Eastern Michigan University
DigitalCommons@EMU

Master's Theses and Doctoral Dissertations | Master's Theses, and Doctoral Dissertations, and Graduate Capstone Projects

2008

Adding simple simulations to video games to improve realism

Leonard Kinnaird-Heether

Follow this and additional works at: http://commons.emich.edu/theses

Part of the Computer Sciences Commons

Recommended Citation
Kinnaird-Heether, Leonard, "Adding simple simulations to video games to improve realism" (2008). *Master's Theses and Doctoral Dissertations*. 198.
http://commons.emich.edu/theses/198

This Open Access Thesis is brought to you for free and open access by the Master's Theses, and Doctoral Dissertations, and Graduate Capstone Projects at DigitalCommons@EMU. It has been accepted for inclusion in Master's Theses and Doctoral Dissertations by an authorized administrator of DigitalCommons@EMU. For more information, please contact lib-ir@emich.edu.

Adding Simple Simulations to Video Games to Improve Realism

by

Leonard Kinnaird-Heether

Thesis

Submitted to the Department of Computer Science

Eastern Michigan University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

Thesis Committee:

Matthew Evett, Ph.D., Chair

Susan Haynes, Ph.D.

William Sverdlik, Ph. D.

July 22, 2008

Ypsilanti, Michigan

ABSTRACT

In this paper a method of enhancing the realism of certain computer game types, particularly First Person Shooter games, is explored by attempting to include intelligent neutral background characters to the game environment. This method also requires that the inclusion of these background characters will not adversely affect the performance of a game by drastically increasing the computational complexity of the game. A simulation was created to show how this can be done using a group of intelligent agents based in a simulated world, and a simplified system of norms designed to influence the agents' behavior. The agents are designed to interact with each other and other objects in their virtual world, in a fashion that could be considered human-like. To show the validity of this method, the simulation was designed to be minimally resource intensive. Furthermore, extensive tests were performed using a variety of initial starting values for various aspects of the simulation, to show that the simulation, and thus the method, can sustain itself long enough to provide an improvement to a gaming environment. The results of the tests are discussed at length, as well as theories as to how this method could be used successfully in computer gaming.

Keywords: Artificial Intelligence, Computer Gaming, Norms, Intelligent Agents

TABLE OF CONTENTS

ABSTRACT………………………………………………………………………….. ii

LIST OF CHARTS………………………………………………………………… v

CHAPTER

I. INTRODUCTION…………………………………………………... 1

II. REVIEW OF LITERATURE……………………………………….. 7

IPD for Emotional NPC Societies in Games…………………………... 7

A-Star Pathfinding for Beginners……………………………………… 10

The Sims: Under the Hood…………………………………………….. 13

III. DESIGN AND METHODOLOGY…………………………………… 17

Overview……………………………………………………………..17

Control Mechanism…………………………………………………..18

Resource Boxes……………………………………………………….19

Units……………………………………………………………………20

Norm System………………………………………………………... 22

Decision Making Process…………………………………………….23

Formal Definitions…………………………………………………...27

Design Summary……………………………………………………..36

Methodology………………………………………………………....36

IV. ANALYSIS OF RESULTS……………………………………………39

V. CONCLUSIONS AND RECOMMENDATIONS…………………….. 71

VI. REFERENCES………………………………………………………… 76

APPENDICES

A. PROGRAM CODE……………………………………………………78

LIST OF FIGURES

Figure 1: Screenshot of the Simulation During a Visualized Run…………………………38

Figure 2: Number of Turns, 0 Temperament, Violence Norm, Stealing Norm…………….40

Figure 3: Number of Births over Turns Elapsed, 0 Temperament, Violence Norm, Stealing Norm………………………………………………………………………………….. 40

Figure 4: Number of Murders over Turns Elapsed, 0 Temperament, Violence Norm, Stealing Norm………………………………………………………………………………….. 41

Figure 5: Final Norm Values over Turns Elapsed, 0 Temperament, Violence Norm, Stealing Norm………………………………………………………………………………….. 42

Figure 6: Number of Turns, -25 Temperament, 0 Violence, 0 Stealing…………………… 43

Figure 7: Number of Turns, -50 Temperament, 0 Violence, 0 Stealing…………………… 44

Figure 8: Number of Turns, 0 Temperament, 0 Violence, -50 Stealing…………………… 46

Figure 9: Number of Turns, 0 Temperament, 0 Violence, -25 Stealing…………………… 46

Figure 10: Number of Turns, 0 Temperament, 0 Violence, 25 Stealing…………………... 47

Figure 11: Number of Turns, 0 Temperament, 0 Violence, 50 Stealing…………………... 47

Figure 12: Number of Turns, -50 Temperament, 0 Violence, -50 Stealing………………...48

Figure 13: Number of Turns, -50 Temperament, 0 Violence, -25 Stealing………………...49

Figure 14: Number of Turns, -50 Temperament, 0 Violence, 25 Stealing………………… 49

Figure 15: Number of Turns, -50 Temperament, 0 Violence, 50 Stealing………………… 50

Figure 16: Number of Turns, -25 Temperament, 0 Violence, -50 Stealing……………….. 50

Figure 17: Number of Turns, -25 Temperament, 0 Violence, -25 Stealing……………….. 51

Figure 18: Number of Turns, -25 Temperament, 0 Violence, 25 Stealing………………… 51

Figure 19: Number of Turns, -25 Temperament, 0 Violence, 50 Stealing………………… 52

Figure 20: Number of Turns, 25 Temperament, 0 Violence, -50 Stealing………………… 53
Figure 21: Number of Turns, 25 Temperament, 0 Violence, -25 Stealing………………… 53
Figure 22: Number of Turns, 25 Temperament, 0 Violence, 25 Stealing…………………. 54
Figure 23: Number of Turns, 25 Temperament, 0 Violence, 50 Stealing…………………. 54
Figure 24: Number of Turns, 50 Temperament, 0 Violence, -50 Stealing………………… 55
Figure 25: Number of Turns, 50 Temperament, 0 Violence, -25 Stealing………………… 55
Figure 26: Number of Turns, 50 Temperament, 0 Violence, 25 Stealing…………………. 56
Figure 27: Number of Turns, 50 Temperament, 0 Violence, 50 Stealing…………………. 56
Figure 28: Number of Turns, 0 Temperament, -50 Violence, 0 Stealing………………….. 57
Figure 29: Number of Turns, 0 Temperament, -25 Violence, 0 Stealing………………….. 58
Figure 30: Number of Turns, 0 Temperament, 25 Violence, 0 Stealing…………………... 58
Figure 31: Number of Turns, 0 Temperament, 50 Violence, 0 Stealing…………………... 59
Figure 32: Number of Turns, -50 Temperament, -50 Violence, 0 Stealing………………...60
Figure 33: Number of Turns, -50 Temperament, -25 Violence, 0 Stealing………………...60
Figure 34: Number of Turns, -50 Temperament, 25 Violence, 0 Stealing………………… 61
Figure 35: Number of Turns, -50 Temperament, 50 Violence, 0 Stealing………………… 61
Figure 36: Number of Turns, -25 Temperament, -50 Violence, 0 Stealing……………….. 62
Figure 37: Number of Turns, -25 Temperament, -25 Violence, 0 Stealing………………...62
Figure 38: Number of Turns, -25 Temperament, 25 Violence, 0 Stealing………………… 63
Figure 39: Number of Turns, -25 Temperament, 50 Violence, 0 Stealing………………… 63
Figure 40: Number of Turns, 25 Temperament, -50 Violence, 0 Stealing………………… 64
Figure 41: Number of Turns, 25 Temperament, -25 Violence, 0 Stealing………………… 65
Figure 42: Number of Turns, 25 Temperament, 25 Violence, 0 Stealing…………………. 65

Figure 43: Number of Turns, 25 Temperament, 50 Violence, 0 Stealing…………………. 66

Figure 44: Number of Turns, 50 Temperament, -50 Violence, 0 Stealing………………… 66

Figure 45: Number of Turns, 50 Temperament, -25 Violence, 0 Stealing………………… 67

Figure 46: Number of Turns, 50 Temperament, 25 Violence, 0 Stealing…………………. 67

Figure 47: Number of Turns, 50 Temperament, 50 Violence, 0 Stealing…………………. 68

Figure 48: Number of Turns, -50 Temperament, 50 Violence, 50 Stealing……………….. 69

Figure 49: Number of Turns, 50 Temperament, -50 Violence, -50 Stealing………………69

CHAPTER I

INTRODUCTION

If the goal of any simulation is to provide the most accurate portrayal of its particular setting as possible, then any change that improves this accuracy is a step in the right direction. When a simulation represents a fantastical environment, where the laws of nature that govern our real environment may be altered, the task of creating an accurate simulation is considerably easier. When the user of a simulation is not familiar with the constraints of the simulated environment, then any constraint, however fantastic, will be assumed to be normal for the simulation. However, when creating a simulation that represents the natural environment in which we dwell, the task is much harder. The user of a simulation that is grounded in reality is intimately acquainted with the rules that govern the natural world, if not through scientific knowledge, then at least through personal experience. Thus the user can accurately identify where the simulation is unable to correctly depict reality. When creating a simulation for a video game that is designed to represent the natural world, the inability to accurately depict reality can be a detriment. Video games are primarily used as a means of entertainment and are supposedly designed to induce a feeling of enjoyment. Thus, inaccuracy in a video game simulation will theoretically decrease the amount of enjoyment a user receives from the game. By this it can be inferred that the value of a simulation that attempts to recreate a real world situation is directly tied to the accuracy of the simulation.

It can be said that video game development is becoming an increasingly important discipline in the field of computer science. It is also true that simulations are an effective means of researching specific situations in the natural world. Because of this, it is not unrealistic to

expect that using a simulation designed to represent aspects of the natural world in a video game would serve to increase the realism of that video game. There is also the added benefit that the same simulation might also be used to advance research in other areas of the field. This paper reports findings that are directly involved with the development of better simulations for video games, but could indirectly be applied to other fields, such as swarm intelligence and agent-based systems, since the simulation is constructed in a similar fashion to these.

Many video games focus on the interaction of a human player with computer-controlled Non-Player Characters. Non-Player Characters, typically called "NPCs" or "bots," are characters within the game world that are controlled by an artificial intelligence (AI) rather than by a human player. These bots come in two general forms: those that are designed to interact with a player and those that are placed into the game to add realism and atmosphere but don't directly affect a player in any way. The bots that interact with a player fall into two categories: those that are designed to help a player and those that are designed to hinder a player.

An example of a bot that helps a player is the quest giver in a Role-Playing Game (RPG), such as *The Legend of Zelda: Twilight Princess* [13] or *World of Warcraft* [10]. Quest giving bots advanced the plotline of a game by instructing the player to complete a task, and then return to the quest giving bot to receive some sort of reward. Another example of a bot helping a player is the teammate bot that can be found in many new First Person Shooters (FPSs), such as *Brothers in Arms* [11] or Third Person Shooters (TPSs), like *Grand Theft Auto* [12]. The teammate bot will assist a player by fighting against an opponent in tandem with the player. In some games, teammate bots are also designed to respond to orders given by a human player, further adding to the realism.

The most common example of a bot that is designed to hinder a player is the opponent

bot. The opponent bot is the most basic kind of bot and can be found in video games since the first time an AI was used in place of a second human player. The opponent bot is multifaceted and can be altered to suit any game genre that requires an opponent. In some games, such as chess or checkers, there will be only one opponent bot, representing the computer player. In a game such as *Pac-Man* [9], there are four opponent bots, each represented by a ghost avatar that navigated the maze, attempting to kill the human player. In a modern game such as a FPS or RPG, there are multiple types of opponent bots, each with different configurations, depending on the game. From these types, a nearly limitless number of different instances of the opponent bot can be created.

Bots that are created for the purpose of adding atmosphere to a game could be classified as neutral bots. These bots do not help or hinder a player in any way; they exist only to improve the realism of a game. Neutral bots are found almost exclusively in RPGs. For example, in the game *World of Warcraft* [10], when the player enters a town setting, there are some quest giving bots, but there are great number of bots that simply move around the town, designed to follow a set of actions that a nondescript townsperson would follow. If a player attempts to interact with these bots, the neutral bot will say or do something that a nondescript townsperson would, but this action will not advance the plot for a player. All modern RPGs contain neutral bots in some capacity, because a game where all the bots were either quest givers, or opponents, would be very sparsely populated and thus unrealistic. Some RPGs have attempted to add another level to the neutral bot by changing the neutral bot/player interaction based on a player's previous actions. In the game *Fable* [14], a player may interact with the neutral bots in a town setting in either positive or negative ways. When the player interacts with the townspeople negatively enough times, the townsfolk will flee when the player comes near them. Conversely, if the

player interacts positively enough times, the townspeople will cheer whenever the player comes near them. *Fable* adds even more realism by allowing the player to interact with a specific neutral bot in an attempt to turn the neutral bot into a love interest. When this happens, the neutral bot becomes a quest-giving bot and will then act to advance the story line for the player. It could be said, however, that since only specific neutral bots in *Fable* can be turned into the player's love interest, these bots are not truly neutral and represent a hybrid of the two classes of bots. Some non-RPG games have also added neutral parties. In *Grand Theft Auto* [12], there are some bots that should be considered truly neutral as they cannot be converted to another type of bot, but there are also some bots that could be considered hybrid bots. *The Sims* [4], by far, has the most advanced neutral bot system. Some of the neutral bots such as those representing the postman, or various other characters that are not constantly present in the game world, are truly neutral and cannot be influenced by the player's actions. All of the other neutral bots are hybrids and are more complicated than any other hybrid bot. Not only can they be turned into bots that affect the player positively, these bots can also be turned into bots that affect the player negatively. A more in-depth analysis of *The Sims* can be found later in the paper.

Even though there have been great strides towards including neutral bots in the RPG genre, rarely will you see bots that are designed to have a neutral opinion of the human player in FPS games. This is especially true for FPS games that are found in a war setting. The most common approach is to concentrate the bulk of the AI on creating opponent bots that react to and attack the player in a realistic fashion. As previously mentioned, there has also been some focus on adding AI-controlled friendly bots. In modern warfare, especially when it takes place in an urban setting, it is rarely the case that all the people at the site of a particular battle are aligned with any of the opposing forces that initiated the battle. Usually in an urban area there will be an

innocent bystander contingent that is ambivalent toward any of the oppositional forces fighting around them. These neutral parties are simply trying to continue their life in as normal a way as possible considering their surroundings. Developers have, to date, largely ignored this part of the actual experience. *Grand Theft Auto* [12] includes a neutral party, but it does not spend nearly as much time concentrating on the opponent AI as games that omit neutral parties. The reasons behind the omission of the neutral party are numerous, but the main problem, in my opinion, is one of resources. Most developers believe that to create an AI that can behave in a humanlike fashion requires a great deal of computer hardware resources. Developers may believe that including neutral parties would not be cost effective in the long run and decide to focus their efforts on making the opponent AI as realistic as possible.

In order to fulfill the true intent of all games that simulate a lifelike situation, some effort must be made to include a neutral party. However, it must be an effort that is made so that the advances made in controlling the enemy AI are not counteracted by a lack of resources. Thus, the problem that this study is trying to solve is: Can a simple simulation, representing a society, be created in a fashion such that this same simulation could be used in a video game setting? Furthermore, could this same simulation be created so that it does not have an adverse affect on the performance of a video game, but so that it improves the overall realism of that game? To show how this can be done, this study will demonstrate a relatively simple artificial society that acts in a limited humanlike fashion but doesn't increase the computational complexity of the simulation dramatically. In this study, a simulation will be created that contains a small number of bots that are not designed to do anything except act as they were going through a simple daily routine. This routine will incorporate simple human actions such as eating and drinking, as well as some more complex social interactions such as giving/stealing, violence, and mating. This

study will also impose a simple norm system that will affect the bots' behavior based on what the current norm system settings are. A norm is defined as "a rule that is socially enforced," [15] and a norm system is a collection of norms for a society. The definition of a norm can be simplified in that the word "norm" is short for normal behavior and represents whatever a society considers to be normal behavior. By using a norm system, the possibility of whether one can show, simply, how humans would react by being placed in situations that are contrary to their ideas of how to act in the world, will be investigated. The case of good people being put into a world where violence is the norm, and what their reactions are, will be of particular interest in this study. The study should be considered successful if it can be shown that a simple AI can be constructed that will look somewhat humanlike and can be shown to run in a resource- and time-efficient way. From this the study can infer that a neutral party can be included into conflict-simulating video games without reducing their playability or performance. In adding this neutral party, the hope is to improve the realism of lifelike simulations in all game genres. Chapter II contains a review of the main works referenced for this paper. Chapter III is a review of the methodologies that went into setting up the experiment that is referenced in this paper. Chapter IV contains a summary of the results obtained for the experiment. Chapter V is a summary of the entire work.

CHAPTER II

REVIEW OF LITERATURE

In order to create the simulation for this thesis, it was important to find other similar studies in the literature to base the simulation on. *IPD for Emotional NPC Societies in Games*, by Chaplin and El-Rhalibi [1], centers on the creation of a simulation containing a group of NPCs or agents that interact with each other and their environment. The agents use simulated emotional behavior to improve realism. The authors believe that by using emotions in choosing agent behavior, the user of the simulation will receive a more rewarding experience. The simulation uses a rule-based system to model the emotions, drives, and agent relationships. A norm system, managed by an implementation of the Iterated Prisoner's Dilemma, is used to determine the interactions between agents. The norm system is very rudimentary and takes effect only when two agents are interacting with each other. These interactions are characterized by honest and deceitful behavior. Agents acting in an honest way are considered to be acting in line with the norm, whereas agents acting deceitfully are considered to be acting contrary to the norm.

It was also important in the research for this thesis to study other instances of the use of norms. The Prisoner's Dilemma refers to a study done in the 1950s in which people were posed a hypothetical situation and asked to make a decision. Chaplin and El-Rhalibi relied heavily on a paper titled *An Evolutionary Approach to Norms*, by Robert Axelrod [2], for their information on this topic. This study provided a foundation for a new track in the field of game theory. In this situation two suspects, A and B, are arrested by the police. The police don't have enough evidence to convict either suspect with the facts at hand. However, the testimony of one of the

suspects against the other would be sufficient to convict the other. The prisoners are separated and given the same option of whether to stay silent or to betray the other prisoner. They are also told that if they both stay silent they will both be sentenced to six months in jail, that if one betrays the other and the other stays silent then the betrayer will be freed while the other will sentenced to ten years, and that if they both betray each other they both receive five years. The dilemma posed to each prisoner is what course they should take to receive the smallest prison sentence, without knowing what decision the other prisoner has made. When applying the classic Prisoner's Dilemma to game theory, it becomes a type of game where the player concentrates on maximizing his or her own payoff. However, this doesn't work well in video games because any rational player will choose to betray another player because it maximizes their gain. The resulting time from betraying is either zero or five years, whereas the resulting time for not betraying is either six months or ten years. Since the outcome of the other player is unknown, the obvious choice is to choose whatever results in the least amount of time. In order to make this idea useful in game theory one can use the Iterated Prisoner's Dilemma (IPD) [1]. The main difference is that in the original, once the prisoner made their choice, the game ended. In the IPD, the game continues after the choice has been made, and there is an option for the offending player to be punished for betraying. This alters the game in that the gain for betraying is not as high because of the possibility of punishment, thus making staying silent a more palatable option. IPD can be expanded into a larger sense by replacing betrayal with one or more different rules. This can be interpreted as a set of norms, in that players can be punished for not following the norms. Thus players will be inclined to maximize their own gain by acting in line with whatever the norms are set to be.

Chaplin and El-Rhalibi define their emotional model of behavior as actions consisting of physiological reactions, cognitive states, and expressive behaviors. They have devised this model using multiple resources from the field of behavioral psychology. The cycle of emotional response in their system starts with an external input being applied to an agent. The agent then has an internal cognitive reaction to the stimulus, and performs a corresponding external action. The set of emotions consists of fear, anger, sadness, and happiness.

Beyond the norms and the emotional model of behavior, the authors have also included a set of drives. These drives are social need, energy need, need for rest, and need for heat. Their agents have relationships with other agents, which can be either bad or good relationships. These relationships compose the social drive. The other drives are physical drives and are representative of variables corresponding to the three remaining drives.

Chaplin and El-Rhalibi use the Iterated Prisoner's Dilemma to help shape the emotional response of their agents. These agents use a matrix of drives and emotions to generate their responses. Their norm system is based on IPD. If an agent wishes to break a norm and do something that is not accepted, it is influenced by two factors that are a part of all agents. First, they have an inherent boldness value that determines whether the agent is bold enough to perform an action despite the implied consequences. There is also a vengefulness value that determines how intent other agents are on punishing an offending agent that is used in the action calculation. Both of these values are affected by the results of breaking norms. Succeeding in breaking a norm without being punished increases boldness. Being punished for breaking a norm will decrease boldness and increase vengefulness. A decision by an agent not to punish another agent acting contrary to the norm convention leads to the creation of meta-norms. With these meta-norms, there is a chance that an agent will be punished for not punishing another

agent. This creates an extra level of complexity in their norm system. The authors also introduce a concept of noise to the decision-making process. The noise they introduce basically effects a small enough change to random individuals that is just enough to change their decision making process from one conclusion to another. The noise is based on an individual's knowledge of the whole system and the past actions of the other agents in the simulation. If one individual recognizes that another has a high level of boldness, then this knowledge will sometimes be applied to the first agent's interactions with the other. Roughly this means that if one agent shows a proclivity toward behavior that is contrary to the norm system, then other agents may be inclined to keep an eye on the offending agent, in anticipation of more anti-normal behavior.

