

2.2.3 Other-Concern 7

2.2.4 Social Situations creating powerful Social Influence 8

2.2.5 Social Influence creates Social Norms 8

2.2.6 Cultures influence Social Norms 9

2.3 Social Networks 9

2.4 Group Dynamics 10

2.5 Social Relationships in Games 11

2.5.1 Prom Week (2012) . . . 12
2.5.2 The Sims (2000-2015) . . . 13
2.5.3 Façade (2005) . . . 14
2.6 Social Dynamics in Games . . . 15
2.6.1 Fable (2004) . . . 15
2.6.2 Shadow of Mordor (2014) . . . 17
2.6.3 Halo (2001) . . . 17
2.7 Current Decision Making Architectures . . . 18
2.7.1 Finite State Machines . . . 18
2.7.2 Behaviour Trees . . . 18
2.7.3 AI Planners . . . 19
2.7.4 Utility Systems . . . 19
2.7.5 Scripted Events . . . 20
2.8 Conclusion . . . 20

Chapter 3 Design **22**
3.1 Model Definition . . . 22
3.1.1 Representing Relationships . . . 23
3.1.2 Graph Interface . . . 28
3.2 Group Behaviour . . . 29
3.2.1 Applications of group behaviour . . . 30
3.3 Messaging System . . . 32
3.3.1 Learning support . . . 32
3.4 Illustrating the Model . . . 33
3.4.1 The Trader . . . 34
3.4.2 The Mob . . . 34
3.5 Model Architecture . . . 35
3.6 Conclusion . . . 36

Chapter 4 Implementation **37**
4.1 Platform Selection . . . 37
4.2 Graph Implementation . . . 38
4.2.1 Graph . . . 38

4.2.2 Relationships . . . 39
4.2.3 Entities (Nodes) . . . 40
4.2.4 Connections (Edges) . . . 42
4.3 Messaging System . . . 43
4.3.1 Broadcast messages . . . 44
4.4 Learning Systems . . . 44
4.4.1 Relationship Evaluation . . . 45
4.4.2 Adopting new Relationships . . . 45
4.5 Model Visualisation . . . 45
4.5.1 Sphere Demo . . . 46
4.5.2 Bob the Guard . . . 46

Chapter 5 Evaluation **48**
5.1 Believable Social Behaviour . . . 48
5.2 Complexity Analysis . . . 51
5.3 Performance . . . 52
5.3.1 Memory . . . 52
5.3.2 Processing . . . 54
5.4 Shortcomings & Challenges . . . 56

Chapter 6 Conclusion **58**
6.1 Contributions . . . 58
6.2 Future Work . . . 60
6.2.1 Improved Learning System . . . 60
6.2.2 Misleading Characters . . . 60
6.2.3 Optimizations . . . 60
6.2.4 Connection and Relationship Decay . . . 61
6.2.5 Integration . . . 61
6.2.6 WideEdge . . . 61
6.3 Final Thoughts . . . 62

Appendices **63**

Bibliography **64**

List of Tables

5.1 Complexity analysis of common relationship graph methods. E are the number of Edges or Connections in the graph, N are the number of Nodes, R are the number of Relationships. 51
5.2 Memory usage with different combinations of entities (nodes), connections (edges) and relationships. The number of relationships captured by the graph is significantly affected by the number of connections and entities e.g. 5 entities each storing 5 connections, with each connection storing 10 relationships would result in 250 (5*5*10) relationships. . . . 54
5.3 Computation performance of the Relationship Graph using different quantities of Entities, Connections and Relationship. 56

List of Figures

2.1 Aristotelian story tension value arc. 14

3.1 Representation of how Nodes, Edges and Relationships are captured in the RelationshipGraph model. Nodes represent Entities, Edges represent Connections from one character to another, and Relationships represent the bond between two characters. Relationships are treated as a distinct component of an Edge and can be assigned to different connections. The directed nature of the graph allows for nodes that have both unidirectional relationships (such as those from node A to node B and to node C) and for bidirectional relationships (such as those between node's B and C). 24

3.2 Representation of a simple RelationshipGraph. Each Node represents a character (or group). The edge's represent inter-relationships. An indirect edge is defined as an observed edge between two other nodes, represented here by the red dashed line that points to the edge from node C to node B. 27

3.3 Representation of how grouping can be captured in the model. Each Node can represents a character or group, in this diagram the Node "G" represents a group and other Nodes specify that they belong to the group "G" by specifying the "MEMBER" relationship to it. In this example, the Nodes "M1", "M2" and "M3" are "MEMBER"'s of the Group represented by Node "G". Node "G" identifies the Node "O" as an "ENEMY" of the Group, all "MEMBER"'s of the Group "G" can inherit this relationship due to their association with the Group i.e. as "O" is an enemy of the group "G", it is therefore the enemy of all members of that group. 31
3.4 Representation of how simple learning works at a high level, a connection is passed to the destination Entity as the payload of a message. The receiving Entity evaluates this Connection and discards or adopts it to its Connection list 33

4.1 All Graph implementations in the prototype are extensions of a base Graph type that implements the IDictionary¡TKey, TValue¿ interface, and a DeepGraph class that extends the Graph with support for multiple TValues per TKey - this is how multiple connections are stored per entity within the graph. 38
4.2 **IRelationship** Interface used in implementations of Relationship classes. The Equals method is used to identify a unique Relationship and is used when traversing the graph. 39
4.3 **INode** Interface used to implement Entities. The Equals method is used to identify a unique Relationship and is used when traversing the graph. The HandleMessage method is used to handle messages passed to the Entity. 40
4.4 **IEdge** Interface used to implement Connections. The "From" method defines the source of the Connection, "To" defines the destination Entity and the Relationship method defines a relationship between between the two Entities. When a comparison is made between two Connections, the graph relies on the implementation of the Equals methods of the **INode** and **IRelationship** interfaces 42

4.5 The **IMessage** Interface is used in the implementations of Message classes that are passed to Entities through the HandleMessage method of **INode** implementations. 43
4.6 Representation of how broadcast messages are implemented in the Graph. A message is sent from Entity "O" to the Group Entity "G". This in turn "Broadcasts" the message to each of its members i.e. those Entities that have a "MEMBER" relationship *to* the group Entity 44
4.7 Screenshot of Bob the Guard. The player can use the interface to communicate with the character. They can send text messages to Bob using the input on the top right, or a Connection message constructed from the options on the left and submitted with the Send Connection button. Questions about Bob's relationships to other characters can be submitted using the other available buttons and his responses are demonstrated in the screenshots above. 47

6.1 Screenshot of the Guard demo. 59

Chapter 1

Introduction

This dissertation investigates believable social dynamics based on a relationship model between Non Player Characters (NPC's) and the player's character. The focus is on NPC's that the player interacts with directly, allowing the nature of the relationships between characters to develop over time based on the actions of the parties to one another.

The hypothesis proposes that the creation of a relationship model or simulated social network would enhance the believability of the NPC's behaviour by providing motivations for their actions based on their connections to other NPC's or to the player. Testing this hypothesis would be performed by the creation of a demonstration environment where NPC's interact with one another and with the player allowing for the model to be observed and compared to existing models.

The model aims to be generic enough to encapsulate simple relationships between a few NPC's and more complex interactions in larger groups, and also to act as a reference for other systems in the game. It is hoped that this will create more believable and immersive gameplay experiences for the player.

1.1 Motivation

All too often, interactions between characters in games that share some form of relationship with one another, whether those relationships are familial (such as those between Alyx in *Half-Life 2* and her father, or the conversations between Joel and

Ellie in *The Last of Us*) or more strict relationships (such as those between a captain and his crew-mates, or of a soldier and their commanding officer) are often hard coded into the games AI or are scripted events. And the growth of these relationships over time are often dependent on the story of the game and not between actual interactions between the player and the NPC's - which can lead to conflicting behaviour; for example, the player attacking a friendly NPC but that NPC continuing to trust the player implicitly due to the requirements of the story.

In games, the goal of *Artificial Intelligence* (AI) is to build believably intelligent entities for the player to interact with. More recently there have been efforts at emergent player driven stories and narratives that evolve from the interactions with the AI system, such as in the Nemesis System of Shadow of Mordor [7]. Game AI has to work with limited resources as the processing time and memory must be shared with other processes that are also important to generating a believable and immersive experience for the player such as graphics, physics, audio and other items. However, the importance of the role of quality AI should not be underestimated as it must also be entertaining in their interactions with the player.

Artificial Intelligence attempts not just to understand but to build intelligent entities. Definitions of AI vary along two main dimensions: those concerned with replicating thought processes and reasoning and those that address behaviour [12]. According to Buckland, artificial intelligence can be split into these two broad camps of strong and weak AI [13]. Where strong AI concerns itself with the creation of systems that mimic the human thought process and the field of weak AI is concerned with the application of AI techniques to the solution of real-world problems [13].

The player often comes into direct contact with NPC's, thus the NPC's affect the immersion of the player in the virtual environment. Unrealistic behaviour of NPC's can break this immersion and pull the player out of the experience damaging their connection to the game. Believable behaviour such as the grunts that flee in the Halo franchise when their numerical or tactical advantage is lost are much more believable than even more recent games such as the Assassins Creed series where enemies all behave similarly and take turns to engage the player in combat instead of forcing their advantage.

1.2 Objectives

The aim of this dissertation is to design a relationship model to improve the believability of Non-Player Characters in games. This model will then be implemented and evaluated to determine if they offer an improvement over existing solutions. The dissertation will aim to meet the following objectives:

1. Offer a tangible improvement over current solutions
2. Support generation of believable actions based on relationships with other NPC's
3. Provide an efficient and generic implementation for use in a wide range of support for static and dynamic systems (e.g. generating a relationship model for procedurally generated bandits)
4. Efficient reference model for use in game by other systems
5. Flexible Model for use in different genres of games

1.3 Roadmap

Chapter 2 reviews the current state of the art of NPC inter-relationships in games and a review of sociological and psychological models of relationships and social interactions. Chapter 3 discusses the design of the Relationship model, describing the design choices made as well as illustrating how the model can be used to create believable characters. Chapter 4 describes the implementation of the model by detailing each of the main components of the model and how they work together, and some advanced behaviours that were implemented based on these components. Chapter 5 evaluates the performance of the model as well as offering a comparative analysis to an existing system and demonstrating how the other system can be implemented more effectively using the model. Finally, Chapter 6 offers a summary of the contributions made by the project, final remarks and potential future work that can improve on the current model.

Chapter 2

State of the Art

The purpose of this chapter is to provide background on the relevant areas and prior work done on the modeling of social dynamics and human inter-relationships in sociology and in games. First, a discussion of some basic sociology and psychological research on human relationships and interactions including an example of a social dynamics model. This is followed by a discussion of Artificial Intelligence (AI) implementations of social dynamics and character relationships in games. Lastly, we review techniques used in the video games industry to try and simulate believable social dynamics.

2.1 Social Dynamics

Durlauf et al. define *Social Dynamics* as the explicit study of the interactions linking individual behaviour and group outcomes [3]. They state that the study begins with the assumption that the actions and choices of individuals affect others around them. As these actions and choices happen sequentially, a feedback loop arises where the past actions of some individuals affect the future behaviour of others. The resulting dynamical system from these interactions is the focus of the study of social dynamics.

2.1.1 Generating a concrete model

Durlauf and Young make note of some methodological "questions" that must be addressed to generate a concrete model [2]:

- **Individuality** The Individuality of the agents or subjects must be maintained at all times. Behavioural rules must apply to individuals rather than representatives (such as aggregates)
- **Randomness** Some allowance for random perturbations arising from variations in the environment, errors in transmission of information, and the diversity of individual responses.
- **Properties** Identification of the aggregate properties of the model that should be captured (or studied). Aggregate properties are important due to the maintenance of individuality and heterogeneity of the agents/characters - which may result in a large and unwieldy state space.
- **Responsiveness** There must be an understanding of how individuals respond to their beliefs concerning the characteristics and behaviours of others.
- **Absorption** How beliefs are formed must be specified. This is a function of the ability of individuals to learn, reason and process information.

Social Dynamics studies the aggregated properties of the resulting stochastic dynamical system of these diverse elements. However, this resulting stochasticity is an element that would lend itself well to games and other interactive models - to provide an element of unpredictability to the virtual environment.

Durlauf, et al., provide a demonstration of how these insights could be embedded in a formal model [3]; consider that we have I agents situated in a social or geographic space that determines lines of communication and degrees of social influence. If we suppose that each agent is situated at the vertex of a directed graph, that each edge (i, j) is weighted by its importance, d_{ij} which is taken as non-negative. If we also suppose that each agent has a finite number of actions X that are observable by others and a state of the system that is a collection of actions by each agent $\omega_t = (\omega_{1,t}, ..., \omega_{I,t})$, where $\omega_{i,t}$ is agent i's action at t. Each agent i is affected by the actions of others, so it is useful to define $\omega_{-i,t} = \omega_{1,t}, ..., \omega_{i-1,t}, \omega_{i+1,t}, ..., \omega_{I,t})$. Over time agents reconsider what they are doing in the light of current circumstances and have the opportunity to alter their actions. Agent i's choice of actions is governed by i's personal preferences concerning actions, independent of what others are doing, plus the actions of others, weighted by their importance to i.

Formally, this may be represented as follows; let θ_i denote a vector of characteristics of i that influence that agent's payoff from each possible action. In choosing an action $\omega \in X$, agent i receives a private payoff $v(\omega_{i,t}, \theta_i)$ plus a social payoff $\sum_{j \neq i} d_{i,j} s(\omega_{i,t}, \omega_{j,t}, \theta_i)$. Therefore each actor will make a choice in order to maximise:

$$U_i(\omega_{i,t}, \omega_{-i,t}, \theta_i) = v(\omega_{i,t}, \theta_i) + \sum_{j \neq i} d_{i,j} s(\omega_{i,t}, \omega_{j,t}, \theta_i)$$

assuming each individuals choice is perfectly predicted from this maximisation problem [3].