Chaplin and El-Rhalibi's paper was the central motivation for this thesis. It is evident that work has been done using norms for simulations and gaming but that these norms were always being explored to a depth that made them unsuitable for some applications. The concept of "drives" is a particularly important aspect of this paper. Using drives makes agents more humanlike in that their actions will be derived from their surroundings, rather than following some sort of fixed behavior; agents will make decisions based on whatever drive is activated because of some need. Combining drives with norms makes the simulation as a whole more like a functioning humanlike society. This is because some drives will cause agents to act in ways that may or may not be contrary to the norm system. When this happens, their behavior will be doubly affected: once by their own needs and again by the norm system determining what actions they can take to fulfill their needs.

In order to create a realistic simulation, it was important to find a way to automate the navigation of the agents in the simulation. In this light, the information contained in *A-Star*

Pathfinding for Beginners [3] was integral in the design of the simulation. This is a very complete webpage detailing exactly how the A-Star algorithm works. The A-Star algorithm [3,17] is often abbreviated "A*." It is a best-first, tree-searching algorithm similar to Dijkstra's algorithm and is designed to find a path of least cost between a starting node and any one goal node. It calculates a path by evaluating nodes using a distance function. The distance function, $f(x)$, is given as $f(x) = g(x)+h(x)$, where $g(x)$ is the distance already traveled and $h(x)$ is a heuristic evaluation of the distance left to travel. The process starts at any given node. The node is "expanded" by taking all of the immediate descendents of the given node and adding them to a priority queue called the "open list." Then the algorithm calculates the $f(x)$ value for every node in the open list and chooses the node that has the lowest value. It places this node in a list called the "closed list," expands the new node, and adds its descendants to the open list, and the process repeats. During the expansion process, nodes that are already on the closed list or nodes that have been defined as impassible are ignored, and the algorithm keeps track of the parent node for any node that has been added to the open list. The algorithm ends when there are no nodes left in the open list, or when a goal node has been added to the closed list. The path can now be found working backwards from the goal node and using the recorded parent of each node until the initial node is reached. The algorithm is very straightforward, and it works very well in game-like situations provided some key conditions are met. Many FPS games use A* pathfinding for opponent bots when they attempt to find the shortest path to the player, or the shortest path to a line of fire to attack the player, or pretty much whenever the bot needs to traverse the game-playing area from one point to another.

The heuristic, represented in the equation by $h(x)$, is the most important part of the equation, because if it doesn't correctly estimate the distance left to a goal node, the A-Star

algorithm will not be able to find the shortest path to a goal node. A-Star works best when movement costs are discrete rather than continuous, such as with a grid. In this case, one can treat each node as a square on the grid, and when that node is expanded, nodes that border that node are added to the open list. Using this scenario, the importance of the heuristic in the algorithm can be shown. The heuristic is basically just a question of movement in this case. A very simple heuristic is known as the Manhattan Distance heuristic. When using this heuristic, the path can only be found using the nodes directly north and south, and directly to the east and west of the current node. Diagonal nodes are not counted in this heuristic, which was probably named for the way people must traverse through the large blocks of buildings in New York City. Despite the fact that this heuristic seems very simple and good for find the shortest path, it is not. If the agent can only move north, south, east, and west, then The Manhattan Distance heuristic is "inadmissible." However, in a more realistic situation where the agent can also move diagonally, the Manhattan Distance heuristic becomes what is known as an "admissible" heuristic. The term "inadmissible" is used to describe a heuristic that overestimates the remaining distance to the goal node. If the heuristic is inadmissible, then it means that A-Star might not always find the shortest path[1]. If including the nodes to the top-left, top-right, bottom-left, and bottom-right of the current node in the Manhattan Distance heuristic is considered, then this new heuristic, called the Manhattan Distance plus Diagonals heuristic, is now admissible. The reasoning behind this can be seen by analyzing the grid representation. A distance that is calculated using the diagonals would be similar to calculating a distance "as the crow flies," which is obviously going to be better than calculating distance in the block-wise fashion initially discussed. In the case where movement costs are fixed between the grid cells and the agent has the capability of moving north, south, east, west, and diagonally, then the Manhattan Distance

[1] More data on the definition of admissibility can be found in any Artificial Intelligence textbook.

plus Diagonals heuristic assures that the shortest path between two nodes will be found by the A-Star algorithm.

This page also provides some insight for making A-Star work efficiently. Because A-Star can require some serious time and space requirements, especially with large game boards, many people have made improvements in the actual implementation of A-Star in practice. The most important improvement that was used in this work was the use of binary heaps to store the open list. Because of the way a binary heap is constructed, the lowest f value node in the open list can be made to stay at the top of the heap, thus making costly searches for this node unnecessary. The author suggests that in practice, using a binary heap to store the open list will result in a two- or three-fold speed increase, and this increase can be even greater for longer paths.

The Sims: Under the Hood [4] is a presentation slideshow from a lecture given at Northwestern University by Ken Forbus, with help from Maxis Software founder, Will Wright. This presentation is concisely describes of how the game *The Sims* works.

Everything in *The Sims* is an object, whether it is things like desks and lamps, non-player characters (NPCs), or player characters. All the objects have sound effects, graphics, and animations that correspond to them. They also have routines that are associated with actions performed on or by the object. In addition to the previous information, the character objects also contain the traits that make up the character. These traits include needs, personality, skills, and relationships. The needs are broken into two categories, physical and mental. The physical needs are hunger, comfort, hygiene, and bladder. The mental needs are energy, fun, social, and room. These needs combine to form a character's happiness level. If a character's needs are not met, then his or her happiness level will be lower. The levels of happiness simulate emotion in a

character.

The characters also have personalities. These personalities define what kind of person the character is in general terms. They are arranged in five pairs, with each pair consisting of two opposite types of personalities. They are sloppy/neat, shy/outgoing, serious/playful, lazy/active, and mean/nice. These personality traits form a kind of dichotomy in the system, such that in an ideal situation, both sides of the dichotomy would be equally represented and thus create a balance between the two. Each character contains one type from of each of these pairs of traits. So a character could be sloppy, shy, serious, lazy, and mean, but not sloppy, neat, shy, playful, and nice. These personality traits are applied to what the author defines as a Motive Engine, which affects what needs are more important than others. For instance, a character who is neat will have a higher need for the physical need hygiene. Each character also has skills that are used for the "Employment" area of the game. In addition to these things, a character has relationships with other characters in the game. These relationships determine what kinds of interactions between characters are available. They also determine how characters' emotions change, based on interactions that they perform or are party to.

In *The Sims* the goal of Non-Player Characters (NPCs) is to maintain a high level of happiness. All of the actions performed by an NPC are designed to fulfill needs, and thus raise their level of happiness. The types of interactions are broken down into two types: social interactions and object interactions. The social interactions are especially important, because they add a great deal to the realism of the game. There are many types of social interactions, and the actions that can be performed between two characters are determined by the relationship value that the characters share. These social interactions can either fail or succeed when performed. The chance of success is based upon a calculation made concerning the current

needs and relationship values of the two characters involved. A successful social interaction will increase the corresponding relationship values and the social need values for each character, whereas an unsuccessful one will lower them. The object interactions are defined by three classifications: normal, pushed, and functional. The normal interactions make up 90% of the interactions and involve actions that don't directly complete any goal in the game world, and aren't directly driven by any need. An example of this would be walking from one part of the game world to another. The functional interactions, however, are the exact opposite. These are goal-oriented actions driven by a need. The functional interactions include things like cleaning, repairing, and cooking. The third type of interactions, pushed interactions, are interactions that are given a higher priority than others. Typically these interactions correspond to emergencies in the game, such as the need to flee from, or put out, a fire. Another example is the need to arrive at work on time, part of the game's "Employment" phase. The pushed interactions take precedence over a character's other interactions and prevent him or her from performing other interactions, unless directed to do so by the user.

The rest of the presentation talks about invisible objects, or objects that aren't directly defined in the game, such as plumbing and phone lines. More importantly, the presentation briefly shows how the Sims uses the A-Star algorithm for pathfinding. The pathfinding is broken into two parts, room-to-room and within-room. The room-to-room pathfinding still uses A-Star, but the goal states are different than with the within-room model. The room-to-room method is more concerned about efficiently navigating through doorways and corridors, whereas the within-room method is more concerned with reaching the actual goal.

The simulation designed for the paper borrows a great deal from *The Sims*. The use of a dichotomy of personality traits and the use of social interactions are the most obvious examples

of this. The simulation also borrows the use of A* pathfinding from *The Sims*, and the concept of using an object as a source of food and water, rather than having resources scattered over the terrain.

CHAPTER III

DESIGN AND METHODOLOGY

Overview

The simulation was designed to test whether or not a group of individuals could perform within a certain set of parameters defined by the norm structure given to the group. The norm structure is defined as the group of norms that are applied to the simulation. It is designed to be similar to a First Person Shooter (FPS) style video game, in that it contains a game map that bots traverse. The video game model is greatly simplified, however, so that the focus can be directed toward the AI and not the video game itself. Simplifying the video game model was part of a larger effort designed to make sure that the simulation was small and lightweight and that the computations it was performing were relatively simple. Much of the simulation was designed similarly to the simulation in the paper by Chaplin, but the Iterated Prisoner's Dilemma concept was omitted. While the Prisoner's Dilemma is very accurate in depicting the development and enforcement of human-like norm systems, to depict it accurately requires resource allocation that is counter-productive to this study. Therefore it was decided that a group could be maintained just by having a structure of norms for them to follow, and this method was included in the simulation. By using this method, as opposed to the Prisoner's Dilemma method, the entities do not perform all actions based on a "least cost to them" style of reasoning. Instead, they only adhere to that way of thinking when they need something. If they have a need for food or water, then they behave like an animal and attempt to obtain the desired resource, by any means necessary. If they have no resource needs, then they behave like a human and attempt to interact with the units that share their environment. In this they explore the dichotomy of norms that has

been imposed on them. To attain a dichotomy, and thus realism, equal amounts of good actions and bad actions are possible in the simulation. The theory was to design a system where neither good nor bad would dominate the world without direct human intervention. The main reasoning behind these design choices is that the experiment is attempting to show that a simple system could work, and being that it was lightweight in terms of size and computational overhead, it could easily be ported into a video game with complex graphics and AI without any serious reduction in playability.

The simulation contains a square board that represents the world that the game is taking place in. The board is divided up into grid squares, or nodes, which can contain a game entity or nothing. In this world two types of entities reside: the AI controlled units and the resource boxes. Both of these will be explained in depth shortly. The simulation is turn based, and the speed can be controlled by the user, up to the point where the simulation is running as fast as the computer it is on can handle. Running in the background, and not represented by an avatar of any kind, is a control mechanism that monitors and controls everything that happens in the simulation. This control mechanism is by far the most complex part of the simulation.

Control Mechanism

The most important part of the simulation is the control mechanism. It is by far the most complex part of the whole program. The control mechanism controls actions that take place in the simulation world. The control mechanism consists of two multidimensional arrays keeping track of unit and resource box positions, a pathfinding function, and the norm system that was developed to guide the individual unit’s behavior. The mechanism provides information that is

used by the units in their decision-making process. In addition, the control mechanism updates the game world based on the actions of the units during a given turn.

The pathfinding system uses the A-Star algorithm that employs the Manhattan Distance method with Diagonals to form paths. The A-Star algorithm, as previously discussed, is a search algorithm that can be used to find the shortest distance between two points. The simulation uses location arrays, and any unoccupied node, or grid position, has a traversing cost of one. If that node contains either a unit or part of a resource box, then it was given a cost of 1000. This prevents paths from traveling into occupied nodes. The simulation uses the Manhattan Distance heuristic with Diagonals, because it was the best heuristic that was admissible for the landscape of the simulation. It is admissible because it never overestimates the remaining distance to the goal node, and thus will always find the shortest path to a goal node if one exists.

Resource Boxes

The resource boxes are an important part of the simulation, and they add to the realism of the system. There are two resource boxes in every instance of the simulation. One contains food, and the other contains water. The user may set the size, in terms of area, of the boxes, and the initial starting amount of each resource. The boxes are placed randomly around the simulation board during initialization. When a unit desires a resource, the first place they look is to a resource box. The units can obtain a resource by moving to any of the grid squares that border the resource box. The resource boxes are not an infinite source of a resource and will run out after a certain amount of time. To counter this, the user can have the resource box refilled with a set number of resource units after a certain number of turns have elapsed with the box being empty. This is to simulate times of plenty and times of famine in the simulation. By

setting the simulation to replenish the resource boxes infrequently, the user can simulate a time of famine to record the unit actions that will occur because of it.

Units

The AI controlled units are also relatively complex and are equipped to make their own decisions. Each unit contains several pieces of information that help it make decisions. These include the unit's age, temperament, hunger, thirst, health, and an array containing data on the unit's relationship with other units. The unit also contains some variables to keep track of its position, its current velocity, and how much food or water it is carrying. Each unit can carry with them up to three units of each resource. There are also variables to store what action the unit is currently attempting to perform and what target it is currently trying to interact with.

Each unit can perform a set of actions. These include idling, eating, drinking, obtaining food, obtaining water, moving randomly, and interacting with other units. When obtaining food or water, the units can get the resource from a resource box, request the resource from another unit, or steal the resource from another unit. When interacting with other units the actions are talking, hugging, slapping, kissing, punching, mating, and killing.

The age variable determines how old the unit is, in terms of simulation turns, and is used when calculating when the unit dies. After the unit reaches a certain age, it is considered dead.

The thirst and hunger variables determine what, if any, needs the unit has for water or food at the current time. These variables start at zero and are incremented by one every turn. Not having access to food or water for long periods of time will affect the unit's health. Every turn the unit's health and thirst variables are incremented by one.

If the unit's hunger value rises above 99, its health value is decremented by one every turn that the hunger value is over 99. The thirst value works similarly, except that the health value is decremented by two every turn. This, theoretically, is more realistic since humans can last longer without food than they can without water [8]. The health variable determines the amount of health the unit has. This variable ranges from zero to 100, and if the unit's health reaches zero, then the unit dies. Health can be affected by a variety of factors. As previously mentioned, not eating or drinking for long periods of time will reduce the unit's health. Being attacked, either punched or slapped, by another unit, will cause the health to decrease by either ten or twenty, respectively. The health variable can be increased by being hugged or kissed by another unit, which will increase the health value by either ten or twenty, respectively. In early designs of the simulation, the health value was incremented every turn to simulate healing, but this was omitted from the final design because it wasn't easy to accurately, and quickly, predict the many variables that compose the human body's ability to regenerate itself. This also keeps the dichotomy of good and evil in the simulation intact without having to design a way that the entities would suddenly start having their health variable decreased every turn.

The temperament variable determines, along with the norm system and the unit's relationships with other units, what actions a unit might take during a given turn. The relationship array tracks how the unit feels about other units with values from -50 to 50. Doing bad things to a unit will decrease that unit's relationship status with the offending unit. Conversely, doing good things will increase the relationship status.

Norm System

The norm system developed for the simulation was created with simplicity in mind. Rather than having rules for every action, actions were grouped into two classes, violence and theft. Each norm value can range between -50 and 50. They default to an initial setting of zero but can be set by the user at program startup. Actions performed during a turn in the simulation will be applied to the current norm values, and then those updated values will be used to determine the actions for the next turn. For every slap, punch, or killing that occurs, the violence norm will decrement by increasing values. For every hug, kiss, and successful mating that occurs, the violence norm will increment by increasing values. For every theft, the theft norm will decrement, and conversely for every donation it will increment.

When deciding on what to do during a current turn when the unit has nothing to do, the unit chooses among six values. If the unit's hunger value is above 49 and the thirst value isn't, then the unit will attempt to eat if it has food or obtain food if it doesn't. If the thirst value is above 49 and the hunger value isn't, then the unit will attempt to drink if it has water or obtain water if it doesn't. If both values are above 49, then the unit will attempt to eat or drink if it has either resource. If not it will try to obtain the resource corresponding to whatever is higher, hunger or thirst. If the unit has resource needs, but that resource is absent in the game world, the unit reverts to the idle state. If the unit has no current resource needs in a turn, then it chooses randomly between idling, eating, drinking, obtaining, moving randomly, or interacting with another unit. The application of the norm system can be seen as applying fuzzy logic to the system. Rather than just making decisions based on one extreme or another, the norm system provides a step-like appearance to the decision-making system.

Decision Making Process

The decision-making process is partly handled by the simulation control mechanism. As mentioned previously, the unit itself makes a decision of what kind of action to perform based on its current resource needs. If the unit requests a resource-obtaining action, the control mechanism goes through a short process to find all the information needed and return it to the unit. When asked to help the unit find a resource, the control mechanism first locates the closest source of food. If the requested resource is in the resource box, then the control mechanism looks there first. The mechanism first finds an empty grid space adjacent to the resource box. It takes those data and passes that to the pathfinding function. The pathfinding function finds the shortest path to the given point and returns the first step in the path. This first step can be applied as a discrete velocity, with values of -1, 0, and 1, which can be applied in terms of an X/Y coordinate system. The control mechanism then passes this velocity information back to the unit. When the control mechanism updates the positions of all the units in the simulation, it applies these values to the unit's current location to determine their new location. In order to prevent collisions between units, this process of finding the closest empty spot, finding a path, and returning the data to the unit is performed every turn until the unit reaches its destination and obtains the resource or the resource box becomes empty.

In the case that the requested resource is unavailable in the corresponding resource box, the control mechanism will find the closest unit that is carrying the requested resource and perform the same calculation as if the unit was a resource box. The major difference in this action involves the theft norm and whether the unit requests the resource or steals it from the other. In this case the control mechanism will return the location of the closest unit that possesses the desired resource. Using the norm system, the unit will then determine how to

approach the target unit when attempting to obtain the resource. If the sum of the unit's temperament, the current value of the theft norm, and the unit's opinion of the target unit is greater than or equal to zero, then the unit will choose to request the resource from the target. If the same sum is less than zero, then the unit will attempt to steal the resource from the target. The process now is identical to the process of getting a resource from the resource box except at the final moment. When the unit reaches the target, the unit will attempt to steal or request food from the target. If the unit has requested the resource, then the theft norm will be incremented by one, and both the unit and the target will have their opinions of each other incremented by three. If the unit is stealing from the target, then the theft norm will be decremented by one, and both the unit and the target will have their opinions of each other decremented by seven. In both cases the unit will gain an instance of the resource, and the target will lose an instance of the resource.

If the unit chooses to move randomly around the simulation world, the control mechanism chooses a random point in the world and sets that as the unit's target. The pathfinder then finds the shortest path, and the unit is given a velocity. The unit will continue doing this until it reaches the target point.

If the unit chooses to interact with another unit, the procedure is similar to that of obtaining a resource from another unit. The first step is to determine what action should be performed. The control mechanism chooses a random target unit and passes it to the requesting unit. If the sum of the unit's temperament, the current value of the violence norm, and the unit's opinion of the target is greater than 25 but less than or equal to 45, then the unit will hug the target. When the unit reaches the target, the violence norm value will be incremented by one. The unit and the target will also have their opinions of each other incremented by five, and the

target will have its health incremented by ten. If the sum of the unit's temperament, the current value of the violence norm, and the unit's opinion of the target is greater than 45, then the unit will kiss the target. When the unit reaches the target, the violence norm value will be incremented by three. The unit and target will also have their opinions of each other incremented by ten, and the target will have its health incremented by 20. If the sum of the unit's temperament, the current value of the violence norm, and the unit's opinion of the target is less than -25 but greater than or equal to -45, then the unit will slap the target. When the unit reaches the target, the violence norm value will be decremented by one. The unit and target will also have their opinions of each other decremented by five, and the target will have its health decremented by ten. If the sum of the unit's temperament, the current value of the violence norm, and the unit's opinion of the target is less than -45, but greater than or equal to -65, then the unit will punch the target. When the unit reaches the target, the violence norm value will be decremented by three. The unit and target will also have their opinions of each other decremented by ten, and the target will have its health decremented by 20. If the sum of the unit's temperament, the current value of the violence norm, and the unit's opinion of the target is less than -65, then the unit will kill the target. When the unit reaches the target, the violence norm will be decremented by five and the target unit will be considered dead.