2.2 Social Psychology

According to Charles Stangor, social psychology is the study of the dynamic relationship between individuals and the people around them. Each individual has their own characteristics, including personality traits, desires, motivations and emotions - all of which have an important impact on our social behaviour [1].

2.2.1 Individual Characteristics and the Social Situation

Charles Stangor states that there is an acceptance that behaviour is influenced by context or the *social situation* [1]. The social situation refers to the "others" that individuals interact with every day including friends and family, religious groups, and other individuals observed on television or read about or interacted with on the web. There are also other individuals that are thought about, remembered or even imagined.

Social psychology studies *social influence* - the process through which others change an individual's thoughts, feelings and behavious and vice-versa.

Kurt Lewin formalized the joint influence of an individual's characteristics and the situational variables with the following equation [1]:

$$behaviour = f(indivudialcharacteristics, socialsituation)$$

Indicating that the behaviour of individuals at any given point is a function of the characteristics of the individual and the influence of the social situation they find themselves in.

According to Vansteenkiste and Ryan, "In the current research, grounded in self determination theory, ... evidence that both peoples healthy tendencies toward growth and integrity and their vulnerabilities to ill-being and psychopathology can to a significant degree be explained by a single underlying principle. Stated simply, basic psychological *need* satisfaction and frustration can substantially account for both the 'dark' and 'bright' side of peoples functioning" [33]. The frustration of the needs by the social situation can lead to negative individual behaviour, while a satisfaction of the needs leads to a positive or neutral individual behaviour.

Stangor says that evolutionary adaption has driven two fundamental motivations that guide individuals [1], these are *self-concern* and *other-concern*:

2.2.2 Self-Concern

The most basic tendency of all living organisms is the desire to protect and enhance one's own life and the lives of those closest to us. This is what drives humans to find shelter or food as doing so is necessary is fundamental for one's own survival. This desire to maintain and enhance the self extends to relatives - those genetically related to us. Humans exhibit *kin selection* - strategies favoring the reproductive success of one's relatives. In addition to kin, humans aim to protect and improve the well-group of our *in-group* - those whom we view as similar and important to us and with whom social connections are shared, even if they aren't genetically related.

2.2.3 Other-Concern

Despite a primary concern with the survival of our selves, our kin and our ingroup, there is also a desire to connect and be accepted by others - *other-concern*. This is reflected by living in communities, working and worshiping in groups, playing together in teams. These interactions facilitate a fundamental goal of finding a romantic partner. These connections facilitate other opportunities that may not be performed by the individual alone e.g. a carpenter to build a house, or a teacher to learn new skills/knowledge. Affiliation is also enjoyable as is being a part of a social group.

The other-concern motivation demonstrates that part of being human involves caring for, assisting, and cooperating with other people. The primary motivations of survival may be selfish, but the survival of our own genes may be improved by the

assistance of those not related to us. This results in moral behaviour between humans. Individuals understand it is wrong to harm each other without a good reason to do so as it works against our own self interests. This results in negative behaviours towards others being viewed as unusual, unexpected and socially disapproved. Negative behaviours may still arise based on the social situation, however the fundamental motivation of other concern results in negative behaviour being an exception rather than the norm.

The goals of self-concern and other-concern may go hand-in-hand or may conflict based on the individuals characteristics. For example, a connection with another individual in a loving relationship results in an individual feeling good about themselves as well sharing a concern for the other. However, when observing another person being attacked the desire to help the other and the desire for self-preservation results in a conflict between self- and other-concern, and the individual must make a decision on whether or not to intervene.

2.2.4 Social Situations creating powerful Social Influence

An important principle of social psychology is that although individuals' characteristics matter, the social situation can often be a stronger determinant of behavior than an individuals' personality. In addition to the people with whom we are currently interacting, we are influenced by people who are not physically present but who are nevertheless part of our thoughts and feelings [1].

2.2.5 Social Influence creates Social Norms

Social influence can occur passively, without any obvious intent of one individual to influence the other, such as when we learn about and adopt the beliefs and behaviors of the people around us, often without really being aware that we are doing so. An example of this occurs when a child adopts the beliefs and values of their parents. In other instances, it is not as subtle and individuals may actively attempt to change the beliefs or behaviours of others, for example with advertising by popular individuals. Social Norms are ways of thinking, feeling or behaving that are shared by group members and perceived by them as appropriate - norms include customs, traditions, standards and rules as well as general values of the group (e.g. timeliness, tidiness, etc.) [1].

2.2.6 Cultures influence Social Norms

A culture represents a group of people, normally living within a given geographical region, who share a common set of social norms, including religious and family values and moral beliefs. The culture in which we live affects our thoughts, feelings, and behavior through teaching, imitation, and other forms of social transmission. Cultural differences for example, with individualistic (self-concern) versus collectivistic orientations (other-concern) guide our everyday behavior [1].

2.3 Social Networks

A social network is a structure of relationships that ties actors to one another. According to Wasserman and Faust, "the unit of analysis ... is not the individual, but an entity consisting of a collection of individuals and the linkages among them" [26]. The network methods focus on *dyads* (ties between two actors), *triads* (between three actors) or larger systems (groups, organisations, etc.). There is a clear link between networks and the social structures and social interactions that are sociology's central concerns [24].

A brief summation of the principles that underline the social network perspective, according to Pescosolido [25], include:

1. Network interactions influence beliefs and attitudes as well as behaviour, action and outcomes.

2. Individuals are neither puppets of the social structure nor purely rational, calculating individuals.

3. Abstract influences can be understood by looking at the set of social interactions that occur within them.

4. Social networks have 3 distinct characteristics: structure (the dimensions of the network ties e.g. size, density), content (what flows across network ties) and function (outcomes of the interactions).

5. Network influence requires the consideration of interactions among these three aspects.

6. Networks may be in sync or in conflict with one another.

7. Social interactions can be positive or negative, helpful or harmful.

8. More is not necessarily better with regard to social ties. Stronger ties are not necessarily more optimal than weak ties due to the opportunities that may be afforded by the weaker ties.

9. Networks across all levels are dynamic, not static, structures and processes.

10. A network perspective allows for, and even calls for, multi-method approaches.

11. Socio-demographic characteristics are potential factors shaping the boundaries of social networks but provide, at best, poor measures of social interaction.

12. Individuals form ties under contextual constraints and interact given social psychological and neurological capacities.

Social Network Analysis is the application of methods and models for the analysis of social network data. In social network analysis the observed attributes of social actors are understood in terms of patterns or structures of ties among the units[26]. Scott states that "Social network analysis is appropriate for for relational data and that techniques developed for the analysis of other types of data are likely to be of only limited value for research that generates data of this kind" [27].

2.4 Group Dynamics

Forsyth [18] states that "Group dynamics are the influential interpersonal process that take place in groups" and "All but an occasional recluse or exile belong to groups, and those who insist on living their lives apart from others, refusing to join any groups, are considered curiosities, eccentrics, or even mentally unsettled" [19] [17]. Kurt Lewin used the term *group dynamics* to stress the powerful, fluid and active impact of these complex social processes on group members [18]. A group is formed by two or more individuals who are connected to one another by social relationships.

Groups can be planned or emergent. Emergent group norms are sustained by a common set of group level processes. Forsyth and Elliot state that, "Informational

influence occurs when the group provides members with information that they can use to make decisions and form opinions" [19].

Groups have differing levels of influence psychologically; with primary groups (close-knit groups such as families, close friendships, neighbourhoods, etc.) having a stronger influence than secondary groups. However secondary groups still have an influence over an individual's place in society. Both of these types of groups provide members with their attitudes, values and identities and teach their members the skills they need to contribute to the group, provide them with the opportunity to discover and internalize their behaviour in response to social norms and others' requirements, thus groups *socialize* individual members [19].

Some of the core assumptions about groups include [18]:

- Groups Are Real:
- Group Processes Are Real
- Groups are more than the Sum of their Parts
- Groups are living systems
- Groups are influential
- Groups shape society

Individuals within a group interact with one another on the task at hand. Groups create interdependence among the members of the group and the interaction within a group is determined by the group structure which defines the roles, norms and interpersonal relations within the group. Group cohesion determines the unity of the group [18].

2.5 Social Relationships in Games

This section reviews the state of the art implementations of social relationships in games. These are typically games with social relationships as part of their core gameplay and are much rarer than other types of action oriented games due to limitations on speed and memory requirements, but see deeper and more experimental projects into modeling social dynamics.

2.5.1 Prom Week (2012)

Prom Week is a game that revolves around the social lives of eighteen characters. According to the paper by McCoy et al. [9], the player is given a set of goals to complete during the week before the prom e.g. getting a date to the prom. These goals are attained by discovering solutions through interactions with the characters and social states. The player works towards these goals by initiating *social exchanges* with each character. These social exchanges are multi-character social interactions that modify the social state connected to the participants. Which social exchanges are available and how each changes the social state are determined by the games AI system *Comme il Faut (CiF)* [9]. The player chooses from an ordered list generated by CiF based on the social state and the character models. The system also determines whether or not a character will respond positively or negatively to a social exchange. This response is emergent from the underlying social simulation rather than from pre-determined scripted content.

Prom week introduces the concept of *Social physics* [9], what the authors describe as "...a simulation of social dynamics specifically crafted for a targeted experience ... an enjoyable simplification tuned for gameplay..." just as platform games don't reproduce realistic physics of the real world. *Prom Week's* AI system provides a knowledge representation and processes that models social interactions to make an ambitious goal of simulation of a large space of contexts and social interactions an attainable one.

These social physics are based on a set of over 5,000 socio-cultural considerations based on ethnographic analysis of pertinent media sources [9]. The analysis of social norms and behaviours in a particular cultural setting, or social biases, prejudices or other patterns of social interaction can also be incorporated into the model. These considerations are used as rules to influence character desires, each adding either a positive or negative numerical weight to the desirability of each potential social exchange.

CiF operates by iterating through a set of processes over a given social state to determine what characters are interested in doing and how they may respond to the other characters taking these social actions with them - this is desire formation. This iterative process results in all the characters having a weighted determination to pursue certain social exchanges with every other character. The player then selects a social exchange for one character to perform with another and this results in an initiator intent

(a character's desired social change) and three roles: an initiator, a responder and a potential third party. If a third party is involved, CiF selects the character with whom the most influence rules pertaining to a third party were true. CiF then determines how a responder reacts based on the social context by calculating a sum for true rules that pertain to responding to the social exchange - if this sum is zero or greater the game responder accepts the intent otherwise it is rejected. The system uses *effects* as a means of narrating the outcome of a social exchange based on the social state of the participants and whether the exchange was accepted or rejected. Each of these effects is associated with a performance realization instantiation. Each instantiation is a set template based dialogue acts and animations. After these instantiations are realized the social state change associated with the chosen effect is applied, including placing an entry into a social facts database to track the exchange for future reference in other social exchanges. Lastly trigger rules are executed over the new social state to account for social changes arising from multiple social exchanges and other elements of the social state [9].

2.5.2 The Sims (2000-2015)

The Sims is a sandbox god-game that simulates the life of a human-like family. The game is recognized as having one of the most influential AIs in the game industry. The AI in the game handles lower level actions such as path-finding but also default behaviour for the NPC's when they are not in interaction with the player [4]. As the characters could survive and behave independently of the player actions, this led to, according to Yoann Bourse, "automatic generation of game narration" [4] where the narrative was an emergent result of the social interactions of the game characters.

Yoann Bourse also states that the social interactions in *The Sims* are based on a score between each pair of Sim characters [4]. The relationship between each pair of characters depends on this score e.g., enemies, friends, etc. which result in different possible actions. A precise rule system determines the outcomes of these social interactions, resulting in negative or positive results based on the mood, personality, the circumstances or sometimes a plain variable.

This is similar to the model described by Kurt Lewin that formalized the joint influence of an individuals' characteristics and the situational variables to determine

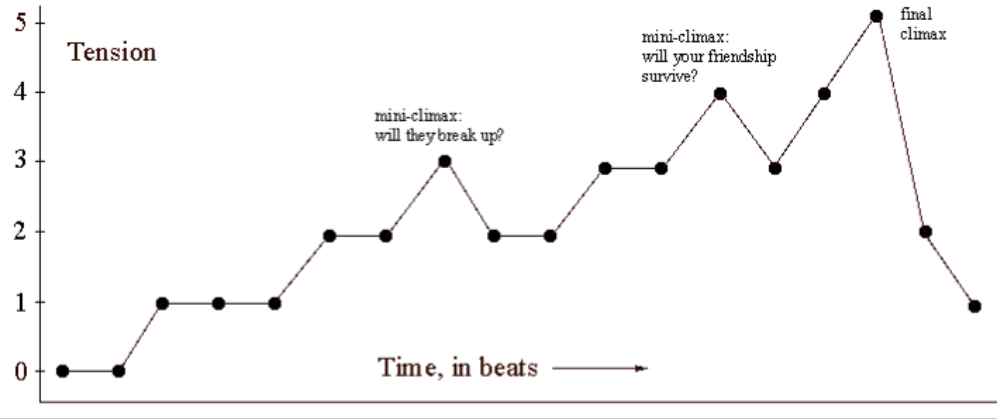

Figure 2.1: Aristotelian story tension value arc.

the character's behaviour.

2.5.3 Façade (2005)

Façade is a short game that "attempted to create a real-time 3D animated experience akin to being on stage with two live actors who are motivated to make a dramatic situation happen", according to authors Michael Mateas and Andrew Stern [22]. The game approached this by developing an architecture that merged drama management, believable agents and natural language processing. Drama management was implemented by providing architectural support for authoring drama beats, which combines aspects of character story.

Drama management in *Façade* is handled by breaking down a narrative into beats and using a beat sequencer to select the next beat in a story based the previous interaction history. A Beat Sequencing Language was developed for *Façade* where the author/developer may annotate each beat with selection knowledge, can define the actions that are performed at various stages in the beat selection process and can define the beats that are accessible by all the tests and actions within a beat [22]. Given a collection of beats, the beat sequencer selects beats for sequencing and a sequencing decision is initiated when the current beat successfully stops. These beats are then scored according to the Aristotelian story tension value arc [22] (*see Figure 1.*).

Façade also utilises *A Behaviour Language* (ABL) - a reactive planning language that supports sequential and parallel behaviours as well as joint behaviours to allow

the NPC's to react to player behaviour as well as to perform actions - sometimes simultaneously [21]. This allows for believable agent behaviour whereby they multitask (e.g. watch the player perform an action such as looking at a painting while conversing with another agent) and react realistically (e.g. stopping mid-sentence if the player performs an action that draws their attention).

However, while *Façade* provides an interesting approach to modeling interactions between NPC's, the approach is limited for a number of reasons. First, the relationship moves through scripted beats and a set overall narrative rather than being emergent behaviour from the characteristics of the agents. Secondly, the approach is time-consuming to develop and set up. For even the short scenario that plays out in the game which can last for about twenty minutes, hundreds of actions and behaviours must be specified and linked between story beats - something that would be better as a result of the interactions rather than painstakingly implemented manually.

2.6 Social Dynamics in Games

This section investigates implementations of social dynamics in games that results in believable behaviour by agents. These are games that incorporate social behaviour in the behaviour of enemies and other NPC's over and above the typical situational awareness or quest giving.

2.6.1 Fable (2004)

Fable is a series of games developed by Lionhead Studios. They are renowned for their implementation of opinionated AI systems where NPC's react to player actions and appearance and develop opinions of them. The games are set in a fantasy world where the player must set out on a quest to achieve their goals either through good deeds or evil deeds - with the player's character and the NPC's reflective those choices.

Fable allows players to craft their own avatars, however it focuses on the lifetime change of the characters' social reputation rather than just the initial choice of the character. To this end the game utilises an opinion system, by which individual NPC's form their own personal opinions of the players character based on observed behaviour. This opinion system was restricted to a subset of the overall gameplay separating it

from conventional RPG quest scripting. This is a popular approach also utilised in games such as *Elder Scrolls V: Skyrim* where random bandits are generated to combat the player as well as having persistent NPC's that live in towns. *Fable*'s opinion system revolves around the player, this means that the NPC's do not have opinions of each other - only of the player.

Fable's opinion system works on a basis of opinion states that describe an instanteneous evaluation of the player's social persona by the NPC [20]. It does not carry any historical context - the NPC has no knowledge of what prior actions led up to the current observed behaviour nor any record of previous expressions of opinion toward the players character. *Fable* uses a multidimensional opinion system that uses several values to describe an opinion state including morality, renown, scariness (of appearance), agreeableness, and attractiveness. A hybrid model of opinions in society is utilised to both present a single overall opinion status to the player as well to have a model that implements the propagation of knowledge in the game world from character to character. The opinion of the player is modeled as a function of the NPC's *relative opinion* of the player and the players own data that tracks the players own activity that may happen away from observable NPC's and these two sets are combined to form an *absolute individual opinion* of the player [20]:

$$individualopinion = f(playeractivity, relativeopinion)$$

The game also includes a village opinion that is used to approximate the propagation of information between villages as opposed to between individuals. This means that the individuals communicate with the village opinion in certain circumstances and this propagates to others and this village opinion is then included as part of the absolute opinion the NPC forms of the player. This allows the NPC's to form an opinion of the player if the have not yet formed one - an approximation of group dynamics, social norms and inherited bias or prejudice [20]:

$$opinion = CombinedOpinion(individualopinion, villageopinion)$$

2.6.2 Shadow of Mordor (2014)

Shadow of Mordor is a game developed by Monolith studios (of F.E.A.R. fame). It is an open-world action game set in the world of Middle Earth from the Lord of the Rings franchise. The game implements an AI system called the Nemesis System that has received plaudits in the games industry. This system was a result of the development team "..targeting our efforts on having the NPC's react and respond to the player, the environment and each other." according Michael de Plater of Monolith studios [7].

de Plater and his team used psychological theories as a reference for identifying and satisfying player needs through *Shadow of Mordor*. *Shadow of Mordor* sought to satisfy the three elemental needs identified in one of the applied theories; *the theory of self-determination* that states that humans have three fundamental needs [7]:

1. **Competence** - a need to feel effective in the environment
2. **Autonomy** - a need to control the course of their lives
3. **Relatedness** - a need to have relationships with others.

de Plater also references Player Experience of Need Satisfaction and GNS Theory as instrumental in the development of the Nemesis System [7]. de Plater also referenced sports as an established system that generates stories from a system based on pre-defined rules, with stories emerging from the interaction of the teams and players and other parties. In this systems, the sport offers the framework for the story and the parties fill in the details - which was an approach attempted in *Shadow of Mordor*. The game utilised a memory system to track the player's actions and indicate them to the player in the narrative [7].

2.6.3 Halo (2001)

The *Halo* games are a series of first person shooter games developed by Bungie and currently by 343 Industries for Microsoft. The games pit the player in the role of a super soldier that must battle alien forces. Alien forces in the games are not one single class of agent but instead are a "covenant" of alien races that have their own inter-relationships based on hierarchy within their society. This affects how different enemies behave when in the presence of other agents, for example, grunts are the weakest enemies and will

usually attack the player with a numerical advantage or when accompanied by a senior elite agent. This behaviour changes when they lose their numerical advantage or their leader is defeated and they instead try to run away from the player.

The *Halo* games implemented this using a hierarchical finite state machine (HFSM) or more specifically a behavior directed acyclic graph (DAG) [5], since a single behaviour or behaviour sub-tree can occupy several locations in the graph. They make a distinction between leaf nodes and non-leaf nodes (branches) in that the role of leaf nodes is to perform an action while the role of non-leaf nodes is to make decisions - with decisions being made based on custom code or on the competition of children nodes with a parent making a final decision based on the child behaviours desire to run or relevancy. They also utilise behaviour impulses to circumvent scenarios where certain behaviours have priority over other behaviours. The developers of the AI also took significant effort to ensure the system was flexible enough for workarounds and for level designers to work with in order to get the desired behaviour [5].

2.7 Current Decision Making Architectures

2.7.1 Finite State Machines

Finite State Machines (FSM) have been common in game development due to their flexibility, simplicity to implement and debug, minor computational cost and their intuitiveness [14]. Traditional FSMs do not scale due to the logic not being reusable as-is; a state cannot be treated as a modular block and referenced from a different context. A new state must be created with different transitions specifically for that new context [16].

This led to the development of *Hierarchical Finite State Machines* that allow for some reuse of logic. This was done by grouping together states into *super-states* to share *generalized transitions* [28].

2.7.2 Behaviour Trees

Behaviour trees improve on FSMs primarily by scaling better due to the removal of transitions to external states for self-contained states - as these states no longer have

transitions they can no longer be referred to as "states" they're just behaviours [29]. When branches are ordered by their desirability, they allow the AI to make use of fallback tactics should a particular behaviour fail [10]. Most recently *This War of Mine*, a smaller title released by Ubisoft, utilises behaviour trees for all its AI [8]. The *Halo 2* AI also implements a hierarchical finite state machine/behaviour tree hybrid [5].

2.7.3 AI Planners

Automated planning is an approach that has recently achieved success with applications as diverse as the Mars rovers, planned sheet-metal bending operations and in games as well [31]. Planning is a formalized process of searching for a sequence of actions to satisfy a goal - this is a process known as plan formulation. Hierarchical task network (HTN) planners do not have an objective defined as a set of goal states but instead as a collection of tasks to perform. HTN planners work by reducing the problem into sub-tasks: they decompose tasks into sub-tasks, and sub-tasks into further sub-sub-tasks and so on until primitive tasks that can be performed by planning operators. The planner utilises a set of methods to determine how to decompose non-primitive tasks into sub-tasks [34].

A Simple Hierarchical Ordered Planner (SHOP) are based on ordered task decomposition a type of HTN planning - they plan for tasks in the same order that they will be executed [30]. This reduces the complexity of reasoning by removing a great deal of uncertainty about the world, and allows the use of expressive domain representations. The *Killzone* video games make use of HTN planners inspired by SHOP [30]. STRIPS is a planning algorithm that searches through the world state by applying operators, and is typically done backwards from the goal state to the world state for performance reasons. *F.E.A.R.* is well known for its use of STRIPS [30].

2.7.4 Utility Systems

A utility system is a voting/scoring system and are often applied to game sub-systems for object or position selection based on a calculation. *The Sims* franchise of games is well known for its use of utility systems, even though this has had less focus in more recent releases [30]. Objects in the game world of *The Sims* correspond to threads and

contain the scripts required to follow for the different actions written in the EDITH custom script language. The object also advertises itself to the characters in the games by broadcasting what it can offer [4].

2.7.5 Scripted Events

Scripted events in games were popularized by the *Half-Life* series and have seen a particular resurgence in recent blockbuster games such as the *Call of Duty*, *Battlefield* and *Bioshock* franchises which rely heavily on scripted events to engage the player with spectacle - even to the extent of directing the player or insisting they follow an NPC along a set path. Scripted events in games handled the dissonant nature of cutscenes in older game titles, as instead of showing the player a pre-captured directed scene the player maintained "control" during the exposition of the story i.e. it happened as you played [11]. These events however happen at the same time, in the same way, with the same characters every time the player encounters. This means that these moments lose their impact after the first time they are experienced as the each time the actions are repeated the player is thrown out of their immersion. Jaimie Kuroiwa suggests implementing semi-randomised scripted events to engage the player [11]. Some games do this on different difficulties (e.g., in *Halo* the soldier companions state different dialogue based on different settings). However, these still remain scripted events with no actual AI driving the interactions except for basic path finding around a scene in some cases.

2.8 Conclusion

This chapter reviewed some basic sociological and psychological approaches and models for the analysis of social dynamics, social psychology, social networks and group dynamics. The goal was to have an overall concept of a few broad ideas with regards to these concepts. The fields of sociology and psychology cover a vast and diverse array of fields of study and limiting the scope of the research done to just high level broad concepts was important to have a limited model for implementation in an AI system.

This was followed by a review of approaches made by different games in their goals of generating believable behaviour from their NPC's. This was split into a review social

relationships that have been directly modelled as part of the gameplay such as *Prom Week* and other games that use social dynamics to add variety to existing gameplay such as *Fable* and *Shadow of Mordor*.

Finally there was a review of current decision making architectures in the games industry. This section focused on the different ways games present an NPC in the world making a decision, either through complex structures such as behaviour trees and AI planners or through scripted events.

Chapter 3

Design

This chapter presents the design of the *RelationshipGraph*, a model that aims to support the creation of believable characters in games based on relationships. This *RelationshipGraph* will be used to drive social interactions among game characters. First, a high level definition of the model is offered, detailing the representation of each of the main components of the model. Next is a description of how group behaviours, an important aspect of social dynamics, is supported by the model by building upon previous components. Following this is a description of the Messaging System and how it can be used with the core components of the graph to implement a simple learning behaviour. An illustration of how the *RelationshipGraph* can be used to create interesting characters is then provided. Finally the model Architecture is described.

3.1 Model Definition

Many interactions occur as a result of the connections between entities. These connections define relationships that can be utilised in decision making or other systems. As described in sections 2.1 and 2.2, relationships between individuals and groups are complex and based on a variety of influences, situations and the characteristics of each individual. Additionally, relationships do not exist in a vacuum and there is a need to grant entities awareness of the inter-relationships among other entities; thus allowing for more complex behaviour to emerge naturally based on what relationships exist within the larger social network.

Explicitly defining a specific implementation of relationships would limit the application and extensibility of the model. With this consideration in mind, the model does not offer a definition of what attributes define a relationship, but instead abstracts the concept of a relationship and leaves its specific implementation to the domain of use. This allows the model to focus on the storage, update and retrieval of these relationships between characters. For example, one system could base relationships on an opinion system similar to that found in the Fable [20] games or on the strength of the relationships between characters, the approach taken in this project.

The storage of the characters and their inter-relationships is best suited to a directed graph, with nodes and edges in the graphs representing characters and the connections between those characters respectively. The actual relationships between characters are stored in the model as an attribute of the connections. This allows for the creation of complex graphs by implementing more advanced implementations in either the nodes, edges or relationships of the graph. For example, graphs can be created that track prior relationship values by storing them in an edge that manages a history of relationship values between two nodes over time. This decoupling significantly improves the potential of the model to handle a large array of applications and requirements..

3.1.1 Representing Relationships

Every node within the graph represents a unique Entity such as an individual character or a group of characters. This concept can be extended to facilitate other types of entities such as locations or items that characters have relationships with. For example, a religious character may have a relationship to their place of worship or sacrifice; a warrior character may be a "Fan" of a particular weapon or amour.

An important point here is that we use the term "Entity" instead of "Character" when referring to the nodes of the RelationshipGraph model. This is because an Entity may be either a group or an individual character, so the term "Entity" acts as a catch-all for both types.

The graph allows for the support of these different types of Entities by keeping the nodes generic. The only requirement when creating an implementation of a node is a method that offers the graph a way to look up a unique Entity.