If the unit's opinion of the target is greater than 15, then the unit will attempt to mate with the target. When the unit reaches the target, the control mechanism will determine whether or not mating is feasible. If there are less than 20 of each kind of resource in the resource boxes, or if other units surround the target unit, then the mating procedure will fail. This part of the design represents an individual's choice not to procreate when the population is overcrowded or when there is a resource shortage. If the mating procedure succeeds based on these

requirements, then a fully functional unit is added to the population. Including a childhood phase in a unit's lifespan was deemed inconsequential, primarily because this would add unnecessary complexity to the simulation. The new unit will be created in a free space next to the target unit. It will have a temperament value equal to the integer value of the average of its parent units. The new unit will also have an initial set of opinions. The new unit will have a value of 50 as its opinion value for each of its parent units. It will also have the combined opinions of both parents. In the case where the two parents have differing opinions of a unit, the integer average will be used. Additionally, each parent unit will have its opinion of the other set to 50, and each parent's opinion of their offspring will also be 50. To provide a way to counteract the killing action, the mating action will increment the violence norm by five. Early versions of the simulation were designed such that mating had no effect on the violence norm, and it appeared that this lack of balance would cause the violence norm to tend toward the lower end of the range. Including mating in the norm calculation is justified, by reasoning that births will have a mediating effect on a society and thus lower the society's proclivity toward violence. This can be seen in among primates, particularly baboons, where fighting males will often grab an infant child in order to stop a conflict in progress [16]. If none of these conditions are met, then the unit will attempt to simply talk to the target. This action has no effect on the norm system and merely stands to provide a way for units to modify their opinions of each other. If the difference between the temperament of the unit and its target is less than 50, then the talking act is seen as successful. If the talking act is successful, then each unit's opinion of each other will be incremented by one. If the talking act is unsuccessful, then each unit's opinion of each other will be decremented by one. This is justified by reasoning that two people with radical

differences in their personal predilection towards good or evil will invariably not be compatible, even in a simple social sense.

Formal Definitions

Definition 1: (Unit). *For all units, U_X at time t, let T_X be the unit's temperament such that* $-50 \leq T_X \leq 50$. *Let A_t^X be the unit's age at time t, such that* $0 \leq A_t^X$. *Let H_t^X be the unit's current health at time t where* $0 \leq H_t^X \leq 99$. *Let Df_t^X be the unit's desire for food (hunger) at time t where* $0 \leq Df_t^X$, *Dw_t^X be the unit's desire for water (thirst) at time t where* $0 \leq Dw_t^X$. *Let F_t^X be the amount of food the unit is carrying at time t where* $0 \leq F_t^X \leq 3$, *and W_t^X be the amount of water the unit is carrying at time t where* $0 \leq W_t^X \leq 3$.

Definition 2: (Opinions). *For all units, U_X and U_Y at time t, let $O_t^{X,Y}$ be the opinion that U_X has for U_Y at time t, such that* $-50 \leq O_t^{X,Y} \leq 50$, *unless X=Y, in which case* $O_t^{X,Y} = 50$.

Definition 3: (Norms). *Let S_t be the stealing norm for the simulation at time t where* $-50 \leq S_t \leq 50$, *and V_t be the violence norm at time t where* $-50 \leq V_t \leq 50$.

Definition 4: (Resource Containers). *Let C_t^F be the amount of food in the food resource box at time t where* $0 \leq C_t^F$, *and let C_t^W be amount of water in the water resource box at time t where* $0 \leq C_t^W$.

Definition 5: (Actions). *For all units, U_X at time t, Let B_t^X be the unit's basic action at turn t such that* $B_t^X \in \{Idle, Move\ Randomly, Eat, Drink, Obtain\ Food, Obtain\ Water, Interact\}$.

Let R_t^X *be the unit's resource action at turn t, such that if*

$B_t^X \in \{Obtain\ Food, Obtain\ Water\}$ *then*

$R_t^X \in \{Get\ Food, Get\ Water, Beg\ Food, Beg\ Water, Steal\ Food, Steal\ Water\}$ *and*

$R_t^X = \emptyset$ *otherwise. Let* I_t^X *be the unit's inter-unit interaction at time t such that if* $B_t^X =$

$Interact$*, then* $I_t^X \in \{Talk, Hug, Kiss, Mate, Slap, Punch, Kill\}$ *and* $I_t^X = \theta$ *otherwise.*

Definition 6: (Unit age).

$$A_{t+1}^X = A_t^X + 1$$

Definition 7: (Hunger).

$$Df_{t+1}^X = \begin{Bmatrix} 0 \ \rightarrow if\ the\ unit\ eats\ during\ turn\ t\ or\ if\ t = -1 \\ Df_t^X + 1 \ \rightarrow \ otherwise \end{Bmatrix}$$

Definition 8: (Thirst).

$$Dw_{t+1}^X = \begin{Bmatrix} 0 \ \rightarrow if\ the\ unit\ drinks\ during\ turn\ t\ or\ if\ t = -1 \\ Dw_t^X + 1 \ \rightarrow \ otherwise \end{Bmatrix}$$

Definition 9: (Health).

$$H_{t+1}^{X} = \left\{ \begin{array}{c} 0 \rightarrow if\ t = -1 \\ else\ H_t^X - 1 \rightarrow\ if\ Df_t^X > 99\ and\ Dw_t^X < 99 \\ else\ H_t^X - 2 \rightarrow\ if\ Df_t^X < 99\ and\ Dw_t^X > 99 \\ else\ H_t^X - 3 \rightarrow\ if\ Df_t^X > 99\ and\ Dw_t^X > 99 \\ else\ H_t^X - 10 \rightarrow\ if\ the\ unit\ was\ slapped\ during\ turn\ t \\ else\ H_t^X + 10 \rightarrow if\ the\ unit\ was\ hugged\ during\ turn\ t \\ else\ H_t^X - 20 \rightarrow if\ the\ unit\ was\ punched\ during\ turn\ t \\ else\ H_t^X + 20 \rightarrow if\ the\ unit\ was\ kissed\ during\ turn\ t \\ else\ H_t^X - 30 \rightarrow if\ the\ unit\ was\ slapped\ and\ bunch\ during\ turn\ t \\ else\ H_t^X + 30 \rightarrow if\ the\ unit\ was\ hugged\ and\ kissed\ during\ turn \\ else\ H_t^X \rightarrow\ otherwise \end{array} \right\}$$

Definition 10: (Basic Action Determination).

$$B_t^X = \left\{ \begin{array}{c} Eat \rightarrow if\ F_t^X > 0\ and\ Df_t^X > 49\ and\ Dw_t^X \leq 49 \\ else\ Eat \rightarrow if\ F_t^X > 0\ and\ W_t^X = 0\ and\ Df_t^X > 49\ and\ Dw_t^X > 49 \\ else\ Drink \rightarrow\ if\ W_t^X > 0\ and\ Dw_t^X > 49\ and\ Df_t^X \leq 49 \\ else\ Drink \rightarrow if\ F_t^X = 0\ and\ W_t^X > 0\ and\ Df_t^X > 49\ and\ Dw_t^X > 49 \\ else\ Obtain\ Food \rightarrow if\ F_t^X = 0\ and\ Df_t^X > 49\ and\ Dw_t^X \leq 49 \\ else\ Obtain\ Food \rightarrow if\ F_t^X = 0\ and\ Df_t^X > 49\ and\ Dw_t^X > 49\ and\ Df_t^X > \ Dw_t^X \\ else\ Obtain\ Water \rightarrow if\ W_t^X = 0\ and\ Dw_t^X > 49\ and\ Df_t^X \leq 49 \\ else\ Obtain\ Water \rightarrow if\ W_t^X = 0\ and\ Df_t^X > 49\ and\ Dw_t^X > 49\ and\ Dw_t^X > \ Df_t^X \\ else\ Random\ Action \rightarrow otherwise \end{array} \right\}$$

Definition 11: (Food Resource Interaction). *If* $C_t^F = 0$ *then it becomes possible that* $\exists$ *a* U_Y *that has been targeted by* U_X *for a resource action and* U_Y *possesses food. Then*

$$R_t^X = \left\{ \begin{array}{c} Get\ Food \rightarrow if\ B_t^X = Obtain\ Food\ and\ C_t^F > 0 \\ Beg\ Food \rightarrow if\ B_t^X = Obtain\ Food\ and\ C_t^F = 0\ and\ T_X + O_t^{X,Y} + S_t \geq 0 \\ Steal\ Food \rightarrow if\ B_t^X = Obtain\ Food\ and\ C_t^F = 0\ and\ T_X + O_t^{X,Y} + S_t < 0 \end{array} \right\}$$

Definition 11a: (Get Food). *Thus if* $R_t^X = Get\ Food$, *then the following will also be true.*

$$F_{t+1}^X = F_t^X + 1$$

And

$$C_{t+1}^{F} = C_{t}^{F} - 1$$

Definition 11b: (Beg Food). *If* $R_t^X = Beg\ Food$ *then,*

$$F_{t+1}^{X} = F_t^X + 1$$

And

$$F_{t+1}^{Y} = F_t^Y - 1$$

And

$$O_{t+1}^{X,Y} = O_t^{X,Y} + 3$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} + 3$$

The stealing norm will also be affected:

$$S_{t+1} = S_t + 1$$

Definition 11c: (Steal Food). *If* $R_t^X = Steal\ Food$ *then,*

$$F_{t+1}^{X} = F_t^X + 1$$

And

$$F_{t+1}^{Y} = F_t^Y - 1$$

And

$$O_{t+1}^{X,Y} = O_t^{X,Y} - 7$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} - 7$$

The stealing norm will also be affected:

$$S_{t+1} = S_t - 1$$

Definition 12: (Water Resource Interaction). *If* $C_t^W = 0$ *then it becomes possible that* ∃ *a* U_Y *that has been targeted by* U_X *for a resource action and* U_Y *possesses water. Then*

$$R_t^X = \begin{Bmatrix} Get\ Water \rightarrow if\ B_t^X = Obtain\ Water\ and\ C_t^W > 0 \\ Beg\ Water \rightarrow if\ B_t^X = Obtain\ Water\ and\ C_t^W = 0\ and\ T_X + O_t^{X,Y} + S_t \geq 0 \\ Steal\ Water \rightarrow if\ B_t^X = Obtain\ Water\ and\ C_t^W = 0\ and\ T_X + O_t^{X,Y} + S_t < 0 \end{Bmatrix}$$

Definition 12a: (Get Water). *Thus if* $R_t^X = Get\ Water$*, then the following will also be true.*

$$W_{t+1}^X = W_t^X + 1$$

And

$$C_{t+1}^W = C_t^W - 1$$

Definition 12b: (Beg Water). *If* $R_t^X = Beg\ Water$ *then,*

$$W_{t+1}^X = W_t^X + 1$$

And

$$W_{t+1}^Y = W_t^Y - 1$$

And

$$O_{t+1}^{X,Y} = O_t^{X,Y} + 3$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} + 3$$

The stealing norm will also be affected:

$$S_{t+1} = S_t + 1$$

Definition 12c: (Steal Water). $If\ R_t^X = Steal\ Water\ then,$

$$W_{t+1}^X = W_t^X + 1$$

And

$$W_{t+1}^Y = W_t^Y - 1$$

And

$$O_{t+1}^{X,Y} = O_t^{X,Y} - 7$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} - 7$$

The stealing norm will also be affected:

$$S_{t+1} = S_t - 1$$

Definition 13: (Inter-unit Interaction Decision). $B_t^X = Interact$, *and* U_Y *is the current interaction target of* U_X. *Then*

$$I_t^X = \left\{ \begin{array}{c} Hug \rightarrow if\ T_X + O_t^{X,Y} + V_t > 25\ and\ T_X + O_t^{X,Y} + V_t \leq 45 \\ Slap \rightarrow if\ T_X + O_t^{X,Y} + V_t < -25\ and\ T_X + O_t^{X,Y} + V_t \geq -45 \\ Kiss \rightarrow if\ T_X + O_t^{X,Y} + V_t > 45 \\ Punch \rightarrow if\ T_X + O_t^{X,Y} + V_t < -45\ and\ T_X + O_t^{X,Y} + V_t \geq -65 \\ Mate \rightarrow if\ O_t^{X,Y} > 15 \\ Kill \rightarrow if T_X + O_t^{X,Y} + V_t < -65 \\ Talk \rightarrow otherwise \end{array} \right\}$$

Definition 13a: (Talking Interaction).

$$O_{t+1}^{X,Y} = \begin{Bmatrix} O_t^{X,Y} + 1 \rightarrow if\ |T_X - T_Y| < 50 \\ O_t^{X,Y} - 1 \rightarrow otherwise \end{Bmatrix}$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = \begin{Bmatrix} O_t^{Y,X} + 1 \rightarrow if\ |T_X - T_Y| < 50 \\ O_t^{Y,X} - 1 \rightarrow otherwise \end{Bmatrix}$$

Definition 13b: (Hug Interaction). $If\ I_t^X = Hug, then$

$$O_{t+1}^{X,Y} = O_t^{X,Y} + 5$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} + 5$$

The health of U_Y *will be affected:*

$$H_{t+1}^{Y} = H_t^{Y} + 10$$

The violence norm will also be affected:

$$V_{t+1} = V_t + 1$$

Definition 13c: (Slap Interaction). $If\ I_t^X = Slap, then$

$$O_{t+1}^{X,Y} = O_t^{X,Y} - 5$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} - 5$$

The health of U_Y *will be affected:*

$$H_{t+1}^{Y} = H_t^{Y} - 10$$

The violence norm will also be affected:

$$V_{t+1} = V_t - 1$$

Definition 13d: (Kiss Interaction). *If* $I_t^X = Kiss$, *then*

$$O_{t+1}^{X,Y} = O_t^{X,Y} + 10$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} + 10$$

The health of U_Y *will be affected:*

$$H_{t+1}^{Y} = H_t^{Y} + 20$$

The violence norm will also be affected:

$$V_{t+1} = V_t + 3$$

Definition 13e: (Punch Interaction). *If* $I_t^X = Punch$, *then*

$$O_{t+1}^{X,Y} = O_t^{X,Y} - 10$$

Similarly, $O_{t+1}^{Y,X}$ *will be affected:*

$$O_{t+1}^{Y,X} = O_t^{Y,X} - 10$$

The health of U_Y *will be affected:*

$$H_{t+1}^{Y} = H_t^{Y} - 20$$

The violence norm will also be affected:

$$V_{t+1} = V_t - 3$$

Definition 13f: (Mating Interaction). *If* $I_t^X = Mate$, *and* $C_t^F \geq 20$, $C_t^W \geq 20$, *and* U_Y *is not surrounded by other units. Then*

$$V_{t+1} = V_t + 5$$

Furthermore, a new unit, U_Z, will be created next to unit U_Y. Let O_t^Z be the set of all opinions belonging to U_Z. Let T_Z be the temperament of U_Z. Thus

$$O_t^Z = O_t^X \cup O_t^Y$$

And

$$T_Z = \frac{T_X + T_Y}{2}$$

And

$$O_{t+1}^{Z,X} = 50$$

And

$$O_{t+1}^{Z,Y} = 50$$

And

$$O_{t+1}^{X,Y} = 50$$

And

$$O_{t+1}^{X,Z} = 50$$

And

$$O_{t+1}^{Y,X} = 50$$

And

$$O_{t+1}^{Y,Z} = 50$$

Definition 13g: (Kill Interaction). *If $I_t^X = Kill$, then*

$$P_{t+1} = P_t - 1$$

And U_Y will be considered dead:

$$U_Y = \emptyset$$

Furthermore the violence norm will be decremented:

$$V_{t+1} = V_t - 5$$

Design Summary

Overall the concept was to create a representation of a very simple human society or, at the very least, a representation of the society of organized primates. To fulfill this goal, a simulation was created to mimic society where the most basic of human needs, food and water, are represented. It could be said that human interaction is a basic human need, but the inclusion of the norm system, as well as the system of entity interaction, more than adequately represent this need. Despite the fact that the simulation did not quantify the need for interaction, as was done with hunger and thirst, the belief is that this representation is still valid. In this simulation, the units seek interaction only when their resource needs are met, and when they do seek interaction, they do it rather randomly. However, in the most basic sense, this is how humans behave in the real world. If the sociological complexity is removed from the equation, it can be theorized that humans are not inclined to search for interaction when they are starving or dying of thirst. Only when these baser needs are fulfilled do humans seek interaction, and that is only if something else doesn't pique our interest first. A simulation has been created with actions that are representative of basic human interaction, and such that life and death are an integral part.

Methodology

When running the simulation, the user has two options. The user can choose either a graphical mode or a faster mode that has no graphical output. The user's input can be conferred via either the graphical interface or the command line. Using the graphical interface to input the

initial values doesn't limit the user to using the graphical mode, nor does using the command line for input restrict the user from using that mode. The simulation also has the option of outputting the final action log from a simulation run to a text file. The user provides several variables that determine the initial starting configuration of the simulation. These variables include a grid size, a starting unit count, a maximum unit count, a starting amount for food in the food resource box, a starting amount for water in the water resource box, a median temperament for the units, an initial value for the theft norm, an initial value for the violence norm, and a maximum number of turns for the simulation. If the user chooses to run the simulation in graphical mode, then they will be able to see the location of each unit during every turn. These units will be color coded by temperament. Units with a temperament value less than -25 will be red, units with a temperament value above 25 will be green, and all other units will be yellow. The graphical interface also shows what actions have been performed during the current turn, what the current status of the resource boxes and norm system values are, the current turn number, and the current number of units. The user can control, via the graphical interface, the speed of the simulation and the amount of resources in the resource boxes.

This experiment was designed to test varying levels of average temperament, violence norm, and theft norm for a set period of time. The simulation was set to start with 5 units, have a maximum unit count of 15, have 1000 of each food and water in the resource boxes, have a 25-turn wait for refilling an empty resource box, and have a 30,000 turn limit. The primary tests performed focused on the average temperament setting. The values used for this setting were -50, -25, 0, 25, and 50 for the average temperament, and zero for the theft and violence norms. These were intended to act as a baseline for the other tests that were performed and as a way to limit the number of tests that needed to be performed in the other areas. By using these as a baseline, data

can be extrapolated from these, when the data from tests involving the norm system settings were analyzed. Further tests were performed on a smaller basis and involved setting the violence norm to -50, -25, 0, 25, and 50 with the average temperament values detailed above for each of the norm system settings. A similar test was performed using the same values for the theft norm, with the same average temperament values. Both of these tests were done in smaller quantities than the initial average temperament tests.

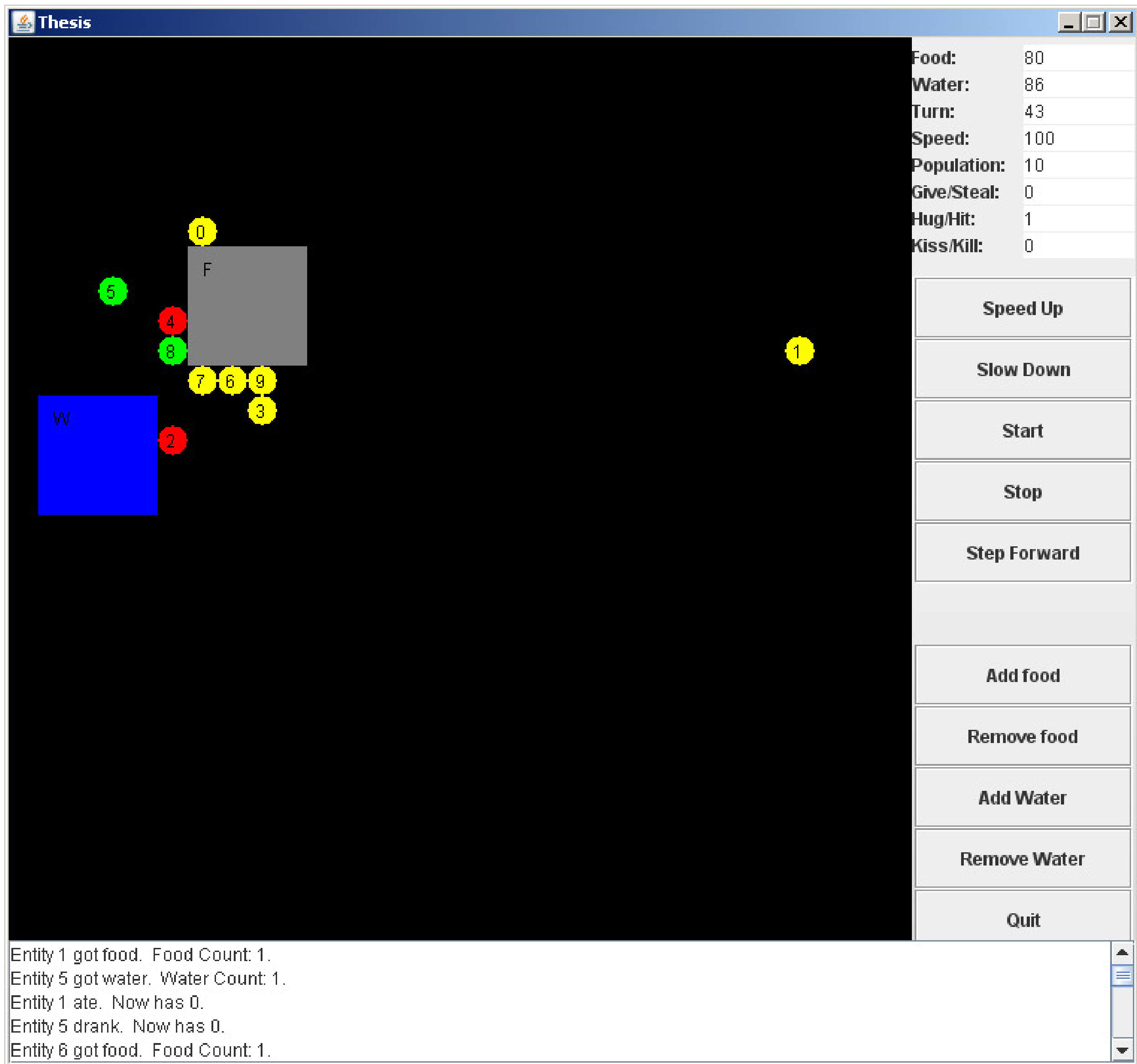

Figure 0 - Screenshot of the Simulation During a Visualized Run

CHAPTER IV

ANALYSIS OF RESULTS

As detailed previously, all the simulations were run with an initial population of five and a population cap of fifteen. The simulation board is a 20 x 20 grid. The resource boxes are filled with 1000 units each to begin and refill after 25 turns of being empty. All the simulations are set to end at turn 30,000. First, the simulation was set to test multiple configurations of the average temperament value. The values used were -50, -25, 0, 25, and 50, and 500 instances of each of these configurations were run. Beyond this, tests on different configurations of the stealing and violence norm with the previously defined average temperament values were also run, but with only 200 instances.