The edges between nodes represent Connections and they capture a link from one

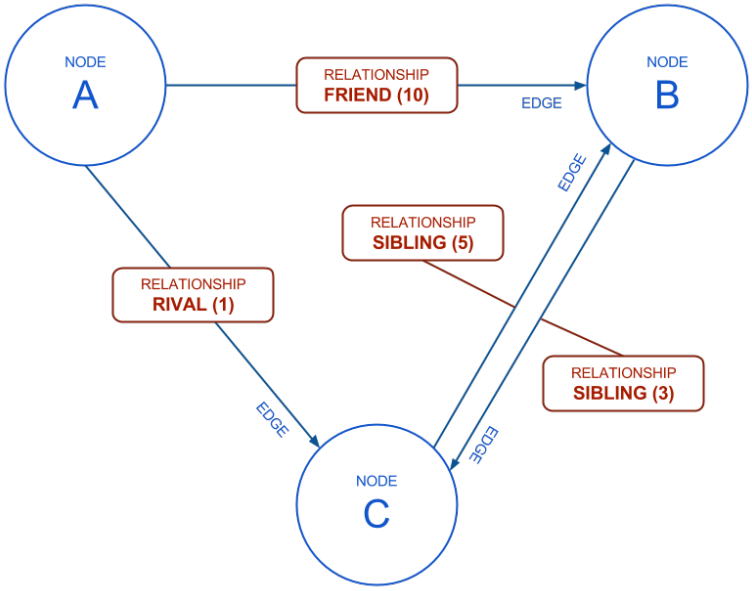

Figure 3.1: Representation of how Nodes, Edges and Relationships are captured in the RelationshipGraph model. Nodes represent Entities, Edges represent Connections from one character to another, and Relationships represent the bond between two characters. Relationships are treated as a distinct component of an Edge and can be assigned to different connections. The directed nature of the graph allows for nodes that have both unidirectional relationships (such as those from node A to node B and to node C) and for bidirectional relationships (such as those between node's B and C).

Entity to another. The RelationshipGraph only supports directed edges; each Connection must specify source and destination Entities and also a Relationship attribute.

An important distinction in the model is that *a relationship is not an edge*. A relationship value is stored instead as an *attribute* of an edge, indicating what the current relationship is from the source Entity to the destination Entity. This means that updating the relationship status is simply a matter of finding the relevant Connection and updating it's relationship attribute. The contents of the Relationship has no bearing on the operation of the Connection. The manner of the storage and management of this relationship attribute is open to the system requirements, allowing for custom graph implementations that can operate with the similar or derived Relationships types. One example of this presented in this project in Section 4.2.4. is the *HistoryEdge* or *DeepEdge*, that stores a list of Relationships that have been set on the edge over time and that can be inspected to observe the progression of the Relationship between the Entities on the edge over time.

The model also makes a distinction between the storage of direct and indirect edges. Direct edges store direct connections between two nodes. Indirect edges store a connection between two other nodes. See Figure 3.1., for an illustration of this, this is explained in more detail in the following section.

The RelationshipGraph supports generic nodes and relationships. This grants the storage of different types of relationships and nodes that can be created based on requirements. For example, a node may represent a character, location or group, and a relationship may define opinions or actions characters have to each other. The graph only requires that the generic types implement certain interfaces for them to work properly, otherwise their implementation is left to system requirements. The consequence of this is the model is not capable of optimizing based on specific implementations. Another consequence is that the model makes no explicit requirements for how unique nodes and relationships may be identified. The models interfaces for nodes and relationships require the implementation of a method that performs this, however the model has no control over how "correct" this is implemented.

Direct and Indirect Edges

When capturing information about relationships, it must be noted that relationships do not exist in a vacuum. Other parties may be aware of the relationships between others. In addition to this, this 3rd party awareness is not always accurate. For example, there may exist a relationship between two characters, Woody and Buzz that indicates that Woody considers Buzz to be his best friend. A third character may also be aware that Woody and Buzz are friends, but may have a different scale for the strength of that friendship (e.g. acquaintances vs best friends) or may have a completely different observation (e.g. may think Buzz and Woody are Rivals).

The model makes consideration for this requirement for awareness of not only a nodes edge's to other neighbouring nodes, but also for the need to store the edges of other nodes, by supporting the concept of direct and indirect edges:

- A *direct* edge in this respect is an edge that originates from one node to other neighbouring nodes within the graph. Hence, they "directly" connect an Entity to another Entity i.e. the Source or "From" attribute of the edge is the same as the node that is storing the edge.

- An *indirect* edge in contrast, is an edge that is not originating from the current node i.e. the Source or "From" attribute of the edge is different from the node that is storing the edge.

This allows for edge's to be stored in one data structure together by using the Source attribute of the edge to distinguish between the two types. The RelationshipGraph makes no requirement that an indirect edge match the true direct edge between two other nodes in value i.e. if the direct edge from node A to node C indicates a relationship value of "FRIEND", another node B storing an indirect edge from node A to node C is allowed to have a different value for the relationship attribute. See Figure 3.2., for a visual representation of the concept of direct and indirect edges that a node may track.

To clarify this further, consider a Relationship that carries a type and a strength value (e.g. a relationship of ["ENEMY,10"]) and 3 character entities: Alice, Tom and Charles. A relationship of ["FRIENDS,4"] may exist from Tom to Charles. This reflects that Tom considers Charles to be a friend, with the strength of that relationship being "4". This relationship is stored in an edge that leads from Tom to Charles and

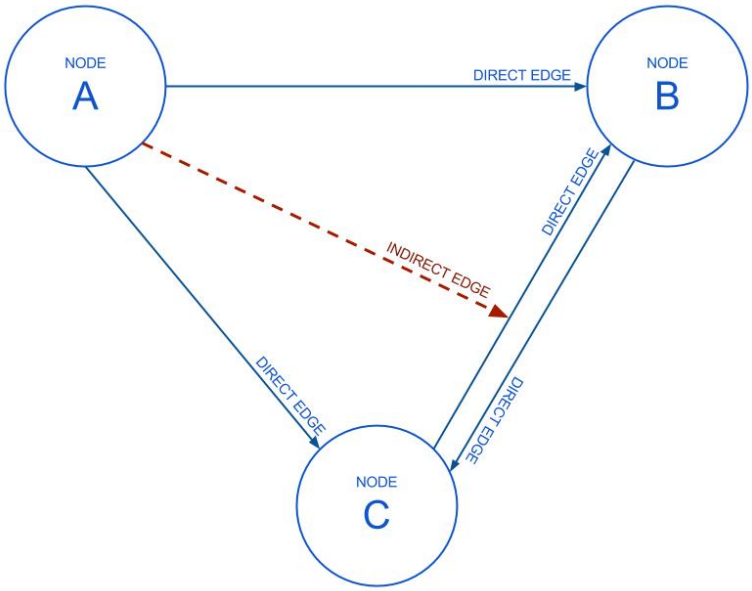

Figure 3.2: Representation of a simple RelationshipGraph. Each Node represents a character (or group). The edge's represent inter-relationships. An indirect edge is defined as an observed edge between two other nodes, represented here by the red dashed line that points to the edge from node C to node B.

is considered a direct relationship as the source of the edge (Tom) is the same as the owner of that edge (Tom). Now consider Alice, who also stores a relationship of ["RIVALS,3"] from Tom to Charles. The edge would indicate that the source is Tom and indicates a connection from Tom to Charles, however the owner of the edge is *Alice*. This indicates that *according to Alice*, Tom considers Charles as a rival - with the strength of that relationship being "3". This is clearly not the same as the true relationship between Tom and Charles is ["FRIENDS,4"] and not ["RIVALS,3"], but the model permits this as the value stored is an *assumption of the relationship between Tom and Charles* on the part of Alice.

This means that the model makes it possible for characters to be mislead about the relationships of others or just the strengths of those relationships - leading to more complex interactions than if we just carried direct references to the true relationship values.

3.1.2 Graph Interface

It is important to have a robust API for requesting information from the model, as well as to update the model. The RelationshipGraph provides an extensive series of functions that allow Entities to update and retrieve their Relationships to other entities.

The graph operates on the assumption of certain interfaces being implemented by nodes, edges and relationships for them to be compatible with the graph. The actual implementation of these interfaces is separate from the implementation of the graph, but will ensure that the components work properly. Nodes and relationships must implement an identification method that is used to compare two nodes or two relationships to see if they are a match - for example, this identification operation can be implemented using unique identifiers for each node based on the number of nodes, or a by using a enumerated type flag for relationship types.

There is no strict requirement from the graph on how to implement this identification method. Each edge however, relies on these interfaces to implemented. Edge's also make a requirement for source and destination nodes to be provided as well as a relationship value. As with nodes and relationships, there is no requirement for how these interfaces are implemented. For example, the project offers two implementations of the edge interface: one is a simple edge that stores a single relationship value, while another stores a list of values as a log of previous relationship states that can also be queried, with the most recent relationship being directly accessible.

How the graph stores, manages and queries the data is transparent to the components. If an Entity wants to know what its relationship is to another Entity, it makes a request through one of the available methods exposed by the RelationshipGraph. The graph can rely both on the Entity's direct Relationship to other Entities as well as by checking what the requesting Entity "knows" about other Entities Relationships (indirect Relationships). For example, a blacksmith may offer a discount to a buyer not because it has any direct relationship to the buyer (indeed, the blacksmith may have never met the buyer before), but because it "knows" the buyer is a regular customer of another blacksmith based on an indirect relationship (possibly obtained via observation of an interaction between the buyer and another blacksmith) that is had stored for itself. (As a quick note, considering the previous description of how the model supports deception of Entities, the Blacksmith may have an incorrect "knowledge" of

the buyers relationship with the rival Blacksmith - and in this case acts against its own interests.)

The following pseudocode provides an outline of how the RelationshipGraph may provide an interface to determine if one Entity has a Relationship with another i.e. a direct edge exists from one node to another node:

Data: source and destination entities
Result: boolean of connection existence between entities
initialization;
while *not at end of source's connections* **do**
    read current;
    if *current source and destination match entities provided* **then**
        return true boolean value;
    else
        go to next connection;
    end
end
return a false boolean value;

Algorithm 1: Basic algorithm to find if a connection from one entity to another exists

3.2 Group Behaviour

As noted in section 2.4, groups are an important aspect of relationships between individuals. Group dynamics may be supported in the model by implementing nodes as representative of groups. This is possible due to the generic nature of the nodes. Entity types that differentiate group nodes from individual nodes and special relationships such as "MEMBER" or "FOLLOWER" that define connections to those groups makes this a straightforward extension of the basic structure. In addition, another way to define a group type is by making them a derivation of the standard Entity type and overriding the standard behaviour with custom group-like behaviours. See Figure 3.3. for an illustration of how groups are captured by the model.

Custom group behaviours such as broadcasting messages to members of a group, or having individuals make decisions based on their group associations, are all possible by building on the atomic operations made available by the core graph functions.

Sending a broadcast message to all members of a group involves passing the message to the group entity, which would in turn pass it on to all those entities that have a "MEMBER" relationship to it by requesting those entities from the graph. Or, considering the blacksmith example again, the blacksmith may decide to refuse to offer services to a buyer if that buyer has poor relations with a guild that the blacksmith may be a "MEMBER" of. This leverages the existing model's support for relationships without the need for extra definitions and memory allocations within the model. An entity could make it's decisions based on the relationships its parent group has with other entities - essentially creating a sort of collective consciousness that all members share and refer to. It also allows other entities to maintain their own relationship with a group instead of with each member of that group, further reducing complexity by using groups as clusters of nodes or as hubs.

3.2.1 Applications of group behaviour

Efficiency

Group behaviour can be used in the creation of smaller graphs by having characters that act based on their group affiliations are instead of their individual relationships. If a character has no individual relationships, then that character could store a reference or identifier to a group that they are members of and use those identifiers to request information from the graph instead of storing many individual characters that all ultimately share the same behaviour that can be defined in a group. This would be ideal for character interactions similar those found in action games where enemies are organised into factions and share behaviours and common targets and objects.

Extensions

An extension of this grouping concept is that other types of nodes may be defined with their own special relationships as needed depending on the system. For example, defining locations as an entity type allows characters to have relationships to certain areas, or even types of areas, in the game world (such as a church, monastery or home); having items as another type allows characters to have bonds with their favourite vehicle, weapon or other items. At a high and abstract level you could even have

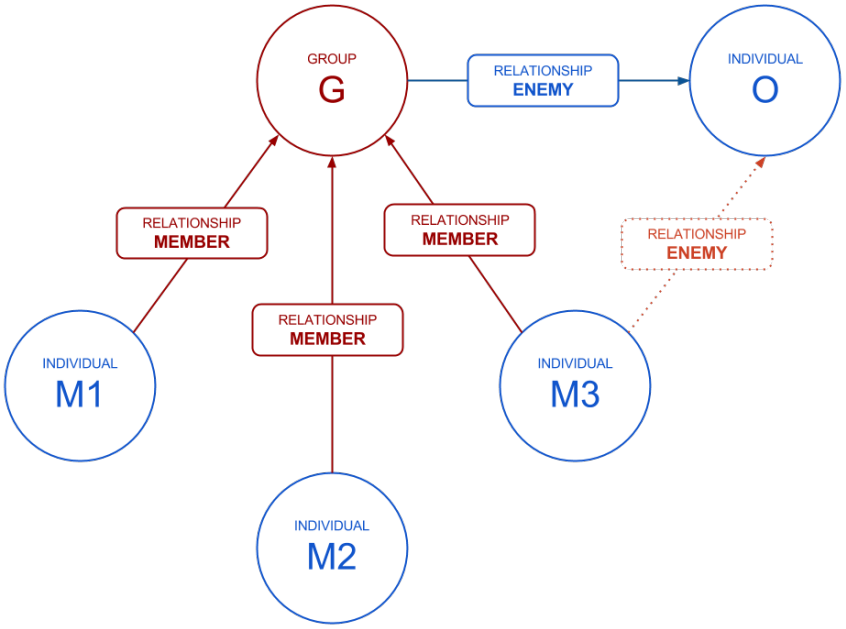

Figure 3.3: Representation of how grouping can be captured in the model. Each Node can represents a character or group, in this diagram the Node "G" represents a group and other Nodes specify that they belong to the group "G" by specifying the "MEMBER" relationship to it. In this example, the Nodes "M1", "M2" and "M3" are "MEMBER"'s of the Group represented by Node "G". Node "G" identifies the Node "O" as an "ENEMY" of the Group, all "MEMBER"'s of the Group "G" can inherit this relationship due to their association with the Group i.e. as "O" is an enemy of the group "G", it is therefore the enemy of all members of that group.

entities with relationships to ideologies or brands.

3.3 Messaging System

The model offers a basic messaging system for sending messages from one node to others using on the graph. At the core of the messaging system is simple node to node delivery of messages. Each node must implement a *HandleMessage* method to accept messages sent to it. All messages must implement the message Interface. As with the node and relationships, the contents of this message are not defined by the model.