The first test performed was a base case where the average temperament, stealing norm, and violence norm values were set at zero. In the graphic, one can see that the simulation duration in turns was either ended at turn 10,001 or made it to the 30,000-turn limit that was set, with very few cases falling between the two. This can be attributed to a lack of reproduction in the simulation. If the units in the simulation failed to reproduce, then the simulation ends at turn 10,001 after the initial generation has died off. Another problem can occur if the simulation has a high murder rate. If four of the five initial units are murdered before any reproduction can occur, then the sole unit left will simply while away the last few remaining turns alone. Once they die, the simulation ends.

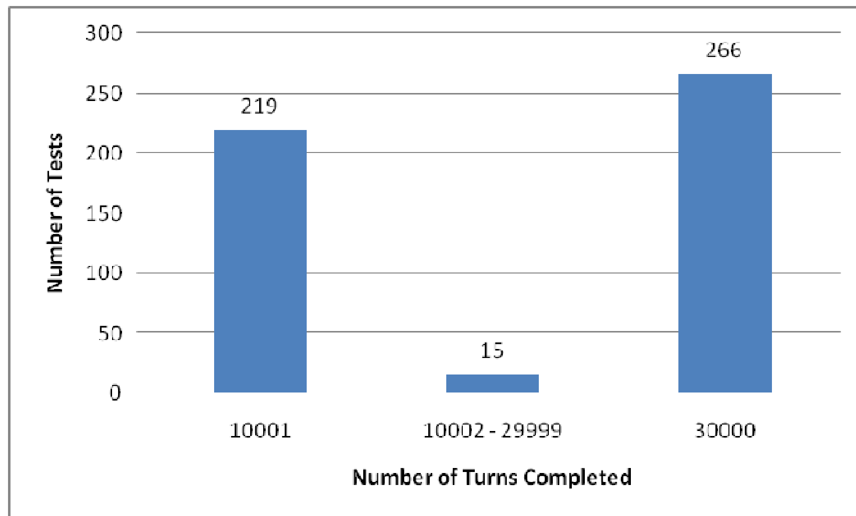

Figure 1 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 0 Violence Norm, and 0 Stealing Norm

If the number of births as a function of the number of turns elapsed is graphed, the evidence that the birth rate definitely contributes to the longevity of the simulation is shown (Fig. 2.)

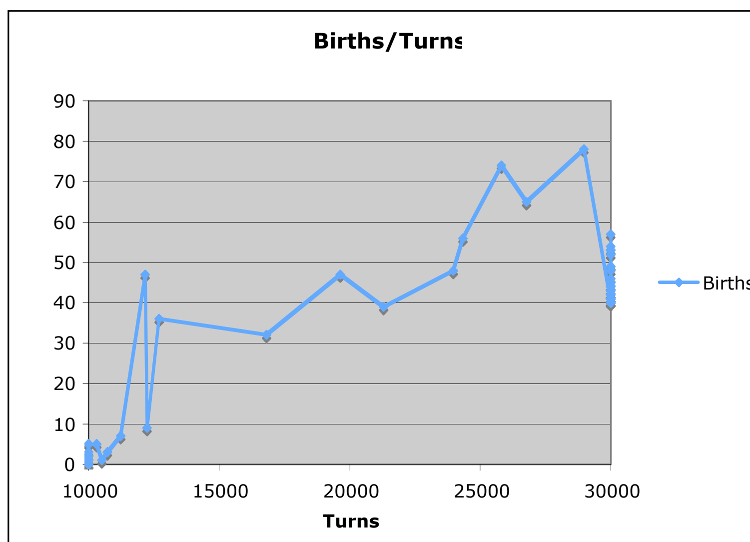

Figure 2 - Number of Births over Turns Elapsed using the settings 0 Temperament, 0 Violence Norm, and 0 Stealing Norm

Equally, when graphing the number of murders as a function of the number of turns elapsed, the fact that the murder rate also has an effect on the longevity of the simulation can be seen (Fig. 3.)

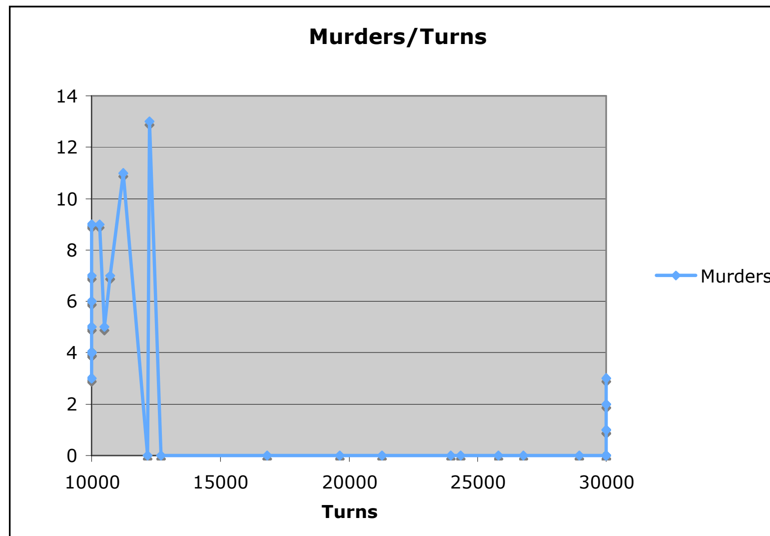

Figure 3 - Number of Murders as a Function of Turns Elapsed, 0 Temperament, Violence Norm, Stealing Norm

It is not just the murders and births that affect the simulation, since these are just side-effects of the larger sense of harmony or discord that permeates the simulation. This sense can be seen in the final values of the stealing and violence norm.

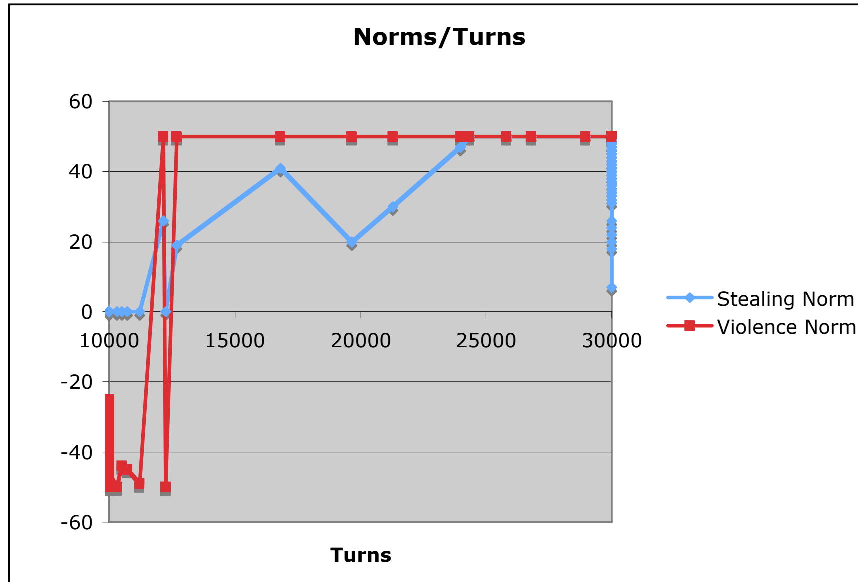

Figure 4 - Final Norm Values over Turns Elapsed using the settings 0 Temperament, 0 Violence Norm, and 0 Stealing Norm

Figure 4 shows how the norms affect the longevity of the simulation. One can see that in the shortest-lived simulations, the violence norm is well below zero. However, as the length of the simulations' life increases, the violence norm levels increase dramatically. One can also see that the stealing norm appears to have some effect on the simulations' longevity but not nearly as much as the violence norm.

Overall, this initial test on the base case shows that basically the success of the simulation depends on the units that inhabit it. If the randomly generated denizens of each simulation tend to be more violent than the norm, the simulation will fail. However, if the denizens choose to live in harmony with one another, then the simulation will succeed. This is fortunate, because it means the simulation is doing a good job representing the real world. One can easily see how the results of the base case are representative of what would happen in a real world simulation. Obviously a society that cannot live together peacefully will not survive long. Conversely a

society that can live in harmony may survive for a long time. Note that when a society is discussed, this is simply referring to a group of individuals who inhabit an area and interact with one another, and not a more complex definition of a society.

To reinforce the findings made in the base case, the simulation was tested with the lower average temperament values -25 and -50. The results are concurrent with what was found in testing the base case. The lower the average temperament value, the less likely the simulation is to reach the 30,000 turn limit. As one would expect, the final norm values support this.

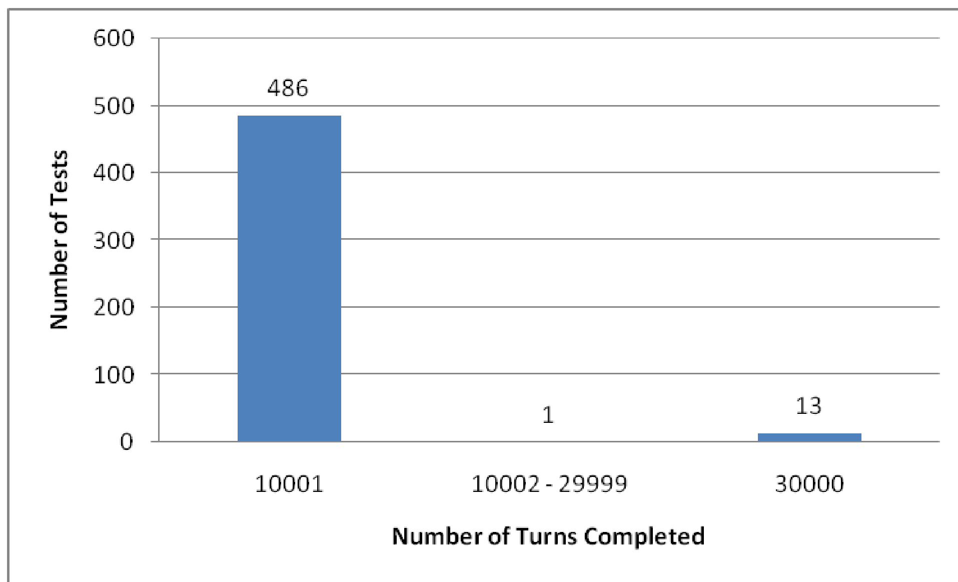

Figure 5 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 0 Violence, and 0 Stealing

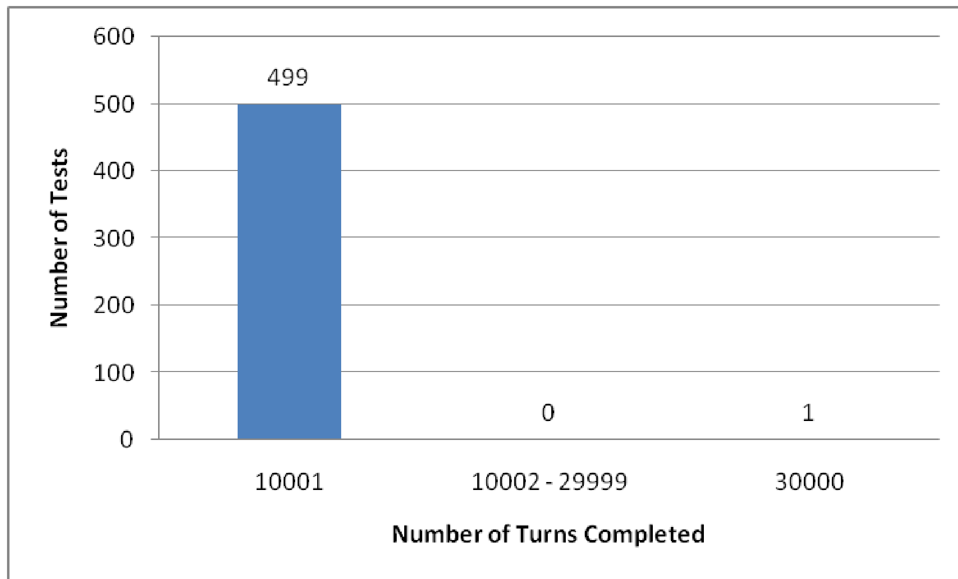

Figure 6 - Number of Tests that Completed a Certain Number of Turns using the settings - 50 Temperament, 0 Violence, and 0 Stealing

When the corresponding positive average temperament values of 25 and 50 were tested, the results were concurrent with the pattern that has developed regarding the average temperament value. In this case, the higher the average temperament, the more likely the simulation is to reach the 30,000 turn limit. This too is also supported by the final norm readings.

The previous tests show that when all norms are set to zero at the starting point, the simulation basically depends on the temperament of its inhabitants to determine its success. This is supported by the observation that if a norm starts to drift toward one extreme, it tends to continue toward that extreme. It should also be noted that the lower the individual's temperament, the more likely the individual is to partake in actions that reduce the norm values; conversely, the higher the temperament, the higher the proclivity toward actions that increase norm values. With that in mind, since the only thing affecting the norms is the actions of the

individuals in the society, then whatever actions those individuals partake in will determine the norms.

When this simulation was created, there was particular interest in learning how a group of individuals would react to being placed under the influence of a norm system that reinforced actions counter to those the individuals would choose without the norm system. To test this, values of -50, -25, 25, and 50 were applied separately to each of the norms. For each of the norm values, the average temperament is set to -50, -25, 0, 25, and 50. All other settings are set to the same as the previous tests. The number of instances of each test was shortened, from 500 to 200. Since it was already established how the simulation will act based on the average temperament, it can be assumed that the same will be true for the smaller set of instances in these tests.

First, the stealing norm was tested with the various values of average temperament. Starting with pure, or zero value, the results show some interesting yet rather expected results. Since it has already been shown that both the number of murders and birth and the final norm values support the findings regarding the length of a simulation, henceforth this paper will only discuss the total number of turns elapsed for each test.

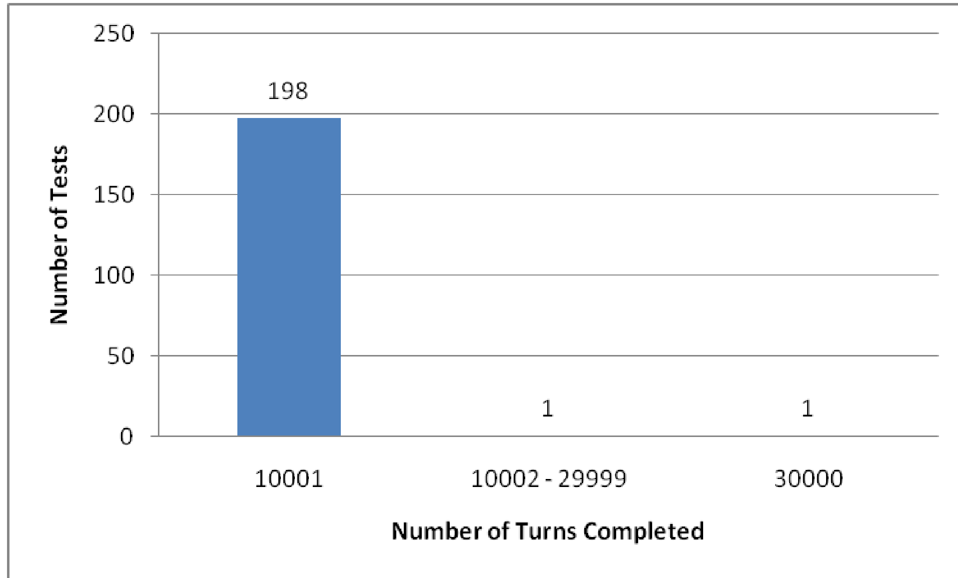

Figure 7 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 0 Violence, and -50 Stealing

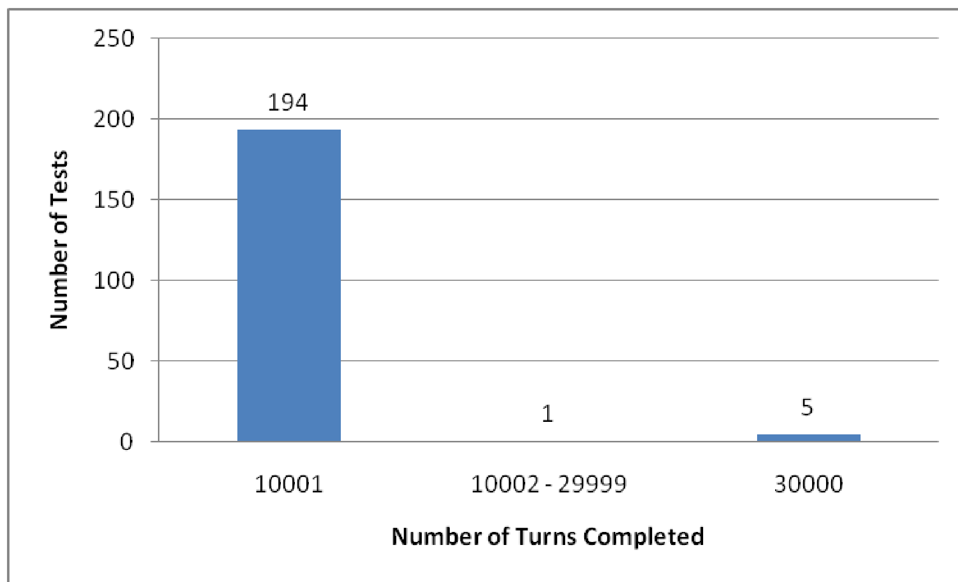

Figure 8 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 0 Violence, and -25 Stealing

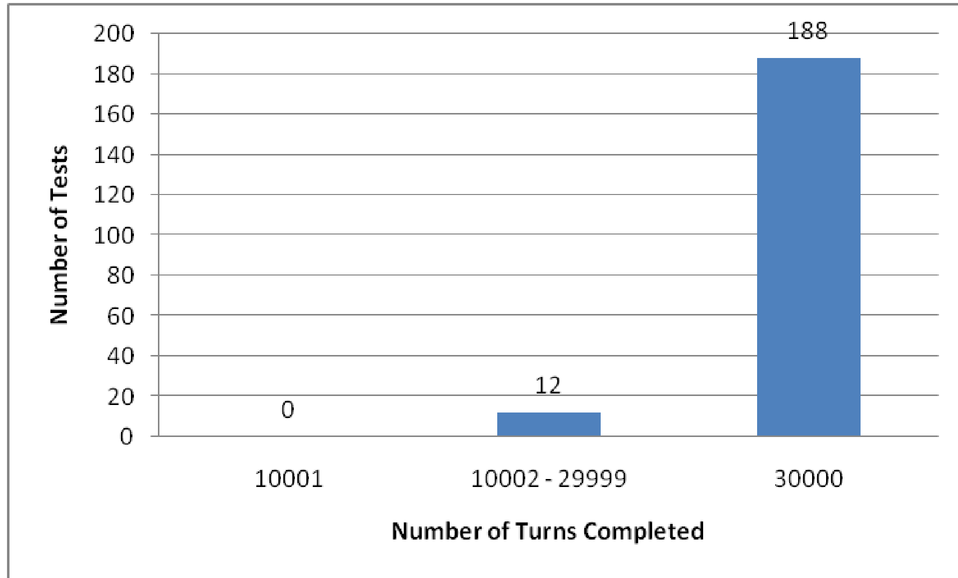

Figure 9 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 0 Violence, and 25 Stealing

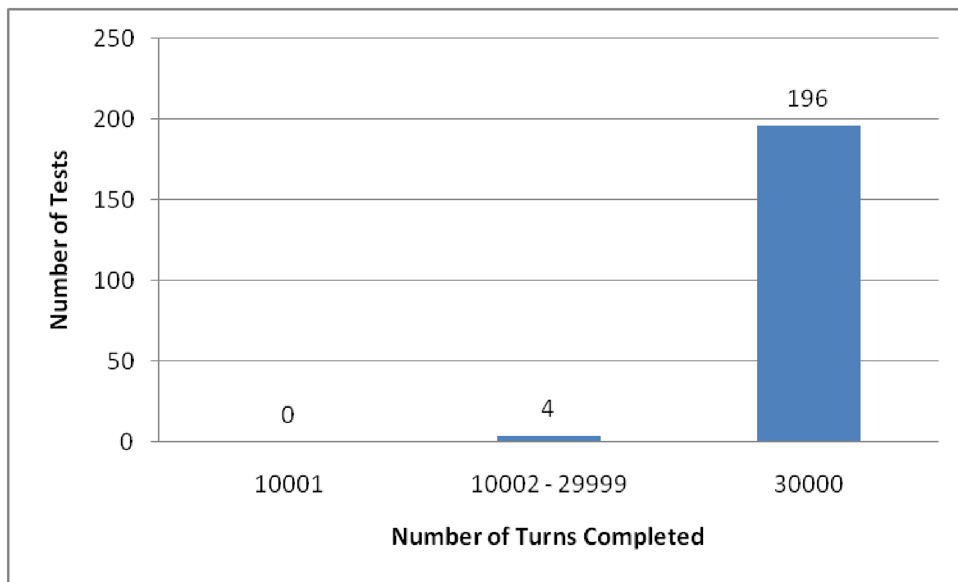

Figure 10 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 0 Violence, and 50 Stealing

Recalling the results shown in Figure 1, the new tests show that the differing values of the stealing norm have a great deal of effect on the longevity of the simulation. The original tests showed that there was a nearly equal distribution of the two extremes, failing at turn 10,001 and continuing on to turn 30,000. When applying the negative values for the stealing norm, the new tests show that very few simulations last to the 30,000-turn limit. Conversely, when applying the positive norm values; the new tests show that almost all the simulations make it to the turn limit.

If, as the previous tests show, the stealing norm has an effect on the longevity of a simulation, then its ability to do so should be either helped or hindered by differing values of the simulations average temperament. The next tests performed were the same as the previous set, except they were done using the negative average temperament values.

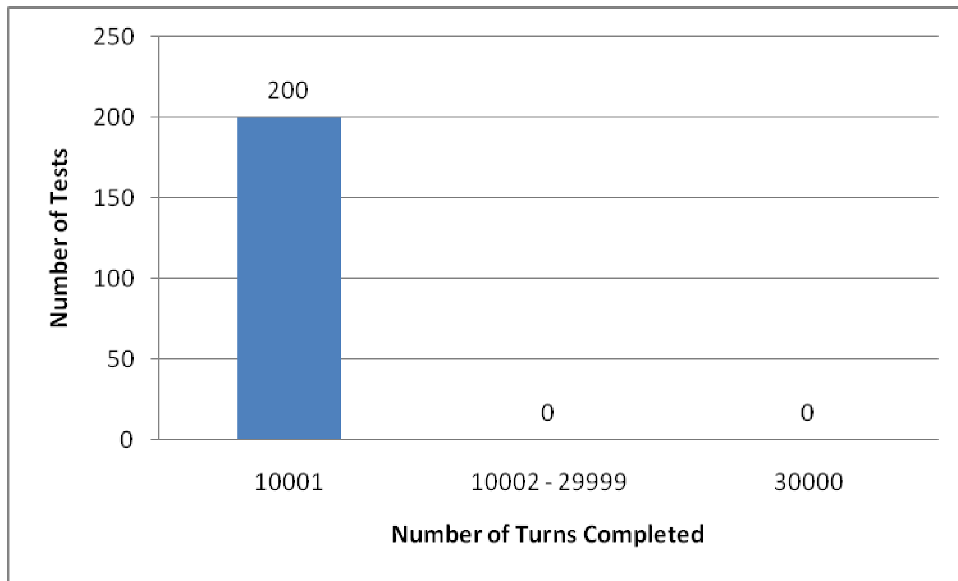

Figure 11 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 0 Violence, and -50 Stealing

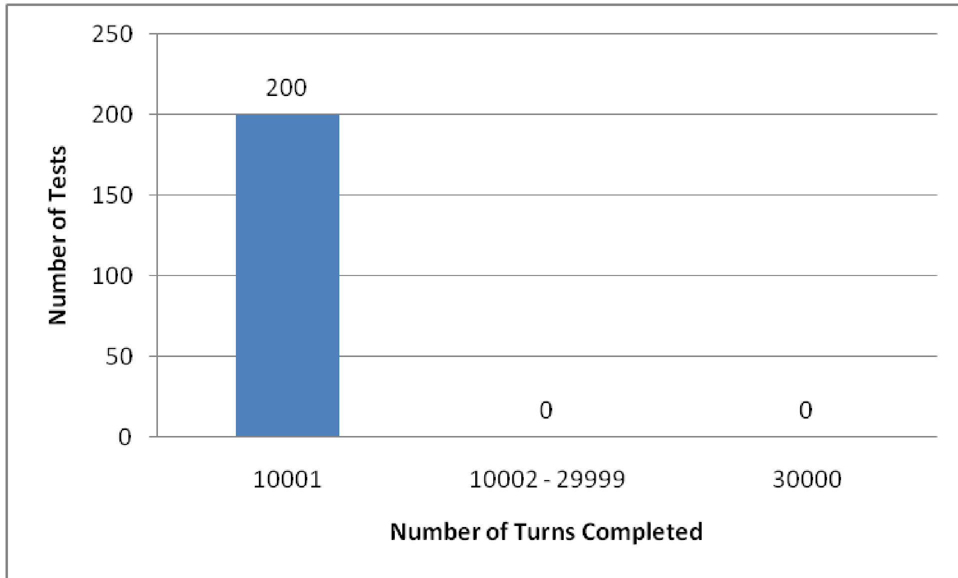

Figure 12 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 0 Violence, and -25 Stealing

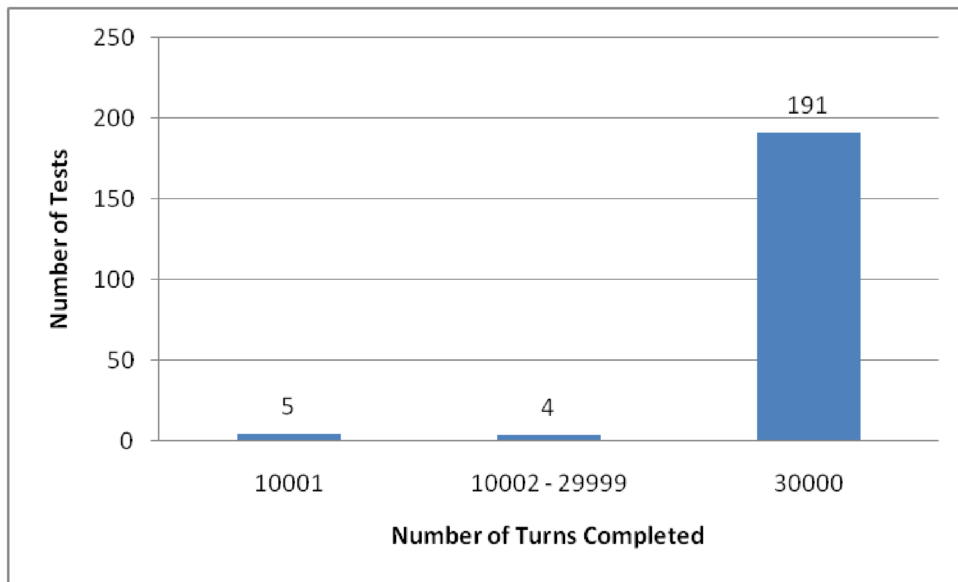

Figure 13 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 0 Violence, and 25 Stealing

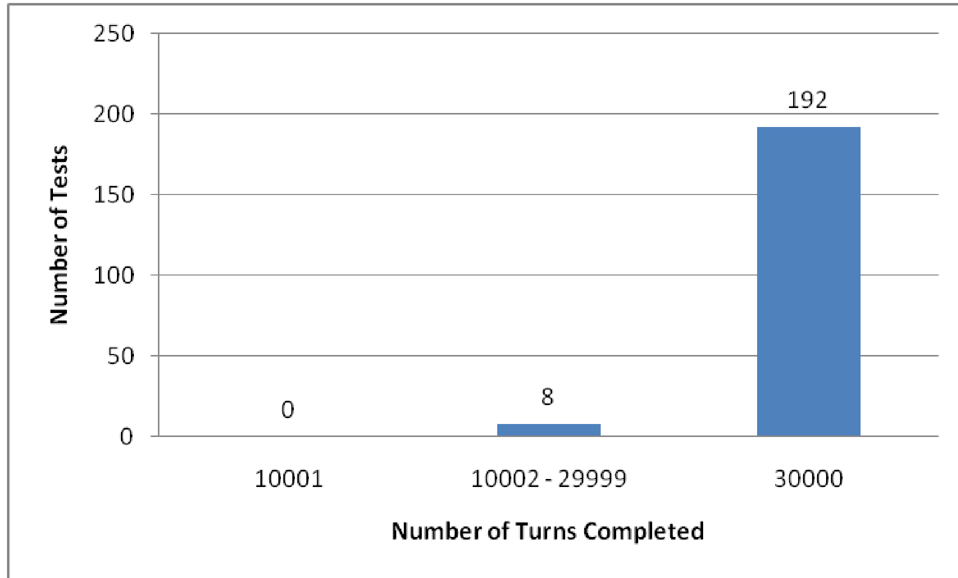

Figure 14 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 0 Violence, and 50 Stealing

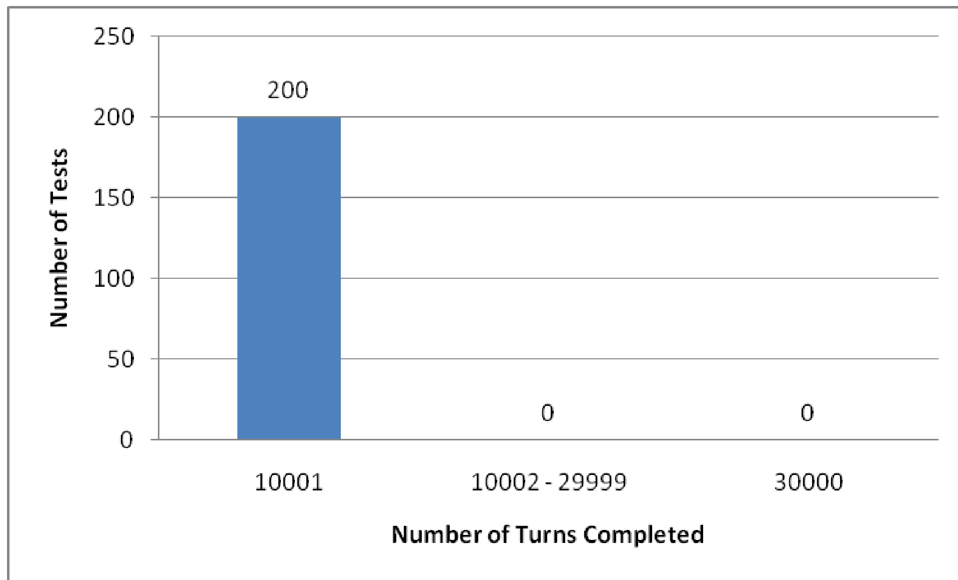

Figure 15 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 0 Violence, and -50 Stealing

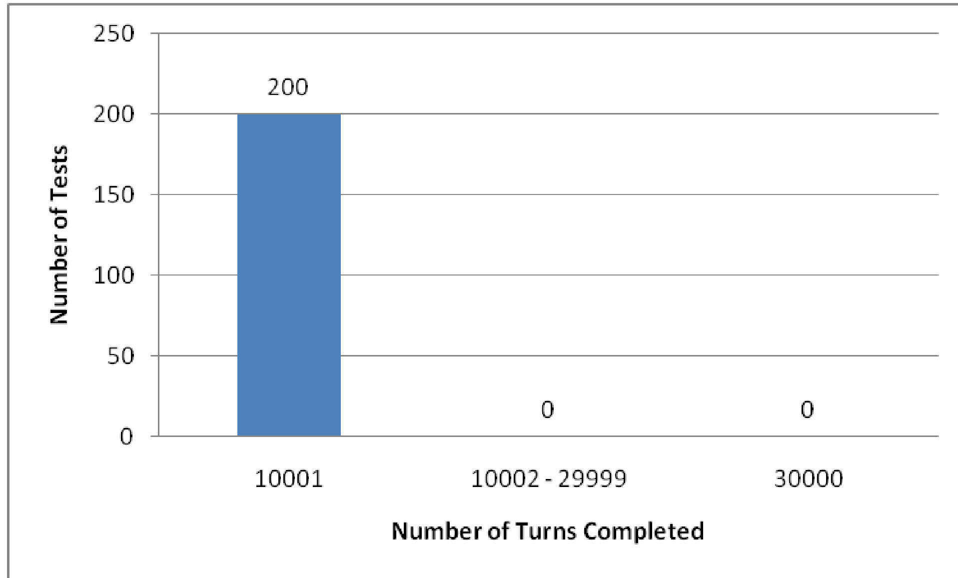

Figure 16 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 0 Violence, and -25 Stealing

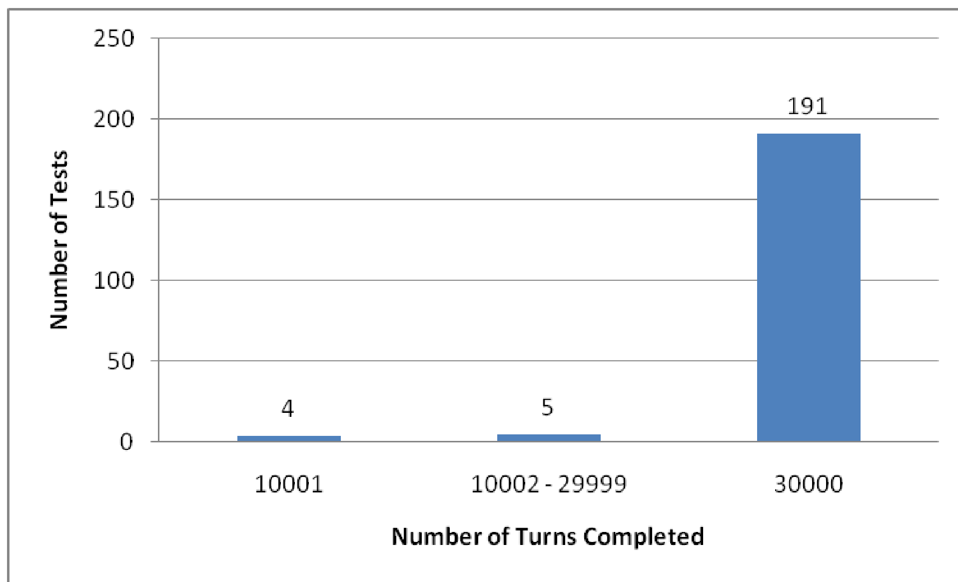

Figure 17 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 0 Violence, and 25 Stealing

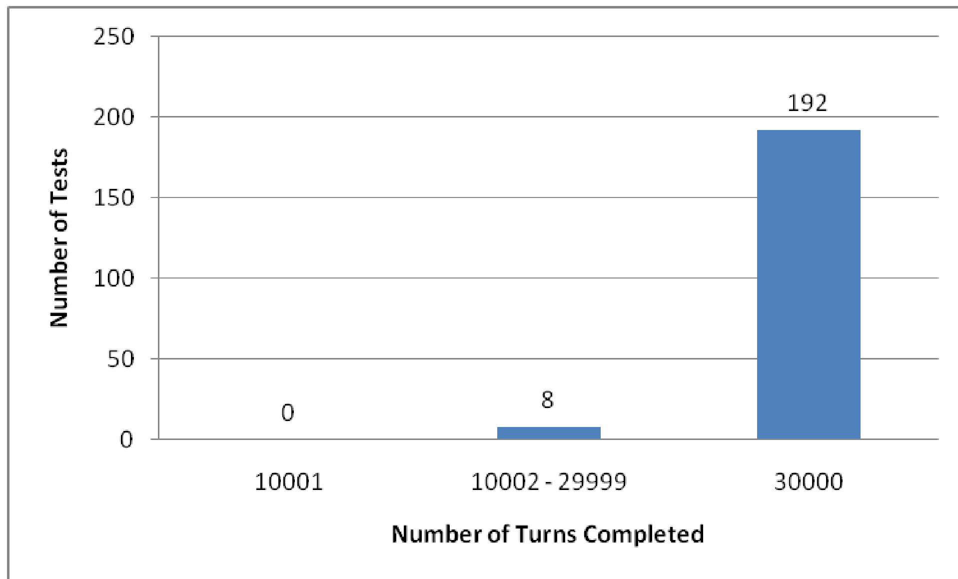

Figure 18 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 0 Violence, and 50 Stealing

From these results one can clearly see that the stealing norm has a great deal to do with the longevity of the simulation. Upon comparing these results to the initial results done on the stealing norm, it is apparent that the stealing norm will make an impact on the final results. If the stealing norm and average temperament value are both negative, then the longevity of the simulation is pushed toward the negative extreme of 10,001. In some cases the tests achieved a uniformity of the final result, in that all the final longevity results were the same. Looking back to the original tests done on the negative temperament values shows that these original tests were unlikely to reach the 30,000-turn limit. Surprisingly, when positive stealing norm values are applied, the simulation is almost guaranteed to reach the 30000-turn limit. This shows that the stealing norm has a large effect on the simulation's longevity.

In order to conclusively show that the stealing norm has a large effect on the longevity of the simulation, it should be shown that the same things that hold for negative temperament

values also hold for positive values. The next tests made were designed to do just that.