The intention of this design choice is for the messaging system to act as a simple but solid foundation for more complex systems and functionality. For example, the graph is capable of broadcasting messages based on relationships to and from an entity by combining this basic messaging system with the graph interface mentioned earlier. The graph can also allow entities to update their relationships based on information gained from messages.

3.3.1 Learning support

Relationships are not static and evolve over time. A method for supporting the adoption of new relationships and connections means that the graph is capable of supporting dynamic and complex inter-relationships that change over the course of multiple interactions between entities. The model (and the graph) achieves this by combining the use of the messaging system and the graph query functions to enable entities to update their own relationships.

An example approach taken within this project is to make use of a Relationship that defines a relationship type and a corresponding strength value (e.g. ["RIVALS", 4]). By passing a Connection that defines source and destination entities as well as a Relationship in a custom message, we can define an implementation for the *HandleMessage* method that accepts this message and makes a choice of what to do with the new information after evaluating it. For example, given a Connection that states that the relationship between two Entities Bob and Charlie is ["RIVALS",4] and sending this in a message to the Entity Alice, the entity can compare the given Connection to what is in the graph. If Alice agrees with the connection information provided, they may

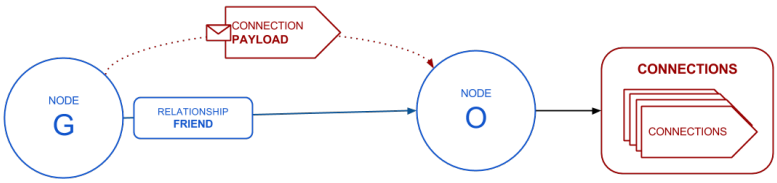

Figure 3.4: Representation of how simple learning works at a high level, a connection is passed to the destination Entity as the payload of a message. The receiving Entity evaluates this Connection and discards or adopts it to its Connection list

choose to reinforce their knowledge by adopting the new strength value. If the entity finds that the information provided conflicts with it's pre-existing values, then it may choose to ignore or adopt the new values based on other analysis (e.g. does it trust the source of the information? Does it have a gullible character trait?).

See Figure 3.4 for a basic illustration of how messages and connections can be used to implement simple learning.

3.4 Illustrating the Model

In many games the relationships between characters are static and often don't reflect or adapt to events occurring in the environment and between other characters. These static relationships are often initially realistic and believable, with characters interacting with one another based on their relationships and desires such as rival gangs attacking each other or fighting over disputed territory. However, the suspension of belief often breaks down when characters retain their relationships irrespective of what occurs within the game, indicating that the characters are scripted or are waiting for a predetermined action to take place before changing their behaviour. This leads to repetitive and obvious behaviour that is not engaging for the player to interact with without extensive, detailed investment in a wide scope of possible interactions that may cause changes in relationships. Without behaviour resulting from dynamic relationships the characters break the illusion of intelligence. Some examples of characters that can benefit from a relationship model to diversify their behaviours are described next.

3.4.1 The Trader

The Trader is an NPC archetype often encountered in games, that allows the player to sell and buy items that they find and use in the game. These NPC's are often confined to their stores, only awaking when prompted by the player. In other games, the NPC's have traits and objectives that determine where they will be within the game environment at a given time, possibly travelling from village to village to buy and sell their wares. Other variations on this archetype are the Blacksmith that often operates similarly to a trader but specializes in buying and selling weaponry and armours.

When players interact with a Trader character, the prices of items, and the behaviour of the traders to the player are often rigidly defined. Players learn to identify which trader has the best deals for certain items, and plan their interactions accordingly. Some incentives may be scripted into certain traders, such as discounts or gift items based on "loyalty" but these are not real relationships and the predictability of their behaviour limits true interaction with the NPC. This often leads to tediousness as the interactions are rote, and players treating Traders as objects rather than characters.

By applying the relationship model more interesting scenarios can be easily generated. A trader can be created that increases discounts as a function of the strength of the relationship between the player and the trader. This strength could be reinforced based on how often the player purchases from the trader. If the trader is aware of the player's relationships with another character they could offer different items for sale (e.g. black market weapons if the player has an antagonistic relationship with local law enforcement), or certain items could be removed based on other interactions based on unforeseen consequences (e.g. killing a farmer that supplied the trader). Or the player may inform the trader of their connections to certain other important characters (whether or not this information is true or not is another matter). The trader may also avoid selling their goods at certain locations or to groups based on their relationships to the those entities, and may change their offers based on changing relationships to these entities based on what happens dynamically in the game.

3.4.2 The Mob

In games characters often act in groups and behave based on their relationships with other groups and also on the hierarchy within their groups. For example, in the Halo [5]

games, enemies have certain behaviours based on their groups (type) and the hierarchy within and between those groups. Grunts act courageous with superior numbers or with stronger support, and flee otherwise. Brutes attack in packs but are antagonistic towards Elites. Brutes and Elites also have hierarchies within their groups with Brute Chieftains at the top of the pyramid often waiting after lower ranked Brutes have engaged the player before attacking.

Not only is it possible to replicate such behaviour using the RelationshipGraph, but it is also possible to extend it further by having individuals combine their individual and group relationships to determine how to act. For example, an individual within the group may change their relationship to the player if they are saved from death from a common enemy, and choose to not target them. In addition the higher level NPC's can issue instructions and orders to lower level NPC's using the messaging system and broadcasting through the group support or communicating directly to one another and choosing what information to provide or leave out based on their relationships e.g. informing a "rival" NPC to attack when it is detrimental to do so.

3.5 Model Architecture

The high-level architecture of the model can be defined based on the requirements identified. Entities each store their own Connections that they are aware of and each of these Connections stores at least one Relationship value for each Connection. Messages can be passed from Entity to Entity by passing the message through the graphs message handler that routes messages to the destination Entity's *HandleMessage* method. An Entity can update the graph and retrieve information to make decisions by looking up their Connections using an identifier method that must be implemented by the Entity. This identifier method is used to look up the Entity. The usage of the identifier is transparent to the Entity, they only implement this method to uniquely identify a specific Entity. A similar method is also used to uniquely identify Relationships.

In order to create an efficient model, all the elements of the graph are stored in one data structure. Direct and Indirect edges make use of the same Edge structure and can be stored together. In addition the implementation of the Relationships, Connections and the Entities is not defined in the model, so any attributes that are used to capture these are not constraints on the use of the model. If keeping track

of previous relationship values is required, that can be captured with a custom Edge implementation. The graph at it's core also works primarily with interfaces, so long as those interfaces are implemented then the model will function. While indirect edges are an important feature of the graph, they also have some more expensive functions. However there is no requirement for the use of indirect edges - the graph can still operate as a standard graph. Messages have no constraint other than the implementation of an interface that the graph uses to pass messages from one Entity to another. This allows the messaging to work with other pre-existing messaging systems.

3.6 Conclusion

The high level design for the model has been presented, demonstrating how the concept of a graph can be extended to take consideration for the needs of indirect edges in the model. The model's architecture explains how the model could be implemented in a single data structure that would allow for efficient querying of information without having to generate large nodes with complex requirements for implementation.

Chapter 4

Implementation

This chapter presents the implementation of the RelationshipGraph. First, the choice is platform is presented, followed by an in-depth description of how the different components of the graph were implemented. Finally, the visualisation of the model is presented.

4.1 Platform Selection

Unity [35] was chosen as the platform for the implementation of the prototype. The engine supports quick iterative development as well as an asset store with hundreds of freely available resources that would be instrumental in the development of the prototype [36]. A previous student, Tiarnan McNulty, had previously worked on a project using the Unity engine [37]. This allowed for a ready made environment for testing the model.

The Unity game engine natively supports 2 languages [40] and these can be used to create custom components for use within a game. The first is UnityScript, a flavour of JavaScript designed specifically for use with the Unity. The second, the C# language, was chosen for the implementation of the model; the C# language is a strongly typed language with support for many traditional object oriented programming features important for the implementation of the model (such as interfaces, generics and inheritance), as well as a strong support for data structures in its collections library that can be used as a foundation and extended [41].

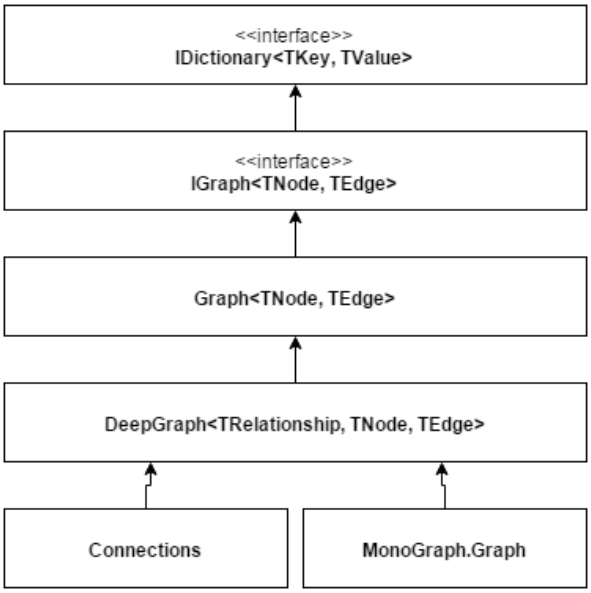

Figure 4.1: All Graph implementations in the prototype are extensions of a base Graph type that implements the IDictionary¡TKey, TValue¿ interface, and a DeepGraph class that extends the Graph with support for multiple TValues per TKey - this is how multiple connections are stored per entity within the graph.

For the purposes of the prototypes generated to test the graph the resources that came with Unity were sufficient. The conversation demo utilises a free model [42] available from the Unity Asset Store that provided some pre-created gestures to provide feedback when querying the character.

4.2 Graph Implementation

The implementation of the graph was split into four main components. Relationships, Entities (Nodes), Connections (Edges), and the Graph itself. Each of these is implements an interface that the other components can use with confidence.

4.2.1 Graph

The graph is treated as an extension of the generic **IDictionary¡TKey, TValue¿** interface [43]. This represents a generic collection of Key/Value pairs. *TKey* and *TValue* specify the types of the keys and values. For the purposes of the prototype, *TKey*

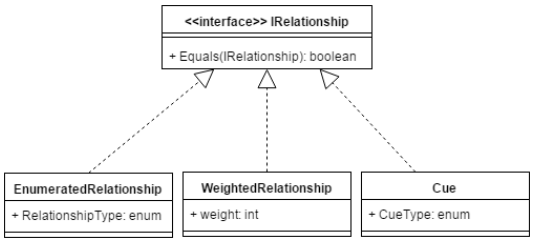

Figure 4.2: **IRelationship** Interface used in implementations of Relationship classes. The Equals method is used to identify a unique Relationship and is used when traversing the graph.

are Node types that implement an **INode** interface (discussed in section 4.2.3) and *TValue* are Edge types that implement an **IEdge¡INode, IRelationship¿** interface (discussed in section 4.2.4) which in turn depend on node and relationship interfaces. Internally the graph stores all the nodes, edges and relationships of the graph in this dictionary implementation. The support for generics is maintained in the implementation allowing for use of custom types with the graph.

The **IDictionary** interface requires certain methods to be implemented and this are handled in a base "Graph" class. This base Graph class is then extended to a "DeepGraph" class that manages multiple *TValue*'s per *TNode*. This separates the concerns of implementing the required methods and properties of the **IDictionary** interface to the Graph class, and the special extensions to that underlying structure that allow for a graph with multiple edge's per node. This DeepGraph class is then used directly or as a foundation for other graphs. This is illustrated in Figure 4.1.

4.2.2 Relationships

Relationships implement an **IRelationship** interface specifically defined for the project. The **IRelationship** interface only requires one method to be implemented that is used to identify unique Relationships. This method is used when traversing the graph to find edges or nodes that match a given relationship.

The model, and by extension the graph, make no requirement on how this distinction between Relationships is defined. For the purposes of the prototype the following examples were created:

- *Enumerated Relationships*: these rely on an enum RelationshipType value to

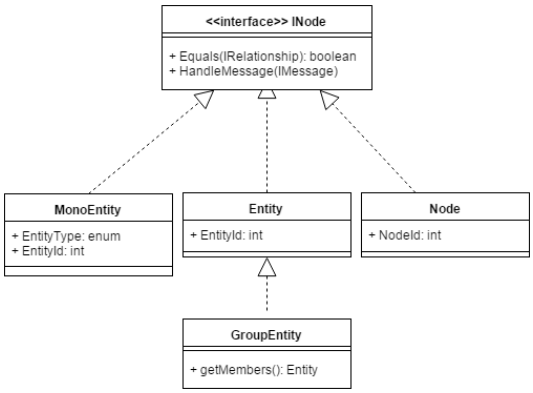

Figure 4.3: **INode** Interface used to implement Entities. The Equals method is used to identify a unique Relationship and is used when traversing the graph. The HandleMessage method is used to handle messages passed to the Entity.

identify relationships. For example, there are relationships such as "FRIEND", "ENEMY", "MEMBER", "BROTHER", "LEADER", "CRUSH", etc. The method compares these enumerations to see if they are the same.

- *Weighted Relationships*: these are an extension of the Enumerated Relationships that add a *weight* value to indicate the strength of the relationship. However, this weight is not used in the required method implementation.

- *Cue*: this was created as an example of using the IRelationship interface to define another type of data that maybe of use within a graph. In this case Cues represent actions that occur in the game world (e.g. one character attacking another) that characters may track.

4.2.3 Entities (Nodes)

Entities are objects that implement the **INode** interface. This interface is virtually identical to the **IRelationship** interface that has a method that must be implemented that is used to uniquely identify Entities. This method, as with the Relationship method, is used when making queries to the graph. Also, as with Relationships, there is no requirement on how this is achieved.

The following are examples of Entities that were created for the purposes of the prototype:

- *Entity*: a basic version that sets a unique integer identifier for each Entity which is then used in comparisons
- *MonoEntity*: a special version that derives from Unity's MonoBehaviour class. This provides the Entity with all the methods available from the Unity class.
- *GroupEntity*: an extension of the standard Entity type that has special Group-like behaviours such as broadcasting messages.