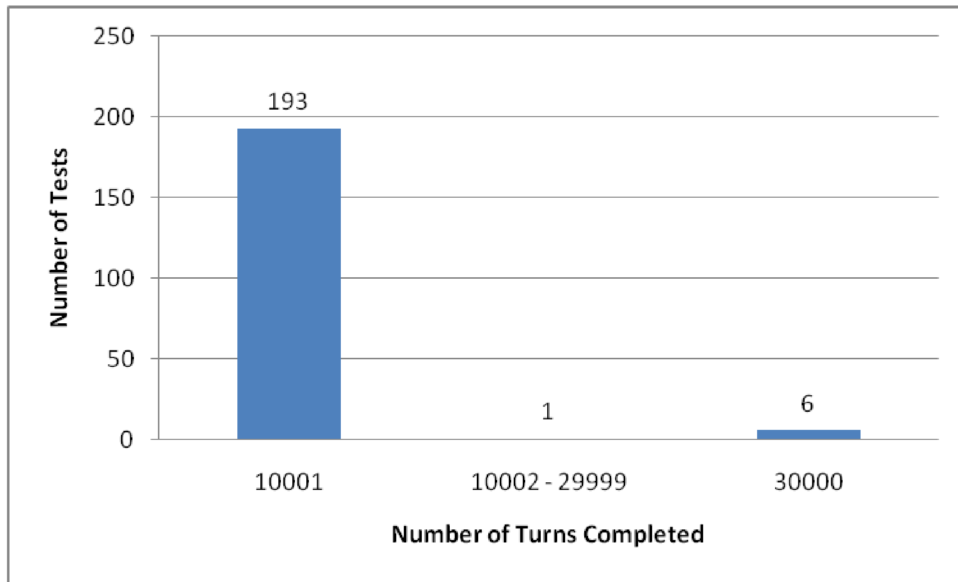

Figure 19 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, 0 Violence, and -50 Stealing

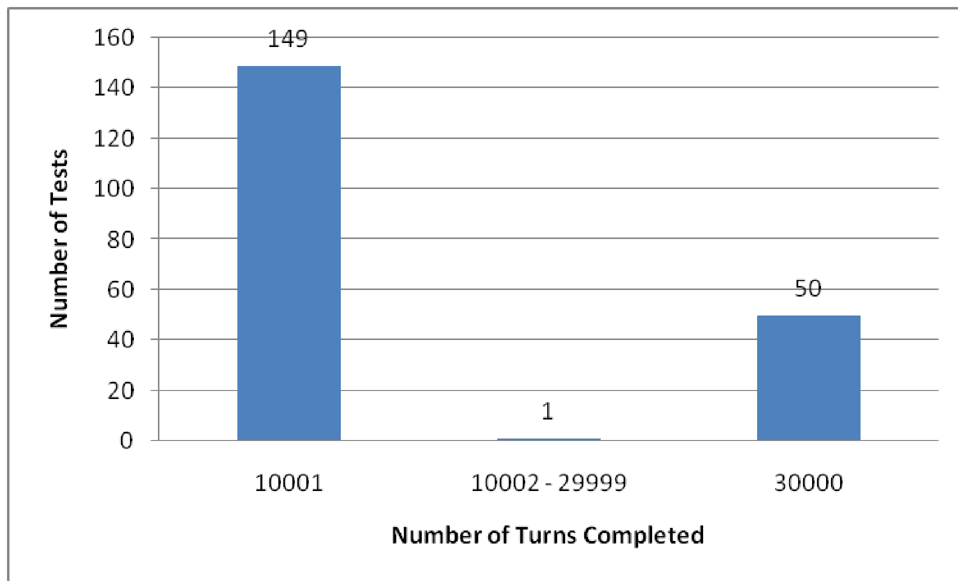

Figure 20 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, 0 Violence, and -25 Stealing

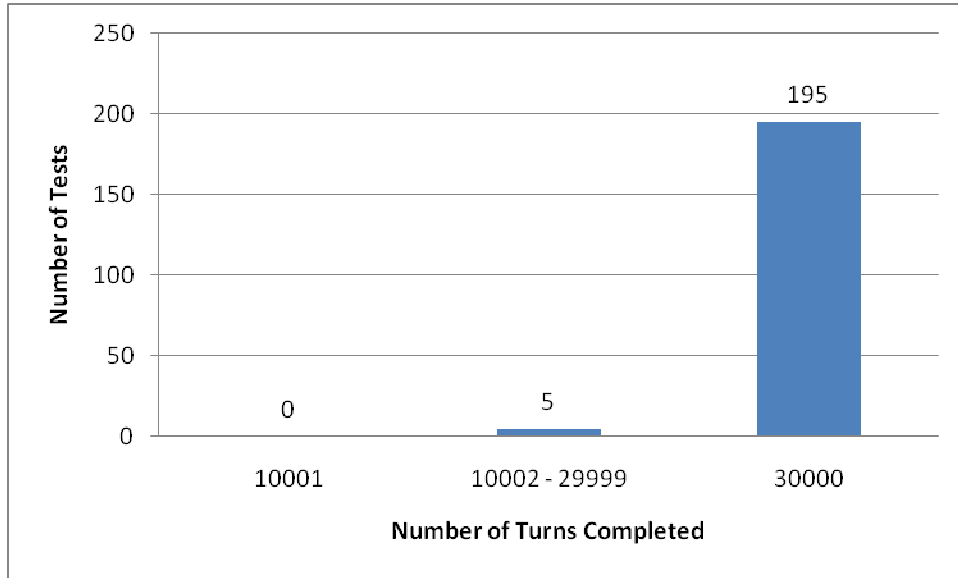

Figure 21 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, 0 Violence, and 25 Stealing

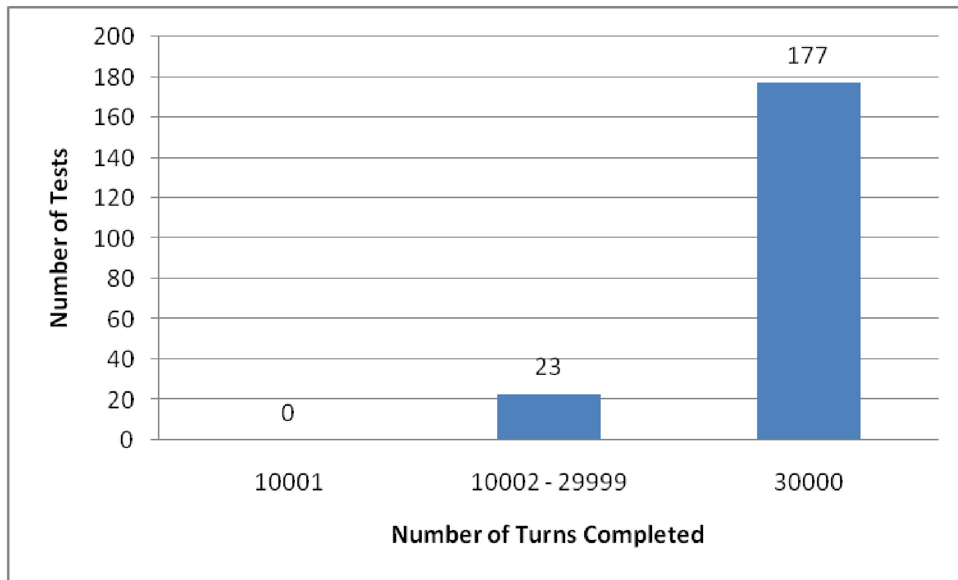

Figure 22 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, 0 Violence, and 50 Stealing

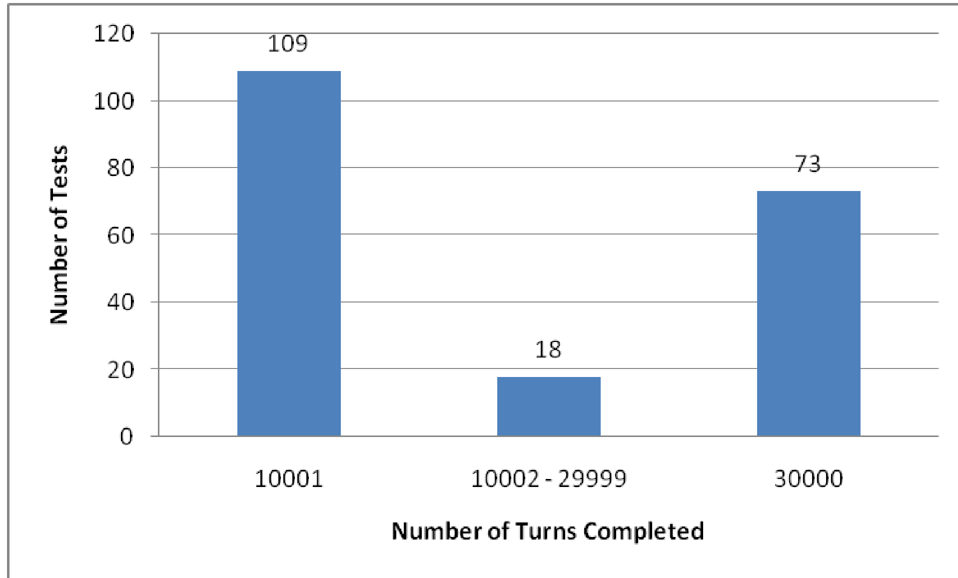

Figure 23 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, 0 Violence, and -50 Stealing

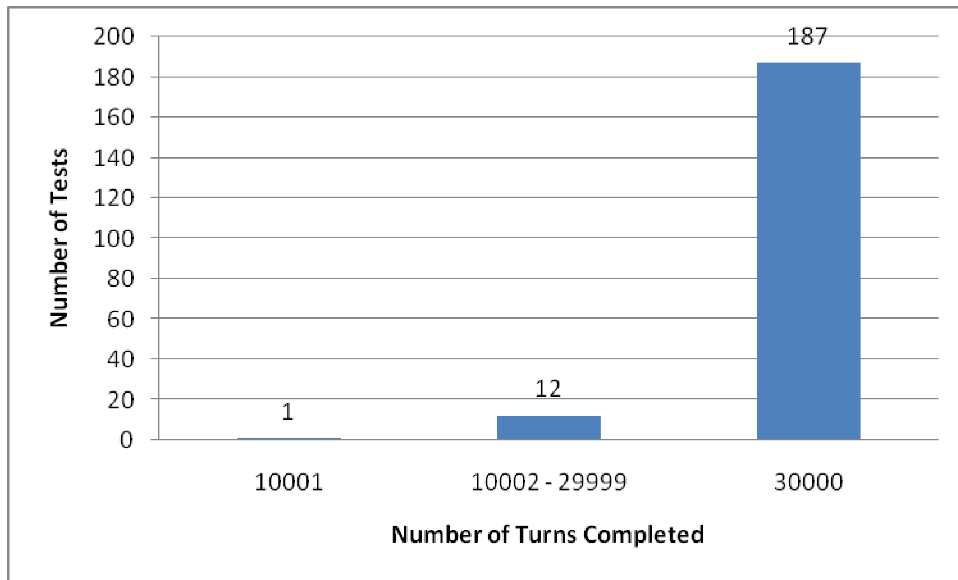

Figure 24 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, 0 Violence, and -25 Stealing

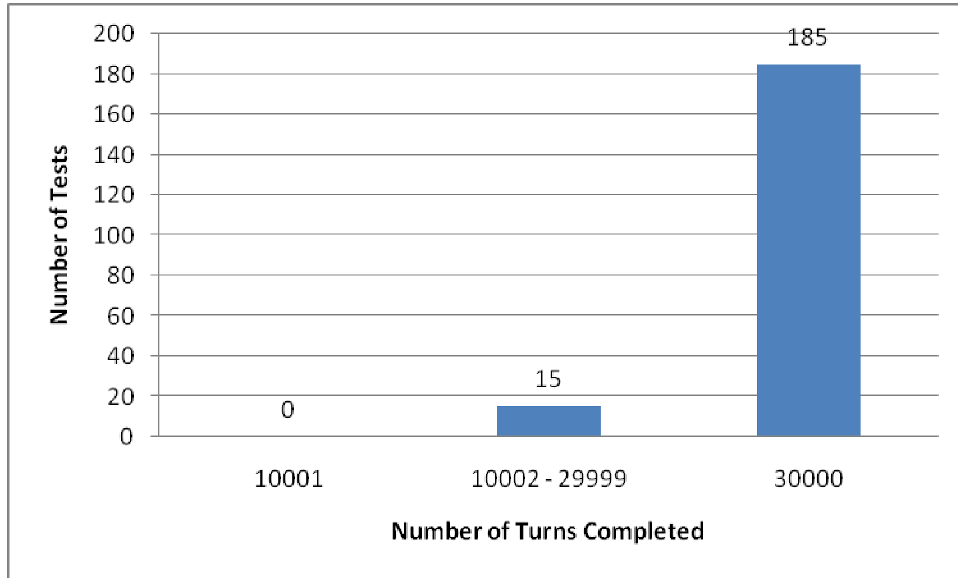

Figure 25 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, 0 Violence, and 25 Stealing

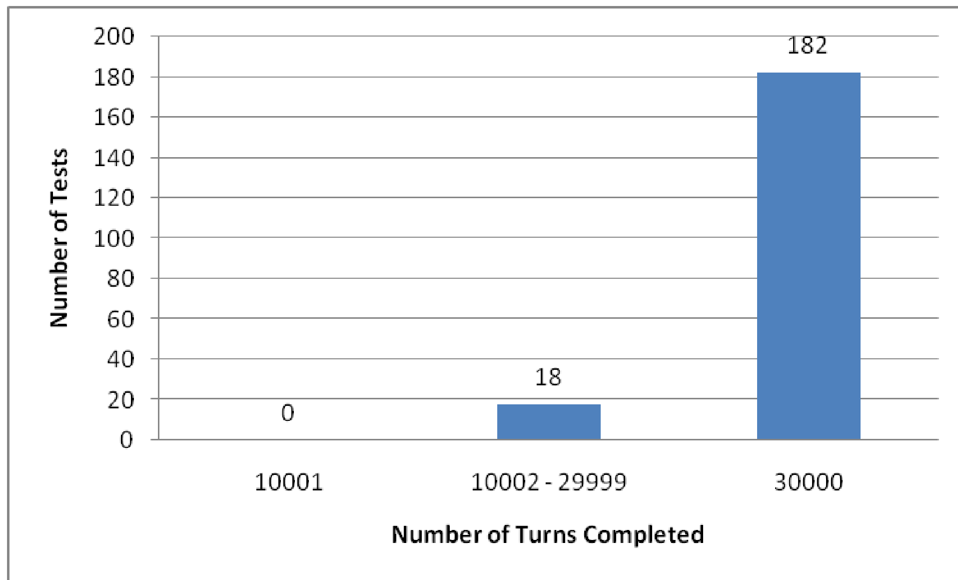

Figure 26 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, 0 Violence, and 50 Stealing

These results also show that the stealing norm has an effect on the longevity of the simulation. However, they are not nearly as convincing as the previous set. They do show that the convergent values increase the number of simulations reaching the positive extreme. On the other hand, the divergent values appear to only slightly affect the longevity of the simulations. For the negative temperament values, the application of positive stealing norms resulted in a large shift toward the positive extreme. In the opposite case, only little more than half of the results showed the same large shift. Regardless, it is still evident that the stealing norm can have a great effect on the longevity of the simulation.

Next, the violence norm was analyzed to see if it had the same effect on the longevity of the simulation as the stealing norm. These tests were performed in exactly the same manner as the tests done on the stealing norm, with the obvious exception of replacing the stealing norm with the violence norm. The tests were done with violence norm values of -50, -25, 0, 25, 50, and temperament values of -50, -25, 25, and 50.

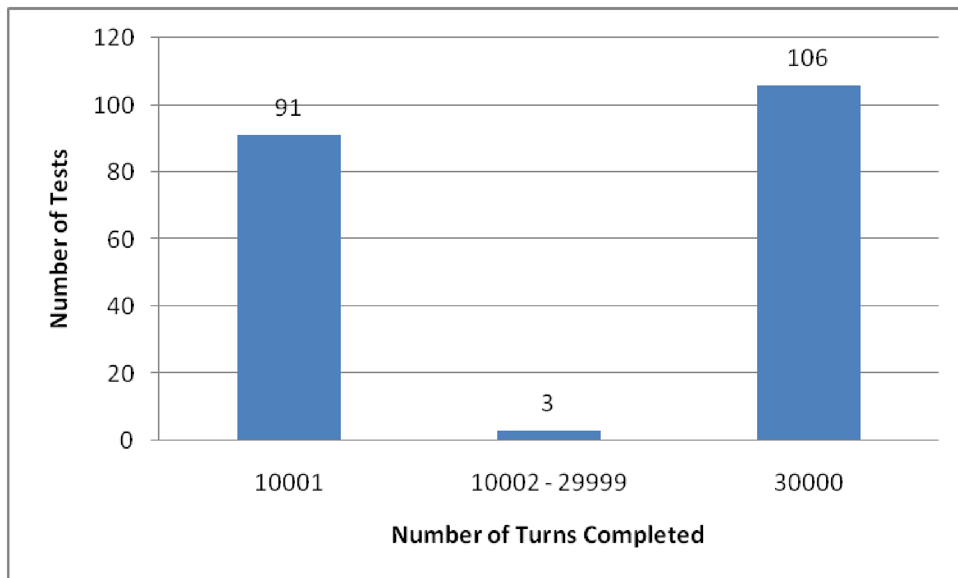

Figure 27 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, -50 Violence, and 0 Stealing

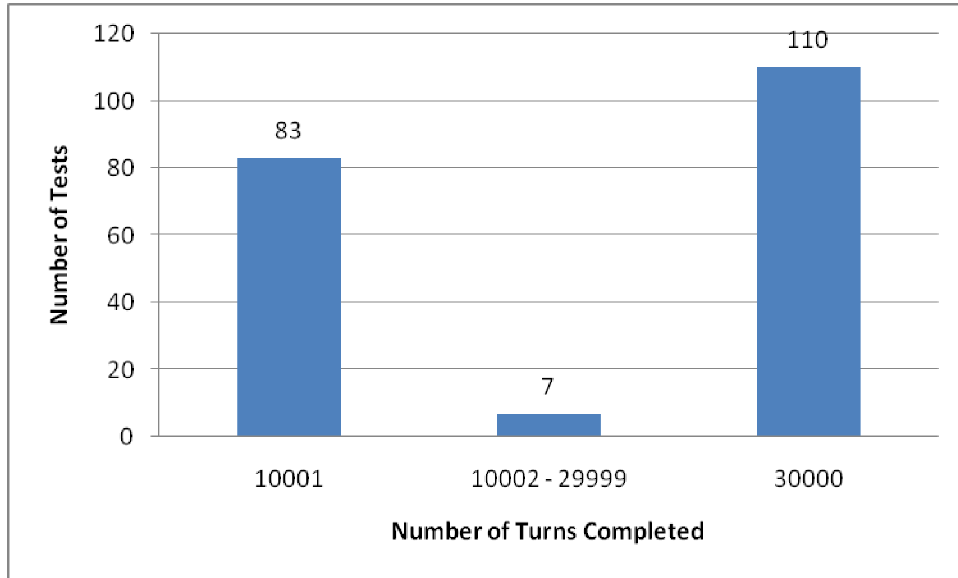

Figure 28 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, -25 Violence, and 0 Stealing

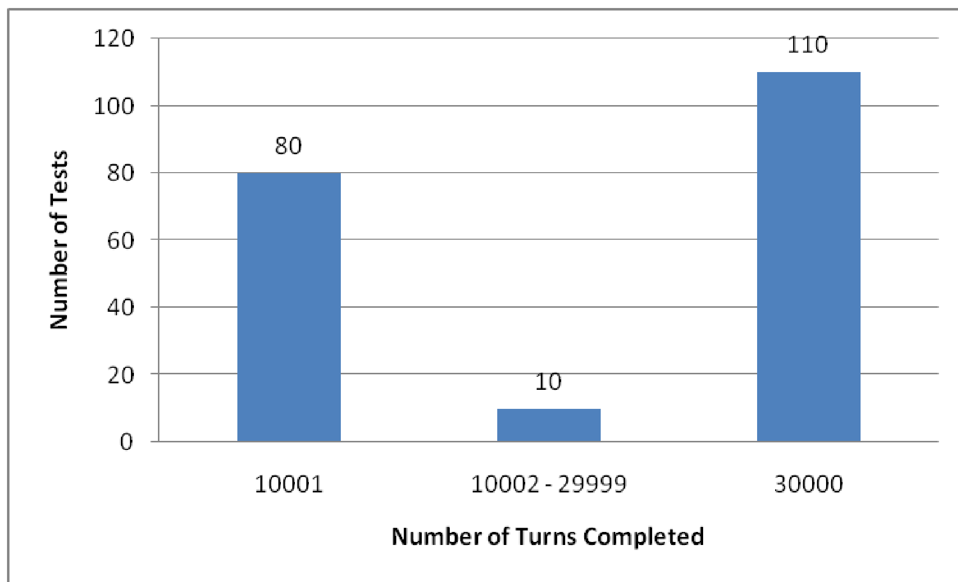

Figure 29 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 25 Violence, and 0 Stealing

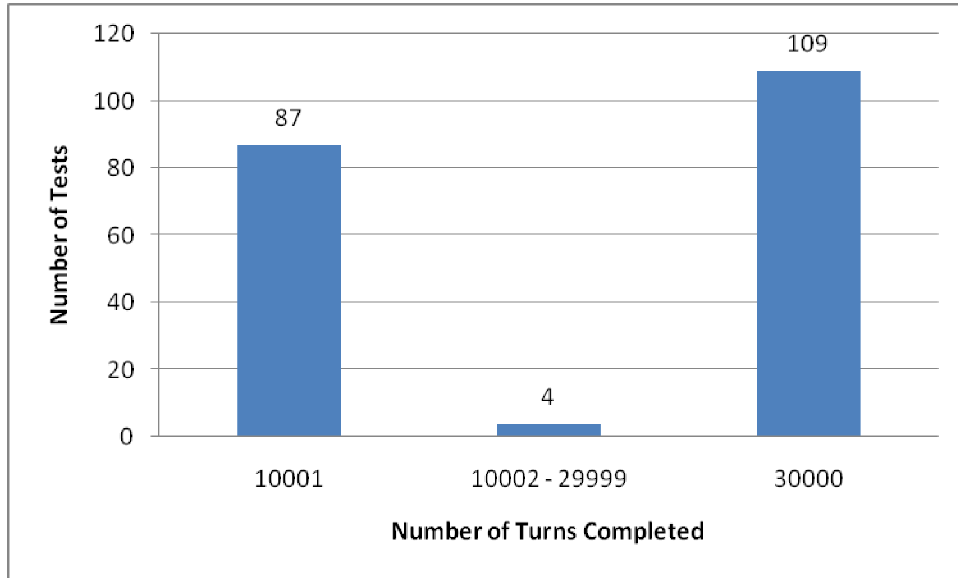

Figure 30 - Number of Tests that Completed a Certain Number of Turns using the settings 0 Temperament, 50 Violence, and 0 Stealing

This result is surprising because it shows that the violence norm has almost no effect on the longevity of the simulation. The original test done upon the totally pure simulation showed an almost identical distribution. This was unexpected since it was assumed that the violence norm would have the most effect on the simulation's longevity. The next tests were performed with the violence norm on the negative temperament values to see if the results showed the same strange trend.