Groups

To support Group Behaviour two approaches were tested: a GroupEntity class and an Entity class with an EntityType attribute. The GroupEntity class is an extension of the standard Entity class that overrides the default Entity behaviour to implement behaviour specific to Groups such broadcasting messages to members of that group. The prototypes used the second approach, which involved defining an EntityType that would be used to identify an Entity as either a Group or a single Entity and choose the appropriate behaviour. The EntityType defines only two values: "GROUP" and "SINGLE". Regardless of approach however, this allows the graph to store Entities of whichever type within a single graph. This was important to the design of the model to provide for compactness and to allow all relationships to be stored in one area for efficient queries. However, it is also possible to have group types stored in their own graph as well if required.

Other Node Types

These two approaches demonstrate the capacity for the Nodes of the graph to represent much more than just Entities and Groups. A Node may be used to represent a location or an item that the character has a relationship with. Using the EntityType attribute allows these new types to be added quickly for testing, and an extension like the GroupEntity demonstrates that more specific implementations are possible (e.g. LocationEntity, ItemEntity).

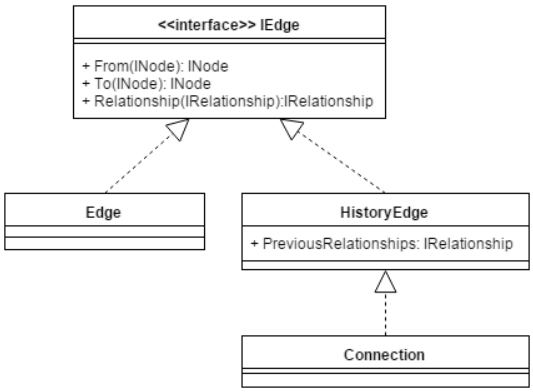

Figure 4.4: **IEdge** Interface used to implement Connections. The "From" method defines the source of the Connection, "To" defines the destination Entity and the Relationship method defines a relationship between between the two Entities. When a comparison is made between two Connections, the graph relies on the implementation of the Equals methods of the **INode** and **IRelationship** interfaces

4.2.4 Connections (Edges)

Connections are objects that implement the **IEdge** interface. This interface requires source and destination Entities (nodes) and an accessor to a Relationship to be defined. Each of these Connections is stored in a list for each Entity in the graph class. If the source Entity of the Connection is the same as the Entity it is stored with then that is defined as a *direct* Connection from the source Entity to the destination Entity, otherwise it is defined as an *indirect* Connection that the Entity is aware of.

The Graph has no requirement to how the Entities nor the Relationship are managed internally by the Connection, only that it accept data passed or requested of it. This allows the Connections to manage this information in as much complexity as possible, one example of this is the HistoryEdge type which stores a history of Relationships.

HistoryEdge / DeepEdge

When a Relationship value is set on the HistoryEdge it is added to a list of Relationships. This allows the HistoryEdge structure to monitor the changes of the Relationship between the Source and Destination Entities over time. The Graph expects just a single Relationship value when it makes a request to a Connection and the HistoryEdge

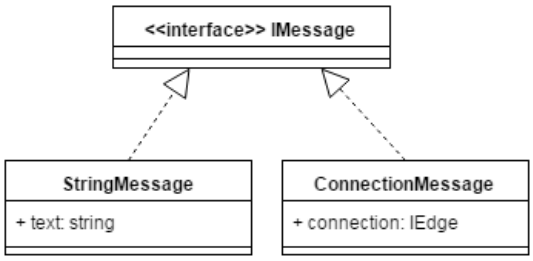

Figure 4.5: The **IMessage** Interface is used in the implementations of Message classes that are passed to Entities through the HandleMessage method of **INode** implementations.

returns the most recent Relationship value, but also offers other methods that allow the previous states to be queried - allowing the Entities to be aware of their Relationship history with other Entities.

4.3 Messaging System

The graph allows messages to be sent to one or more Entities based on the relationships between those Entities. Fundamentally, this relies on a basic Entity to Entity messaging functionality that is combined with the methods that expose Connections between Entities.

A Message is an implementation of the **IMessage** interface (See Figure 4.5). This is an empty interface with no requirements on what the contents of the message will be. A message is passed to the HandleMessage method that is a part of the **INode** interface and is required to be implemented by every Entity. The node can then act based on the message received.

The prototype used two implementations of the **IMessage** interface:

- *StringMessage*: carries a simple string payload.
- *ConnectionMessage*: carries a Connection as its payload. This can be used by the recipient of the message for activities such as learning new relationship (covered in section 4.4).

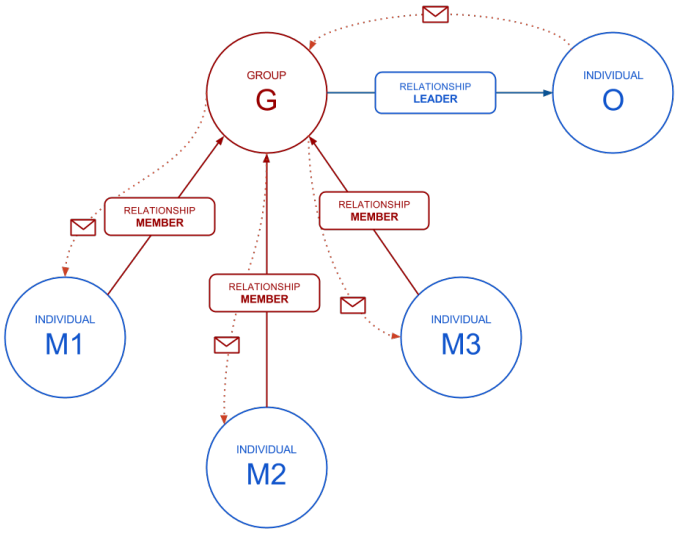

Figure 4.6: Representation of how broadcast messages are implemented in the Graph. A message is sent from Entity "O" to the Group Entity "G". This in turn "Broadcasts" the message to each of its members i.e. those Entities that have a "MEMBER" relationship *to* the group Entity

4.3.1 Broadcast messages

Broadcasting messages to the members of a group, or to a set of Entities with a common Relationship (for example, sending a message to support in a combat scenario) is achieved by querying the graph for the Entities that match the given relationship and passing the message to each Entity's HandleMessage method. An illustration of this can be seen in Figure 4.2.

4.4 Learning Systems

The prototype offers a simple Learning System as an example of a combination of all the components of the graph to offer some complex behaviours as a result of relationships between entities. A Connection about two Entities is defined with a certain Relationship, this relationship is passed to an Entity using a ConnectionMessage. The Entity compares the Connection in the message to its known connections by querying the Graph. The Entity may then choose to adopt this new connection data into its known connections, or ignore it.

4.4.1 Relationship Evaluation

When a Connection is received by an Entity, it will query the Graph for a Connection that is a match. A match is considered another Connection with the same Source and Destination values. These matches are determined using the mandatory method that is defined in the **INode** interface.

If a pre-existing Connection is found, the next step is to compare the Relationship types and weights. If the Relationship values are the same then the two Connection's match, if they differ then there is a conflict between the Connection data provided and the Entity's pre-existing knowledge. For example, *Relationship.RelationshipType.FRIEND == Relationship.RelationshipType.FRIEND* is considered a *match* whereas *Relationship.RelationshipType.FRIEND == Relationship.RelationshipType.ENEMY* would be considered a *conflict*.

4.4.2 Adopting new Relationships

To resolve a conflict, the Entities in the prototype compare the weights of the two differing Relationships and adopts the Relationship with a greater value. This is a gross simplification; however, it is ideal as a demonstration of how an Entity can "learn" about a new Relationship.

If a matching Connection doesn't exist in the graph (i.e. the Entity has no Connection stored with a matching Source and Destination Entity), the Entity for the purposes of the prototype (and to keep things simple) will adopt the new Connection data. However, we can see that this adoption could be extended to be based on the traits of the Entity (is it gullible?), or how much it trusts the source of the Message (which in turn may be based on the Relationship the Entity has to the source of the message e.g. believe a friend, ignore an enemy).

4.5 Model Visualisation

Two demo's were created to demonstrate the potential of the graph. The Sphere demo is used to demonstrate the graph working with a large number of characters, as a reference for each character to determine their behaviour and to demonstrate basic messaging. The second demo, Bob the Guard, is used to demonstrate more advanced

functionality such as storing previous relationship values, the learning system, messages and group behaviour.

4.5.1 Sphere Demo

The Sphere demo is an environment with 200 character spheres with randomly specified relationships to a player controlled character sphere. The character sphere's have certain steering behaviours based on those in Buckland [15], with the character spheres choosing steering behaviours when they come into range of the player character. The choice of steering behaviour is based on the Relationship to the player character and is determined at the beginning of the demo. When the player comes within range of another character, a query is made to the Relationship Graph to determine the Relationship and a behaviour (PURSUIT or EVADE, with colours to demonstrate either behaviour) is chosen based on this.

In addition, at the beginning of the demo, characters that are close to each other will share their Relationship to the player character with another adopting new Relationships. This communication and adoption is demonstrated using by characters changing their colours to yellow when they have adopted a new behaviour.

4.5.2 Bob the Guard

Bob the Guard is a demo that features a character that the player can interact with by sending messages and asking questions through an on screen interface as demonstrated in Figure 4.7.

The Bob character stores his relationships to other characters in the Relationship-Graph. Even though the other characters are not displayed on screen, they exist as characters within the graph. Relationships among these characters is also captured in the graph.

When a ConnectionMessage is sent to Bob, he performs a learning behaviour as described in Section 4.4. In addition to this, Bob can analyse previous relationship states when queried using the "History With?" button by comparing the selected character and selected relationship type to it's internal model. Group behaviour is demonstrate by asking Bob what their relationship is with other Character's - one character is not in Bob's model (he has no knowledge of the existence of the other character), however

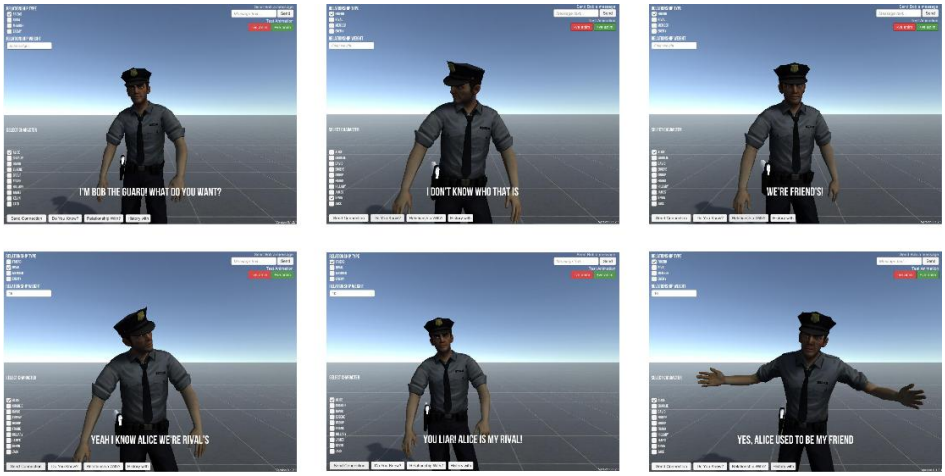

Figure 4.7: Screenshot of Bob the Guard. The player can use the interface to communicate with the character. They can send text messages to Bob using the input on the top right, or a Connection message constructed from the options on the left and submitted with the Send Connection button. Questions about Bob's relationships to other characters can be submitted using the other available buttons and his responses are demonstrated in the screenshots above.

Bob will identify that character as having a certain relationship based on the relationship on of his groups has with that character i.e. if a character is an enemy of the group Bob belongs to, then that character is Bob's enemy.

Chapter 5

Evaluation

This chapter presents an evaluation of the project. First, is an assessment of the relationship model in the portrayal of believable social behaviour among game characters successfully, as well as how successful the prototype is to meeting the objectives of the project offered in Section 1.2. Following this is a comparative analysis to other design approaches from other games as well as discussions on how these other systems could be implemented using the relationship model. Following this is an examination of the performance of the prototype, with suggestions on how to tune the implementation. Finally the major shortcomings of the implementation are discussed.

5.1 Believable Social Behaviour

Relationships between characters provides a foundation for the creation of behaviours that is often only possible with scripted behaviours or highly customized systems. Complex behaviours are made possibly by taking advantage of the generic storage of information by the graph and by combining components offered by the core functionalities of the graph. Characters use the graph to determine how to behave based on current or previous relationships, or how to communicate information to each other These behaviours not easily possible with scripted behaviours and are not as dynamic. The complexity of the relationships is only limited by the types of connections and relationship defined.

A direct comparison is made with the *Comme il Faut (CiF)* AI system implemented

in the social simulation game *Prom Week* [9]. The *CiF* system has high level components that include: Relationships, Social Networks, Statuses, Traits, a Social Fact Database and a Cultural Knowledge Base. The Relationship component offers "binary, reciprocal and public connections between characters", with the three relationships being "friends", "dating" and "enemies". The Social Network component comprises "scalar, non-reciprocal and private feelings from one character toward another", these three networks are identified as "Buddy", "Romance" and "Cool". Statuses are used to reflect a characters "feelings", or to represent internal and external states such as embarrassment or popularity, and Traits are used to define a characters personality using permanent attributes such as "Competitiveness" or "Attractiveness". The Social Fact Database stores a history of interactions between characters, while the Cultural Knowledge Base provides relationships each character has to objects in the social world that each has a "popular" opinion associated with it e.g. "lame" objects [9].

The following example is used to demonstrate the components in *Prom Week*: two characters Simon and Naomi may have a relationship of "friends". Naomi has a trait of attractiveness, and Simon a status of a "crush" on Naomi resulting in high "romance" network values for Simon towards Naomi, with the Cultural Knowledge Base indicating that Simon likes "lame" objects (like calculators) and Naomi prefers "cool" objects (like footballs). The Social Fact Database stores an embarrassing previous encounter between Simon and Naomi. This shows how the components of the *CiF* system can be used to create social situations with a potential for drama.