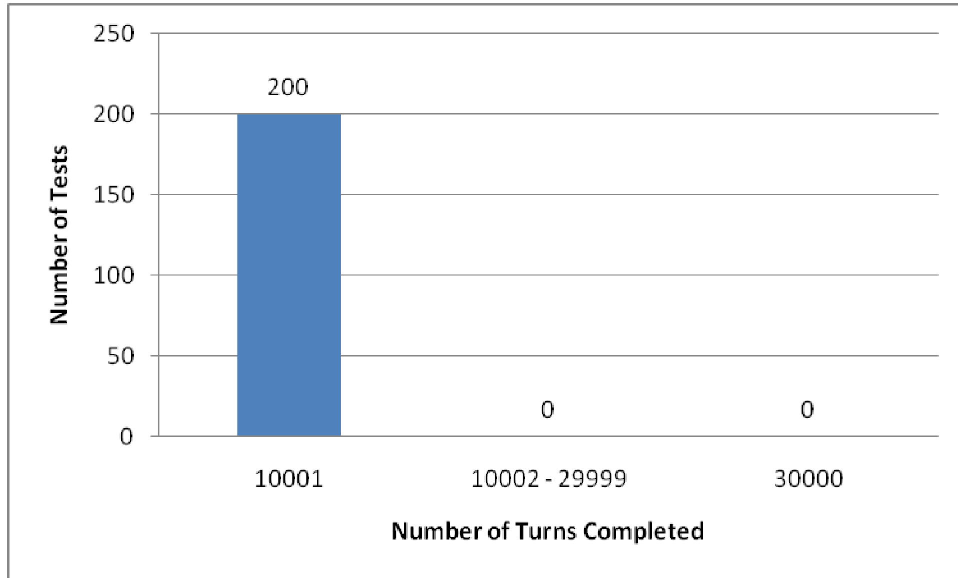

Figure 31 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, -50 Violence, and 0 Stealing

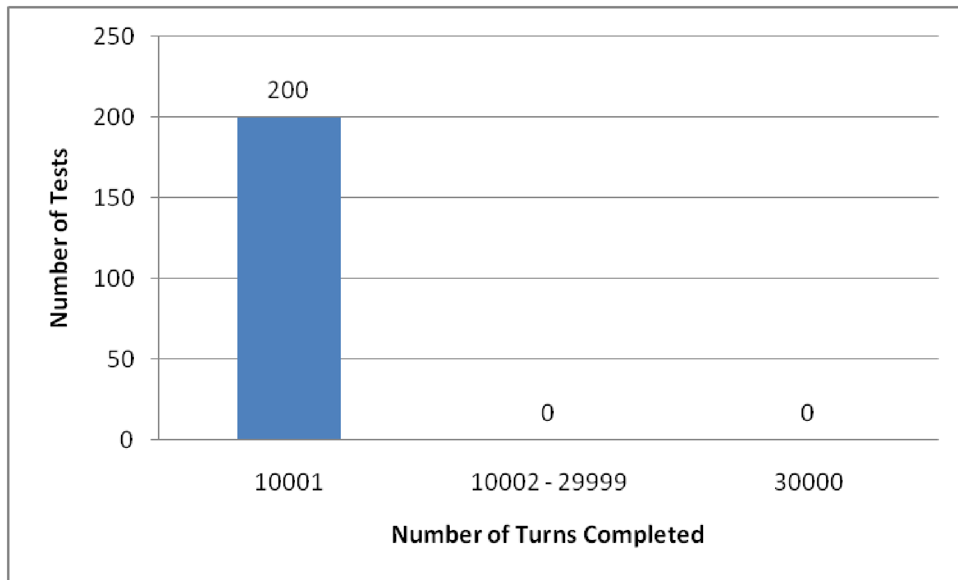

Figure 32 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, -25 Violence, and 0 Stealing

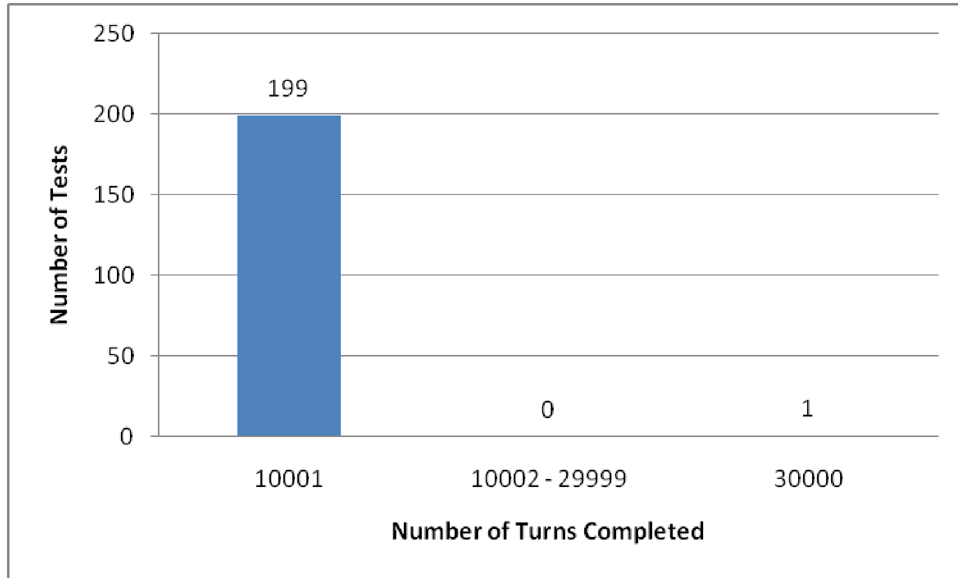

Figure 33 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 25 Violence, and 0 Stealing

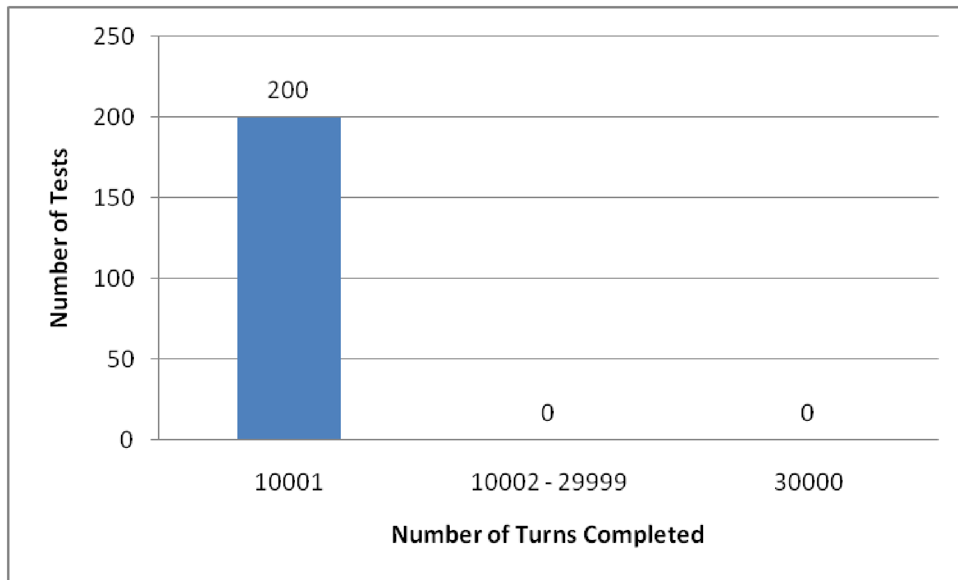

Figure 34 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 50 Violence, and 0 Stealing

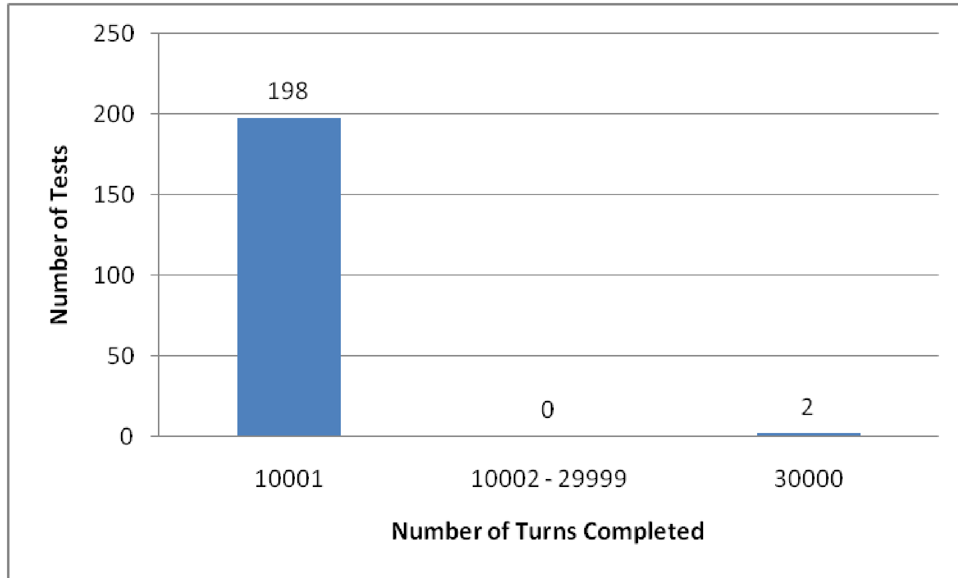

Figure 35 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, -50 Violence, and 0 Stealing

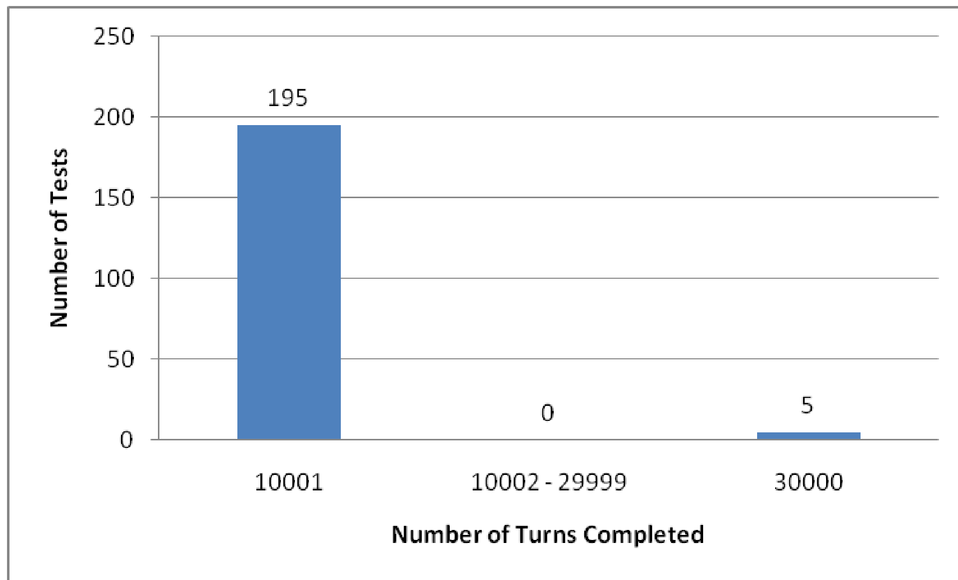

Figure 36 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, -25 Violence, and 0 Stealing

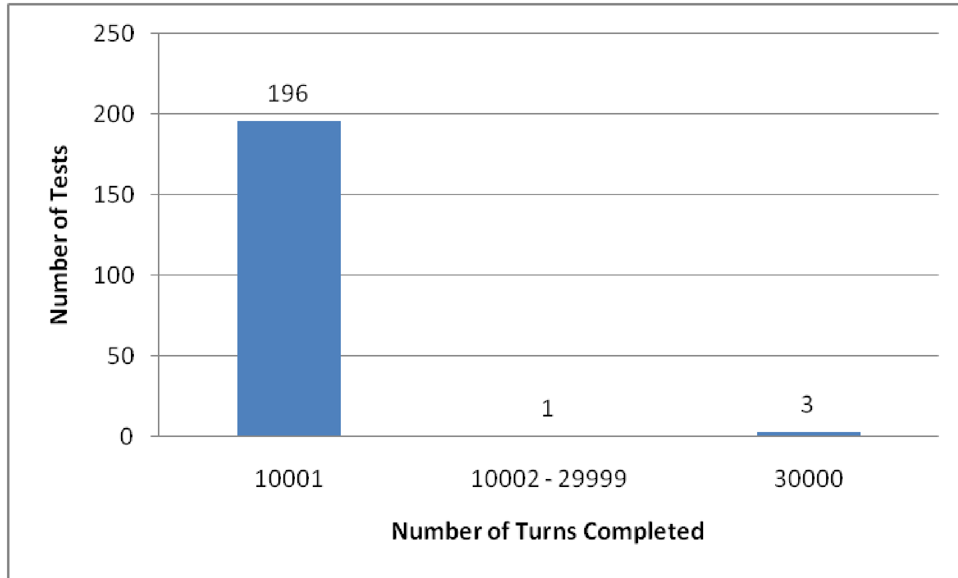

Figure 37 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 25 Violence, and 0 Stealing

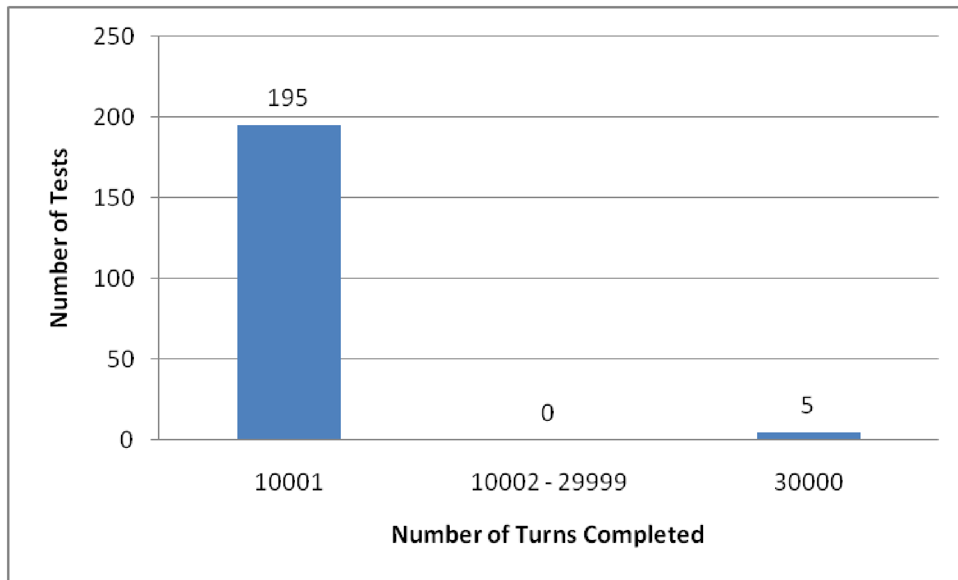

Figure 38 - Number of Tests that Completed a Certain Number of Turns using the settings -25 Temperament, 50 Violence, and 0 Stealing

This shows again that the violence norm has very little effect on the longevity of the simulation. Neither the convergent or divergent results show a great deal of change from the original results on the pure violence norm tests. The positive violence norm values were also tested to see if they too had little meaningful effect on the longevity of the simulation.

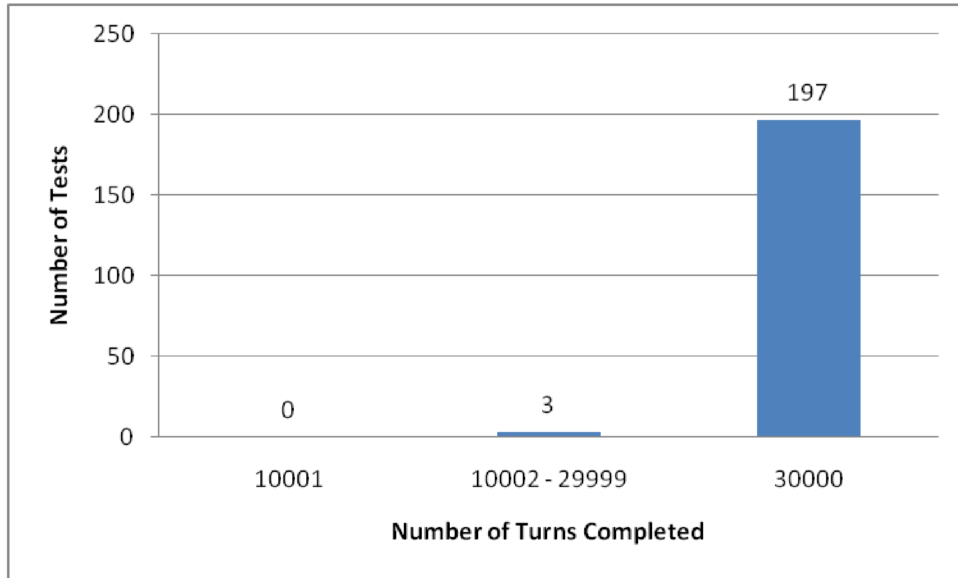

Figure 39 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, -50 Violence, and 0 Stealing

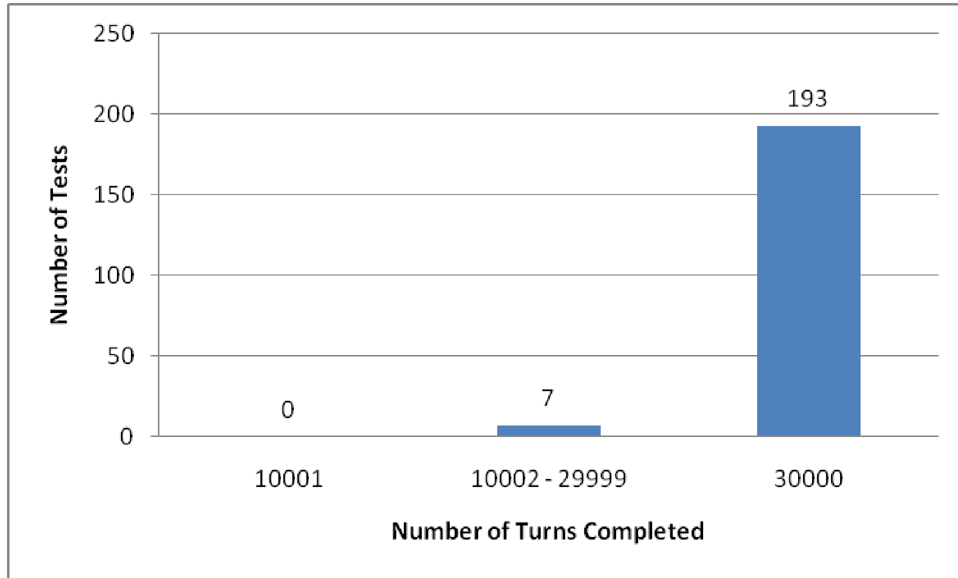

Figure 40 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, -25 Violence, and 0 Stealing

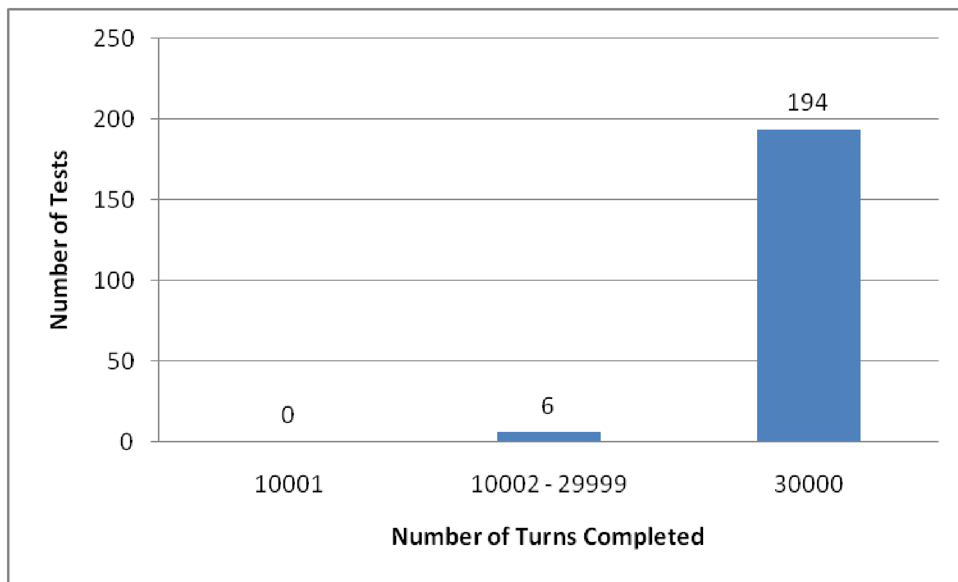

Figure 41 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, 25 Violence, and 0 Stealing