However, this complexity of this system is reflected by the number of components that must each be kept up to date when characters interact with another or with social items. This complexity can be reduced by using the *RelationshipGraph* to capture this information in one structure. Graph Nodes can be defined as either representing an item or a character. Character nodes can store traits and statuses in a similar manner to the *CiF* system, and item nodes can store their cultural significance (i.e. its popular opinion) in a similar fashion to the Cultural Knowledge Base. So we shall focus instead on how the Relationships, Social networks and and Social Fact Database can be implemented using the RelationshipGraph.

Relationships are a simple conversion as they already denote relationships from one character to another such as "friends". However, similar relationships can also be defined for characters towards items - this covers the other feature of the Cultural

Knowledge Base that also tracks characters relationships towards social objects e.g. a character can be a "FAN" of a calculator object indicating that they like that object. This allows use to store all the relationships, items and characters in one structure.

Social Networks values are used to track private feelings to other characters that are not reciprocated back, this means that the feelings are in one direction leading from one character to the next but not back again. This is a functionality immediately offered by the graph being a *directed* graph - allowing unidirectional connections between nodes. This observation demonstrates that Relationships according to the *CiF* system are bidirectional connections between nodes i.e. "Friendship" between Simon and Naomi is a "Friend" relationship leading from Simon to Naomi, and from Naomi to Simon. While a Social network value is unidirectional i.e. "Romance" is a relationship from Simon to Naomi, that Naomi is unaware of. This break down of the two demonstrates how both can be captured in the RelationshipGraph.

The Social Fact Database is a repository of previous encounters between two characters. These are stored as events such as "Simon misundertood Naomi asking for help on homework as a romantic advance" [9]. There are two approaches that can be made to capture similar information in the RelationshipGraph. The first is to extend the connections between two characters to store this additional information along with the relationship value. The second is to ignore the information, and instead focus on storing the *consequences* of these actions - which is typically a change in the relationship value using a HistoryEdge. Characters are more likely to remember that they "used to be friends" than the exact event that caused the change in the relationship. However, both of these approaches could be used together as well.

The *CiF* system also makes use of so called *social games* that are used to create *social effects*. For example, if a character plays a game called "Share Interest" with another, the resulting effect could be the two characters both liking a "cool" or "lame" object and bonding over a common interest [9]. This can also be captured by the RelationshipGraph by making use of the Message system to share messages about other items in the graph and relationships to those items (and since the items are also nodes, a structure such as a ConnectionMessage may be used for this purpose) and updating those characters states and relationships as effects of receiving this messages. These social games have 3 roles: an initiator, a responder and a possible third party or observer. This can also be captured by the message system by having "public

messages" that can be observed by nearby characters or by having the responder or recipient passing the information on to third parties.

This demonstrates that the *CiF* system can be replicated using the RelationshipGraph. By taking advantage of the other features of the graph, other advanced social behaviours could be added such as having characters that operate in cliques by using the grouping features, or having characters that can spread rumours or gossip with misleading relationship information by using indirect edges. In fact, a direct improvement is with the Relationships of the *Comme il Faut* system. *CiF* assumes that relationships that are bidirectional are the same from one character to another, when in reality characters may be friends but the social hierarchy of those friendships differs from character to character. In other words, Simon and Naomi may be friends but Simon may value that friendship with Naomi "more" than Naomi may reciprocate it.

5.2 Complexity Analysis

Function	*Size*	*Worst Case Complexity*
GetConnections	# of Edges	1
AddConnection	1 Edge	1
RemoveConnection	# of Edges	E
GetDirectConnections	# of Edges	E
GetInDirectConnections	# of Edges	E
FindByRelationship	# of Edges	E
FindByRelationshipToOther	# of Edges	E
FindByRelationshipHistory	# of Edges	E * R
SendMessage	1 Message	1
BroadcastMessage	# of Edges	E * N

Table 5.1: Complexity analysis of common relationship graph methods. E are the number of Edges or Connections in the graph, N are the number of Nodes, R are the number of Relationships.

Table 5.1 shows the complexity analysis of the RelationshipGraph most common methods. In this table, E represents the number of Edge's or Connections in the graph, R represents the number of Relationships and N represents the number of Nodes or

Entities. GetConnections performs a retrieval of all the Connections using the given Node as an index. In contrast all other retrieval methods for Direct and Indirect Edge's or based on relationship values requires a traversal of all the Edge's for each Node to check the attributes of each Edge - in a worst case scenario all the edge's of the graph may be stored for just one node. When checking the relationship history, we must also traverse through the set of stored relationship values at each edge. Sending Messages is handled by directly passing the message to the destination node, however Broadcasting messages requires a combination of one of the previous retrieval methods followed by a submission to each node.

5.3 Performance

The Unity Profiler [45] was used to evaluate the performance of the graph in terms of CPU usage and memory allocation within the Unity Engine. The efficiency of the model was on of the primary objectives of the model and to meet this the model would require a small memory and computational footprint.

An empty scene was used as the benchmark for the tests and increases to the number of relationships, entities and connections were made to examine the performance of the graph against this benchmark. As each connection may have multiple relationships (Using the HistoryEdge described in section 3.1.1), and an entity may have multiple connections large increases in memory can be expected with for example: 100 entities, with 100 connections each and each connection storing 100 relationships will result in 100*100*100 = 1,000,000 relationships and 10,000 connections stored in the relationships which is a highly unlikely scenario to occur in actual use and is presented as an outlier.

5.3.1 Memory

Unity reserves memory pools and then manages allocations of memory within these pools to reduce frequent queries of the underlying operating system for memory. This means that the total amount of reserved memory rarely changes. However, the Unity profiler does offer feedback on the used memory from the reserve pools and this is used to determine how the graph affects the memory of the prototype. As this value

fluctuates during the running of the prototype, an average over the first 20 frames of the scene is calculated. Table 5.2 presents the memory usage captured by using the Unity profiler.

An important consequence of the design of the model is that large numbers of Entities and Connections affect the number of Relationships. In Table 5.2. a row indicating that there are 100 Entities, 50 Connections and 10 Relationships in reality means that there are 10 Relationships stored for each Connection, and 50 Connections stored for each Entity, resulting in 50000 (100 x 50 x 10) Relationship being stored by the graph. The consequence of this is evident with extremely large numbers of Entities that can cause memory to balloon in size with even just a small number of Relationships per connection. Otherwise, it can be observed that the size of memory required grows linearly with increasing numbers of Entities, Connections and Relationships.

In practice the number of Relationships will not likely exceed 4-5 per Connection and Connections themselves would rarely be expected to exceed 50 per entity. This is due to characters generally having memorable interactions with other characters in smaller areas, a game with an excess of 1000 characters would not require each character to each capture 999 connections to every other character in the game world. Instead, the character would have a smaller set of characters that they interact with frequently based on their environment. Connections to other characters are also only added to the model on demand. This means that if two characters never meet, or never establish a relationship then the model does not capture a "STRANGER" relationship, the connection is simply never stored. This would mean a realistic expectation of memory usage around 5-6MB which is more than adequate when taking into consideration the number of other systems that could make use of the graph data.

Other steps can be added to limit the number of Connections and Relationships each character stores by implementing functionality to "forget" connections and older relationship values if they are not refreshed by frequent interaction with other characters. For example, two characters may be established as friends that were once enemies. The previous relationship status of "ENEMY" can be eventually removed over time as the two characters "forget" they were ever enemies before unless this value is reinforced. If the two characters never interact ever again, the entire Connection between the two may be removed from the graph entirely.

Entities	*Connections*	*Relationships*	*Average Allocated Memory*
100	1	10	0.1 MB
100	10	10	0.56 MB
100	25	10	1.38 MB
100	50	10	2.69 MB
100	100	10	5.28 MB
200	100	10	10.52 MB
500	100	10	20.51 MB
1000	100	10	45.78 MB
5000	100	10	213.55 MB
10000	100	10	488.45 MB
100	100	1	1.71 MB
100	100	10	5.34 MB
100	100	50	19.35 MB
100	100	100	30.55 MB
100	100	200	61 MB

Table 5.2: Memory usage with different combinations of entities (nodes), connections (edges) and relationships. The number of relationships captured by the graph is significantly affected by the number of connections and entities e.g. 5 entities each storing 5 connections, with each connection storing 10 relationships would result in 250 (5*5*10) relationships.

5.3.2 Processing

CPU usage was also tested using the Unity profiler. A simple script responds to input and makes a request from the graph, each of these spikes in CPU usage is logged by the profiler and the speed of the execution and what percentage of the total processing time was consumed are captured in the profiler. Samplings from this log are then averaged and used as the results of the analysis presented in Table 5.3 for different combinations of entities, connections and relationships.

The tests were performed by first populating the graph with data and then making a query of that data. The query chosen for these tests is the most computationally intensive method in the graph as it must traverse the entire graph checking each Entities

connections. The method returns all entities in the graph that have had or currently have a particular relationship to a given Entity. This means that the query not only looks at all connections stored for each entity to check their destination attributes, but also checks the historical values stored in HistoryEdge extensions that are used as Connections by the graph i.e. the iterates over all the Connections in the graph checking each one's destination attribute, and then iterates through each matching connections relationship history for a matching relationship value. Each matching connection has a reference to their source attribute stored in a list that is the return value of the method. To ensure the test is suitably demanding, all entities were defined as having the same Relationships with each other - with the requested relationship value always stored as the oldest possible relationship state; this forces the method to check every relationship in every connection for every entity that is using the graph.

As to be expected, the computation time and frame time percentage increases with more entities and connections. However, even with ever increasing demands still remains at a manageable level for real-time interaction. The large frame time percentage for large combinations of entities and connections is of concern, however, in practice the number of connections per entity would rarely get as large as 50 unless the graph nodes were also used to track different types of items. Increasing numbers of Relationships do not appear to affect the performance of the traversal of the graph too severely with performance remaining below 5 percent.

Some steps may be taken to improve the performance of the graph. For example, if the data in the graph is not required immediately in the current frame it can be run in a separate thread (or in Unity in a Coroutine) and returning the result after processing is complete. Another step is to use multiple graphs for capturing different types of relationships, for example having one graph handle relationships to other characters and another to groups or items or locations. In a typical game, the number of characters involved in a request are not typically all the characters that exist in the game world but those in the immediate environment which reduces the length of the processing considerably.

Taking consideration that these results, while growing large in some scenarios, are worst case results arising when making a request of the most computationally demanding query offered by the graph; the RelationshipGraph can be said to meet the performance objectives of the project.

Table 5.3: Computation performance of the Relationship Graph using different quantities of Entities, Connections and Relationship.

Entities	*Connections*	*Relationships*	*Avg. Query Time*	*Avg. % of Frame Time*
5	5	5	0.042 ms	0.28%
10	5	5	0.078 ms	0.9%
50	5	5	0.422 ms	4.52%
100	5	5	1.266 ms	8.98%
50	1	5	0.256 ms	2.42%
50	5	5	0.41 ms	3.12%
50	10	5	0.602 ms	5.02%
50	20	5	1.008 ms	6.36%
50	50	5	2.094 ms	14.34%
10	10	1	0.08 ms	0.62%
10	10	5	0.116 ms	0.9%
10	10	10	0.138 ms	1.2%
10	10	50	0.352 ms	2.44%
10	10	100	0.668 ms	4.78%

5.4 Shortcomings & Challenges

The graph has a number of shortcomings in its implementation. These are all directly related to the manner in which the graph stores edges for each node. Internally the graph uses a Dictionary data structure to store a list of edges for each node - using the nodes as Keys, these edge lists are quickly retrievable. However, searching these lists for a matching value results in significantly poor performance as many of the query methods exposed by the Graph internally iterate over these list items at least once, and in some cases may iterate over edges of multiple nodes (as is the case with the method used in the processing performance tests in section 5.3.2).

To further affect the efficiency of the graph in searching for matching terms; the HistoryEdge stores previous Relationship values in a list as well. This leads to some methods with a complexity of $O(n^2)$. The approach taken in the implementation of

the model was to create a working prototype first and then attempt to find more efficient algorithms for the traversal of the graph, as well as determine more efficient data storage. However this was not successful.

The graph stores references to objects - this can lead to a graph with significant memory requirements if the objects are sufficiently large. This is due to the use of generics in the implementation of the graph. A better approach maybe to use objects that only implement an interface, and have the interface explicitly require the implementation of a more efficient value that can be stored in lieu of a whole object i.e. storing unique identifiers to nodes and relationships instead of whole objects. In fact early prototypes of the graph worked with Identifiers before moving to wholly generic objects.

At the beginning of the project, there was a challenge to explicitly define what a "Relationship" was with regards to the model and for the purposes of implementation. For some time Relationships were treated more like Opinions as can be found in the Fable games. Making the observation that Relationships are a state between two Entities was important to the design choice to separate the implementation of the Relationships from that of the Edge's. Once this distinction was made, the model was able to gel and handle even greater complexity with ease.

The current implementation of Relationships and Nodes relies on the IEquatable¡T¿ interface for efficient use of the Dictionary and List data structures offered by C# Collections. This was ultimately a poor decision as it interferes with extensions of the Nodes and Relationships that may want to implement that interface for their own comparisons. Having a custom method with similar functionality that can also be used as an Identifier would be better suited for the graph.

Chapter 6

Conclusion

The purpose of this chapter is to summarise the contributions made by the dissertation, present some potential future work and finally end with some closing remarks.

6.1 Contributions

Though future work can make numerous improvements, the following contributions can be identified from the work done so far.

Concerning the area of believability of social behaviour based on relationship among characters, the project presents a working prototype that was fully implemented both in Unity and as a standalone C# project for use in other environments. The Relationship Graph grants characters the capacity to make decisions based on their current and past relationships to one another. The prototype also demonstrates advanced concepts such as group behaviour, basic learning and adoption of new relationships, and messaging functionality that are built on the core components of the model.

The Guard demo as seeing Figure 6.1, demonstrates all the primary working features of the model working within Unity including: handling different message types, learning new messages, querying the graph about current and previous relationships, and determining relationships with other characters based on group associations. The Sphere demo demonstrates how the model can be used as a reference to quickly create simple gameplay prototypes by defining relationships among game characters. These two demonstrations offer operational proof of the relationship model functioning within

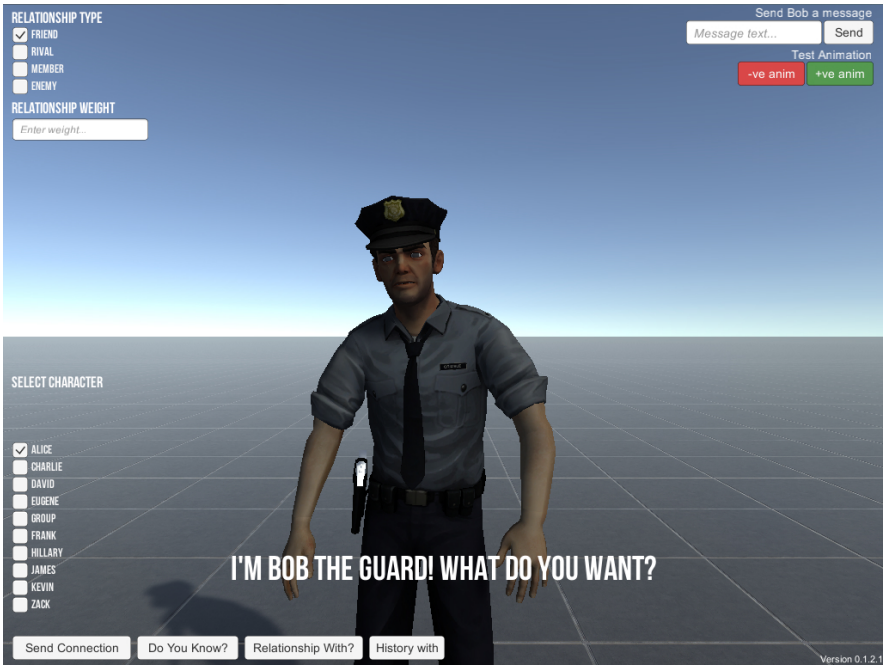

Figure 6.1: Screenshot of the Guard demo.

a game environment.

The main contribution of the model is in an efficient manner of storing knowledge of a characters relationships. Relationships to other characters were broken down into two primary types: direct and indirect types. Here a direct relationship represents an immediate relationship from one character to another. For example, a character "knows" that they are related to their brother directly. An indirect relationship represents knowledge of the relationship between two other characters. For example, a character "knows" that his son has a friend. However, that information may be imprecise, that relationship between the son and the friend may no longer exist as a direct relationship but continues to exist within a characters indirect relationships - in this case the character assumes that the son has a friend. Intentionally falsifying this information allows us to mislead characters. Although this aspect of the model was not adequately implemented, it remains a promising and interesting path for interactions with AI characters.

The improvements and practicality of the relationship model over other approaches is demonstrated in Section 5.1 where the relationship model is compared favourably to

the *Comme il Faut* system used in the game *Prom Week*.

6.2 Future Work

The following are some possible areas for the extension and improvement of the model.

6.2.1 Improved Learning System

The learning system implemented in the project is extremely simple. As described Section 4.4.2, a better approach would involve evaluating the Relationship with the source of the information to determine if it is trustworthy, or incorporating personality traits that would define a character as "gullible" and there more willing to accept new information.

6.2.2 Misleading Characters

While possible, this functionality was not investigated more. The possibility to provide an agent with false information is an activity not easily found in many modern games. While stealth games offer this as a mechanic (make noise at one area lead the NPC there while sneaking around another way), this is typically a scripted behaviour and more complex "lies" such as tricking a rival or strong opponent into a compromising situation is not. For example, if a player must defeat an opponent to claim a guarded prize, the player could avoid a physical confrontation by convincing their opponent that their brother has died and they must go attend the funeral while graciously offering to protect the prize until they return. Such interactions would be possible by using the graph as foundation for these relationships.

6.2.3 Optimizations

As described in Section 5.4, the graph was not implemented optimally. Many of the methods require long iterations of almost the entire graph, and this increases in complexity with the use of edges that store a history of Relationships. Completely replacing the list with another dictionary or hash data structure that offers quick lookups is recommended.

Additionally, replacing the extension of the IEquatable Interface's Equals method with a custom version would allow this interface to still be available for custom implementations or derivations of the components that would like to override the Equals and "==" operators.

To reduce the memory requirement, replacing generic objects with interfaces would allow for more efficient lookups to the Dictionary structure. This would mean that the internal data structure could work with an interface that implements an interface with a unique Identifier method and stores this Identifier as the key instead of a whole generic object.

6.2.4 Connection and Relationship Decay

Having Connections and Relationships that decay in value and are eventually removed at some threshold was an unmet desirable for the project. This is a feature that would allow for not just efficient memory use by removing old an unreferenced data, but also improve the believability of the model. Relationships evolve over time, for example, some friendships are lost due to lack of interaction with another person leading to a loss of familiarity.

6.2.5 Integration

The relationship model can work well as a reference for other game systems, such as the side quest generator by Sarah Noonan [38] or Tony Cullen's project on temporal factors [39]. Requesting the family members to generate a quest to save one of them, or determining a characters that would naturally group together at night or during a storm or for periodic ceremonies based on lunar cycles are all possible.

6.2.6 WideEdge

The project implemented the concept of a *HistoryEdge* which maintained a history of changes to the Relationship between two entities. This can also be observed to be a "DeepEdge" in that Relationship values are stored one after the other in a list. It may also be possible to turn this on its side and store multiple Relationship values

at once resulting in a "WideEdge" and use a function to combine them all into one representative value, or just leave them directly accessible.

6.3 Final Thoughts

The relationship model proposed in this dissertation has potential for use in a wide range of games. It can be used not just as a reference structure for game systems, but also as a high level definition for how game characters interact with one another.

Advanced functionality such as learning systems, group behaviour and broadcasting messages were demonstrated to be possible by combining the basic components of the model together into a whole greater than its constituent parts. The underlying graph model has further capabilities that naturally emerge, such as characters that are presumptuous or may carry incomplete or inconsistent knowledge.

While further work is required to improve its efficiency and implementation, the model was shown to be capable of creating believable character behaviour based on relationships between characters and of knowledge of other characters relationships.

Appendix

1. **Reference Materials**: A disc with all the project files (e.g. Unity project, RelationshipGraph project, models, etc.) used in the creation of this project is attached to the back of this dissertation.

2. **Source code**: The source code for the RelationshipGraph is also available on Github: https://github.com/pandaboy/RelationshipGraph

Bibliography

[1] Stangor, Charles *"Defining Social Psychology: History and Principles"*, Principles of Social Psychology, v. 1.0, http://catalog.flatworldknowledge.com/bookhub/2105?e=stangorsocial_1.0-ch01_s01, retrieved 10th May 2015

[2] Durlauf, Steven N. and Young, H. Peyton *"Social Dynamics"*, The New Social Economics

[3] Durlauf, Steven N. and Young, H. Peyton *"Social Dynamics"*, The New Social Economics, p3-5

[4] Bourse, Yoann *"Artificial Intelligence in The Sims series"*, http://www.yoannbourse.com/ressources/docs/ens/sims-rapport.pdf, retrieved 13th May 2015

[5] Damian, Isla *"GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI"*, http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php, retrieved 13th May 2015

[6] de Plater, Michael *"Postmortem: Monolith Productions' Middle-earth: Shadow of Mordor"*, http://www.gamasutra.com/view/news/234421/Postmortem_Monolith_Productions_Middleearth_Shadow_of_Mordor.php, retrieved 13th May 2015

[7] Graft, Kris *"Designing Shadow of Mordor's Nemesis system"*, http://www.gamasutra.com/view/news/235777/Designing_Shadow_of_Mordors_Nemesis_system.php, retrieved 13th May 2015

[8] Alexander, Leigh *"Road to the IGF: 11Bit studios' This War of Mine"*, http://www.gamasutra.com/view/news/236554/Road_to_the_IGF_11Bit_studios_This_War_of_Mine.php, retrieved 15th May 2015

[9] Josh McCoy, Mike Treanor, Ben Samuel, Aaron A. Reed, Michael Mateas, Noah Wardrip-Fruin, *"Prom Week: Designing past the game/story dilemma"*, University of California Santa Cruz, Expressive Intelligence Studio

[10] Simpson, Chris *"Behavior trees for AI: How they work"*, http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php, retrieved 15th May 2015

[11] Kuroiwa, Jaimie *"Unscripting the Scripted Event"* http://www.gamasutra.com/blogs/JaimeKuroiwa/20090515/83949/Unscripting_The_Scripted_Event.php retrieved 15th May 2015

[12] Russell, S and Norvig P. *"Artificial Intelligence A Modern Approach"*, What is AI?,

[13] Buckland, Mat *" Programming Game AI By Example"*, Introduction

[14] Buckland, Mat *"Programming Game AI By Example"*, State-Driven Agent Design

[15] Buckland, Mat *"Programming Game AI By Example"*, How to Create Autonomously Moving Game Agents

[16] Champandard, Alex J. *"On Finite State Machines and Reusability"*, AiGameDev.com http://aigamedev.com/open/article/fsm-reusable/, retrieved 13th May 2015

[17] Storr A. *"Solitude: A Return to the Self"*,

[18] Forsyth, Donelson R. *"Introduction to Group Dynamics"*, Group Dynamics

[19] Forsyth, Donelson R. and Elliott, Timothy R. *"Group Dynamics and Psychological Well-Being: The Impact of Groups on Adjustment and Dysfunction"*, Group Dynamics

[20] Russell, A. *"Opinion Systems"*, Ai Programming Wisdom 3

[21] Mateas M., Stern A. *"Procedural Authorship: A Case-Study Of the Interactive Drama Facade"*,

[22] Mateas M., Stern A. *"Facade: An Experiment in Building a Fully-Realized Interactive Drama"*,

[23] Pescosolido, Bernice A., Bryant, Clifton D. and Peck, Dennis L. *"The Sociology of Social Networks"*, 21st Century Sociology

[24] Pescosolido, Bernice A., Bryant, Clifton D. and Peck, Dennis L. *"The Sociology of Social Networks"*, 21st Century Sociology, 208

[25] Pescosolido, Bernice A., Bryant, Clifton D. and Peck, Dennis L. *"The Sociology of Social Networks"*, 21st Century Sociology, 210-212

[26] Wasserman S, Faust K. *"Social Network Analysis in the Social and Behavioral Sciences"*, Social Network Analysis: Methods and Applications

[27] Scott J, *"Relations and Attributes"*, Social Network Analysis, 3

[28] Champandard, Alex J. *"The Gist of Hierarchical FSM"*, AiGameDev.com `http://aigamedev.com/open/article/hfsm-gist/`, retrieved 13th May 2015

[29] Champandard, Alex J. *"Understanding Behavior Trees"*, AiGameDev.com `http://aigamedev.com/open/article/bt-overview/`, retrieved 13th May 2015

[30] Champandard, Alex J. *"Planning in Games: An Overview and Lessons Learned"* AiGameDev.com `http://aigamedev.com/open/review/planning-in-games/`, retrieved 13th May 2015

[31] Nau, Dana S. *"Current Trends in Automated Planning"* American Association for Artificial Intelligence. `http://www.cs.umd.edu/~nau/papers/nau2007current.pdf`, retrieved 14th May 2015

[32] Nau, Dana S., Cao, Y., Lotem A., and Munoz-Avila H. *"SHOP: Simple Hierarchical Ordered Planner"* http://www.cs.umd.edu/~nau/papers/nau1999shop.pdf, retrieved 15th May 2015

[33] Vansteenkiste M., Ryan M. R. *"On Psychological Growth and Vulnerability: Basic Psychological Need Satisfaction and Need Frustration as a Unifying Principle"*, http://www.selfdeterminationtheory.org/wp-content/uploads/2014/07/2013_VansteenkisteRyan_JOPI2.pdf, retrieved 15th May 2015

[34] *"Description of the SHOP Project"* http://www.cs.umd.edu/projects/shop/description.html, retrieved 15th May 2015

[35] *"Unity3D"* http://unity3d.com/, retrieved 22nd August 2015

[36] *"THE BEST DEVELOPMENT PLATFORM FOR CREATING GAMES"* http://unity3d.com/unity, retrieved 22nd August 2015

[37] Tiarnan McNulty *"Residual Memory for Background Characters in Complex Environments"*, University of Dublin, Trinity College, September 2014

[38] Sarah Noonan *"Side Quest Generation using Interactive Storytelling for Open World Role Playing Games"*, University of Dublin, Trinity College, August 2014

[39] Tony Cullen *"Modelling Environmental and Temporal Factors on Background Characters in Open World Games"*, University of Dublin, Trinity College, August 2015

[40] *"Creating and Using Scripts"* http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html, retrieved 22nd August 2015

[41] *"Collections (C# and Visual Basic)"* https://msdn.microsoft.com/en-us/library/ybcx56wz.aspx, Microsoft Developer Network, retrieved 22nd August 2015

[42] *"Basic Gesture Motion"* https://www.assetstore.unity3d.com/en/#!/content/25852, Unity Asset Store

[43] *"IDictionary¡TKey, TValue¿ Interface"* https://msdn.microsoft.com/en-us/library/s4ys34ea.aspx, Microsoft Developer Network, retrieved 23rd August 2015

[44] *"IEquatable¡T¿ Interface"* https://msdn.microsoft.com/en-us/library/ms131187(v=vs.110).aspx, Microsoft Developer Network, retrieved 30th August 2015

[45] *"Profiler"* http://docs.unity3d.com/Manual/Profiler.html, retrieved 29th August 2015