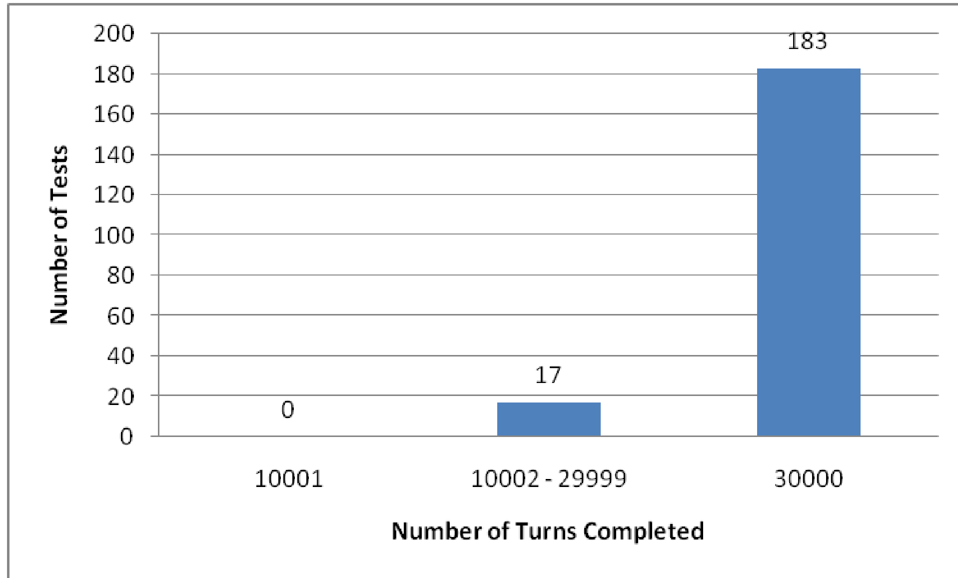

Figure 42 - Number of Tests that Completed a Certain Number of Turns using the settings 25 Temperament, 50 Violence, and 0 Stealing

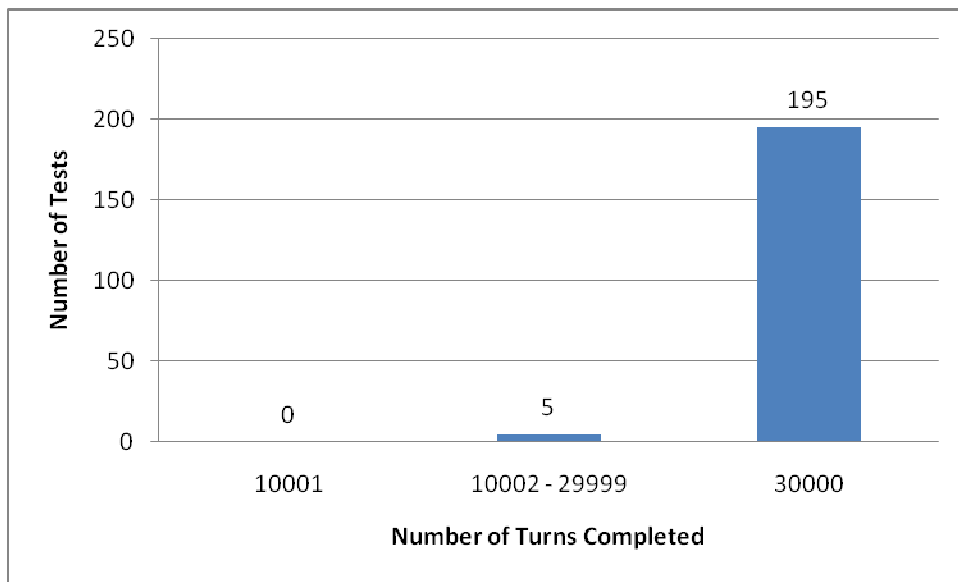

Figure 43 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, -50 Violence, and 0 Stealing

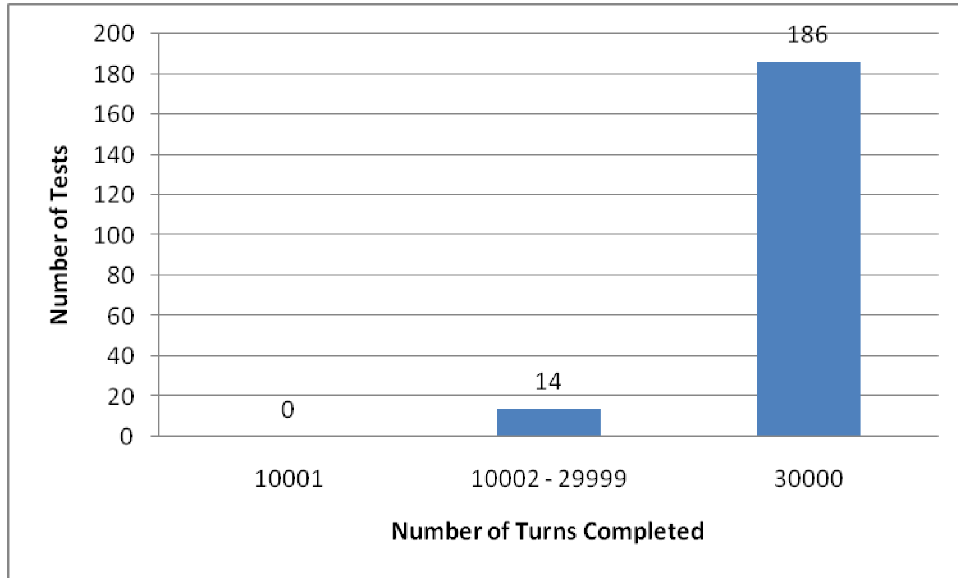

Figure 44 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, -25 Violence, and 0 Stealing

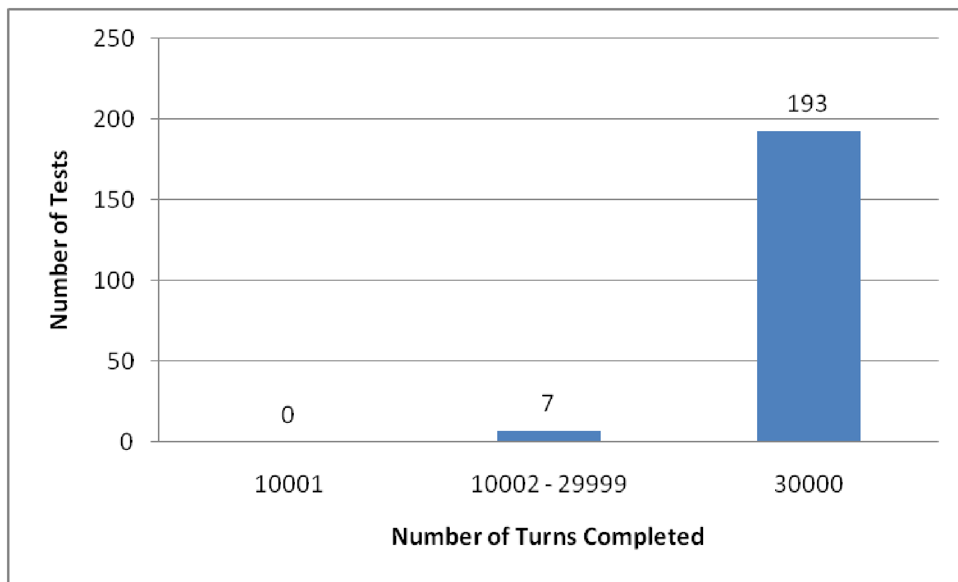

Figure 45 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, 25 Violence, and 0 Stealing

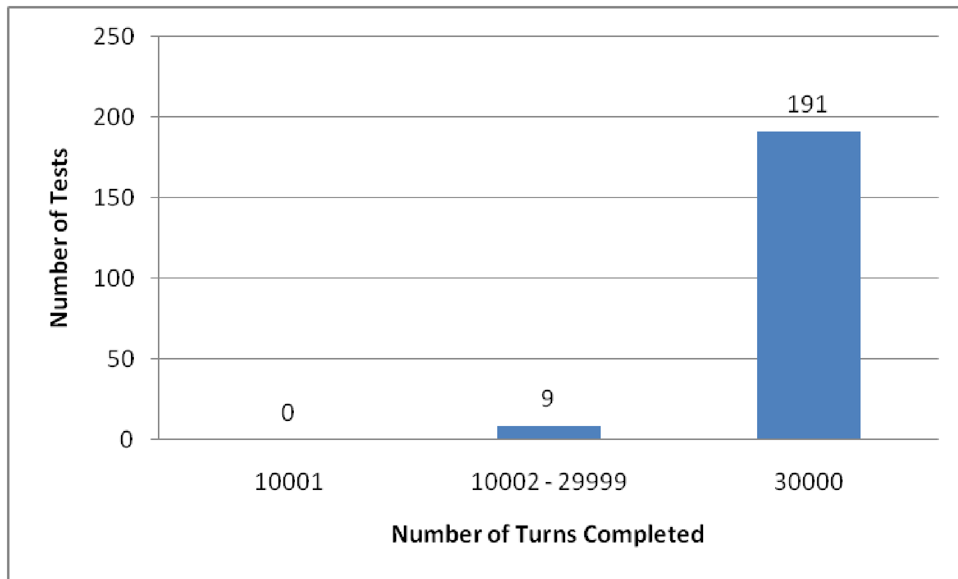

Figure 46 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, 50 Violence, and 0 Stealing

As with the negative values of the violence norm, the positive values have very little effect on the longevity of the simulation. The fact that the violence norm has such a small effect on the simulation, whereas the stealing norm has such a large effect, was counter to the ideas that went into designing the simulation. It was theorized when the simulation was created that a premium would be placed on violent behavior and that this behavior would be the guiding force in the simulation.

The final test run in this experiment was designed to see how well a society that operated normally on one extreme of the temperament value would operate in a norm system representing the opposite extremes in the norm values. To test this, the -50 temperament value was tested with a value of 50 for the stealing and violence norms. For completeness, the opposite case, a 50 temperament value, with a value of -50 for the stealing and violence norm, was also tested. These tests are the culmination of the entire project, and the results of these tests were those that were desired from the beginning.

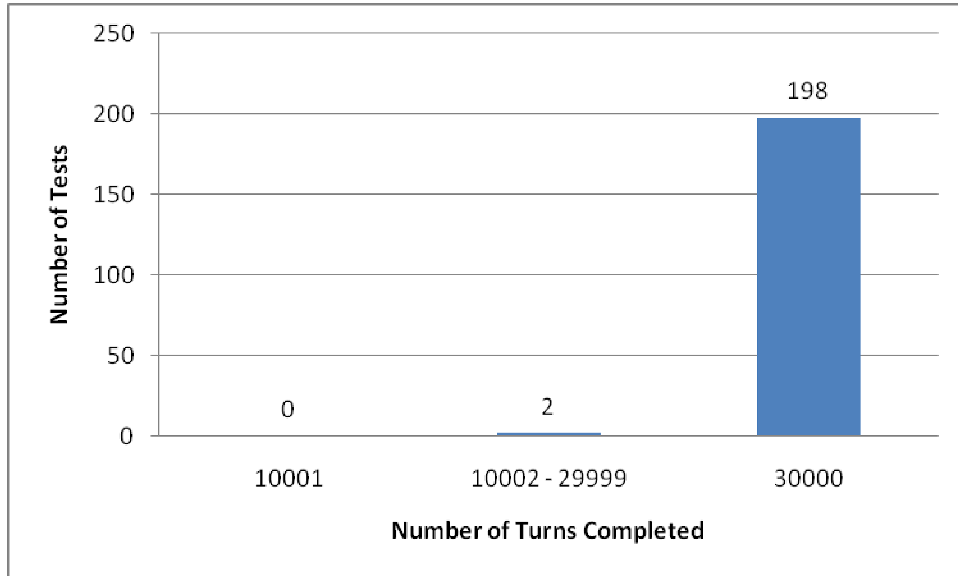

Figure 47 - Number of Tests that Completed a Certain Number of Turns using the settings -50 Temperament, 50 Violence, and 50 Stealing

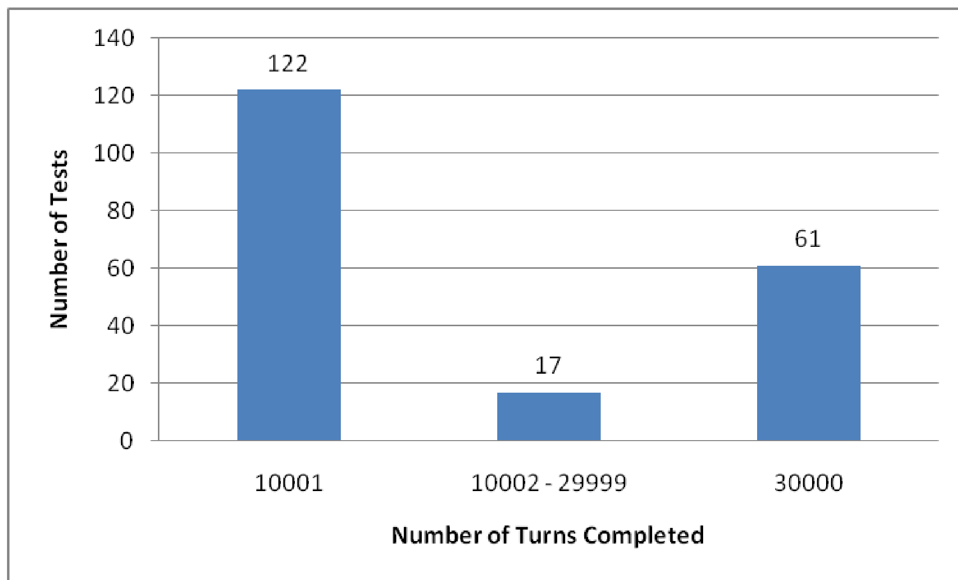

Figure 48 - Number of Tests that Completed a Certain Number of Turns using the settings 50 Temperament, -50 Violence, and -50 Stealing

Both results are concurrent with the other tests that were performed previously. In a small way it shows that this simulation was programmed to act in a very lifelike fashion. It could be said that societies that are inherently violent are more likely to be convinced to act in a peaceful manner than an inherently peaceful society would be to act violently. In these cases, human nature plays the most important role, particularly in the fact that humans will adapt to very difficult circumstances.

The tests repeated in this section show conclusively that norms, as they were developed for this simulation, can be used to control the overall behavior of the units in the simulation, in a deterministic way. The tests showed initially that when the simulation was run with all starting parameters initialized to zero, the number of test cases that reached the 30,000-turn limit was nearly equal to the number that did not. Also shown was the fact that simulations that made it to the 30,000-turn limit typically had higher birth rates and lower murder rates than simulations that did not. Because the tests were run with a wide range of stating parameters, it can be said that this experiment has conclusively shown that the stealing norm, and to a lesser extent the violence norm, has an effect on the probability of a specific run of the simulation reaching the 30,000-turn limit. Thus it can be assumed, since there is a direct correlation between types of behavior and the 30,000-turn limit, that the norms also have an effect on behavior.

CHAPTER V

CONCLUSION AND RECOMMENDATIONS

This experiment should be considered a partial success. While it failed to show that all societies could be forced by a controlling norm system to run until they reached a certain turn limit, it did show that norms are effective in manipulating the way societies work. This experiment also showed that it is not required for the norm systems to be vast and complex. Norm systems are useful outside of the context of the Prisoner's Dilemma, and relatively simple calculations can produce relatively complex and realistic results.

Moreover, because these calculations can be accomplished so quickly, it is feasible to include a small simulation, which includes a simple norm system, into complex First Person Shooter style games or other non-simulation style games.

In terms of the results that were gained, the fact that the stealing norm had such a large impact on the simulation was particularly surprising. Almost as surprising was the fact that the violence norm had little effect. These results can be explained by analyzing the way that certain events trigger changes in the norm system and the entities themselves. For example, as the simulation reaches a point where food becomes scarce, having a positive stealing norm induces entities to donate food to others. These donations improve inter-entity relations in the simulation and thus improve the chance that two entities will successfully mate. Thus the increase in good behavior leads to an increase in mating, thereby making the simulation more resistant to rogue entities going around killing everyone, or other disasters. One might ask that why, if the good vibes being passed around by the stealing norm are increasing mating, and thus enhancing the possibility of the simulation reaching the turn limit, then why doesn't the violence norm do the

same thing? One could theorize that the answer to that lies in the fact that the stealing norm is directly tied to resource acquisition. This resource acquisition is primary to an entities survival and represents a basic need of the entity. So in the case where food is lacking, the entity will be forced into performing an action that deals with the stealing norm because of its basic need for resources. The positive, or negative, influence of the stealing norm value is secondary, whereas with the violence norm, actions performed dealing with this norm are only done after an entity has resolved any resource issues that it may have. Thus the actions of the entity dealing with the violence norm are not as likely to occur; therefore, they will not have nearly as much effect.

The results of the tests of how societies react to being placed in a norm system that acts counter to the societies' inherent beliefs were also very pleasing. This was a test that had a bit of personal interest in it. These tests are important in terms of using this simulation to represent a neutral third party in a first-person-shooter-based war game. The experiment found that a simulation with good inhabitants could indeed survive in a world where the norm system indirectly "pressured" them to behave badly. While not all the test simulations reached the turn limit, with some alteration of the rules governing the simulation, all simulations could be made to reach the turn limit.

If the representation of the simulation is applied in the game world, then the actions being performed look realistic. To prove this, assume that one turn in the game world represents the time it takes an entity to move 5 feet. This is a convenient measurement since if it is also assumed that one game turn represents one second in game time, then that means that the entity moves at approximately 3.5 miles per hour. Coincidentally, this is precisely the average walking speed for a human[2]. In actuality, using the one-turn-per-second measurement may be a gross overestimate, since the most hardware-intensive settings of the simulation still produce several

[2] http://www.runeed.com/health/228-2-health-4.html

turns per second. However, it would be prudent to assume that any modern video game would employ graphical content vastly more complex than what was used in the simulation. This, combined with the fact that there would also be many other resource demands in a modern video game, suggests that the one-turn-per-second may be accurate. If one chooses to use the one-turn-per-second measurement, then 30,000 seconds, or approximately 8.3 hours, is equivalent to 30,000 turns. This is far longer than most people will play a video game in one sitting. The fact that the 10,000-turn maximum lifetime of an entity would most likely have to be increased greatly, probably to a near infinite value to make things seem realistic, adds further credence to the notion that this simulation could be used to represent a realistic gameplay environment.

The results found in this experiment have implications in other fields, particularly those that involve swarm intelligence [18]. Swarm intelligence is a discipline in the field of artificial intelligence that deals with swarms, groups of decentralized, self-organized, autonomous agents, and the collective intelligent behavior that such swarms exhibit. Typically the agents are very simple in design and are programmed to interact with other agents in a group setting. The agents are also designed to follow a set of simple rules, the caveat to this being that the rules must be self-imposed as there is no centralized control structure to speak of. Despite the lack of central control determining how the agents behave, the inter-agent interaction usually leads to the emergence of complex behavior in the swarm. Some examples of swarm intelligence in the natural world include colonies of ants, flocks of birds, packs of wolves, or schools of fish. In implementing swarms, typically a group of agents are placed in a world. This world represents a search space, or the domain of the problem. The search space is objectified into a multi-dimensional representation of the problem, and the agents occupy a specific position in the space. The position that the agent holds in the search space represents variables that equate to a

solution for the problem the swarm is trying to solve. These agents also, typically, have a fitness value associated with the accuracy of their solution to the problem. How the agents in the swarm behave is up to the designer and can be tailored to the problem being analyzed. Typically the agents' behavior will be implemented in such a way that solutions with higher fitness will dominate the swarm. In this way, through cooperation, the agents in the swarm will develop an accurate solution to the given problem.

In order to describe how this experiment relates to swarm intelligence, it is necessary to completely drop most of the trappings of the simulation that was created. The concepts of stealing, violence, and agents acting like humans must be left behind. Focusing solely on the norms and the entities, the corresponding connection is very clear. Suppose in some scenario, a swarm of agents is created to solve a problem. The agents apply a brand of un-normed knowledge and, for some reason, continually produce a result that is infeasible. To prevent these agents from continually repeating this mistake, a human overseer, or perhaps even an artificial one, could inform these agents, via a norm, that they should be less inclined to explore this option. Conversely, if a solution is thought to be in one area of a search space, the agents could be induced, again via a norm, to actively comb this area of the search space for a solution. However, in applying a system of norms, it could be said that one of the basic tenets of swarm intelligence, the lack of a centralized control structure, has been violated. So an attempt to apply norms to swarm intelligence would actually be creating a hybrid system rather than enhancing the study of swarm intelligence.

To conclude, the behavior of human-like agents can be controlled using a norm system, and under certain conditions the norm system can control the longevity of a simulation that mimics some aspects of human society. Therefore, norms can be used to cause a system of

independent agents to exhibit communal behavior. This experiment also showed that it would not be unrealistic to use a similar simulation in an FPS-style video game. This simulation could be used to represent a group of third-party agents designed to interact with the player and with other agents. Because of these reasons, it is evident that further study in this field is warranted.

REFERENCES

[1] Chaplin, D. J. and Rhalibi, A. E. 2004. IPD for emotional NPC societies in games. In *Proceedings of the 2004 ACM SIGCHI international Conference on Advances in Computer Entertainment Technology* (Singapore, June 03 - 05, 2005). ACE '04, vol. 74. ACM, New York, NY, 51-60.

[2] Axelrod, R, "An Evolutionary Approach to Norms", The Complexity of Cooperation, Robert Axelrod, p 44-68,Princeton University Press, Chichester,England

[3] Lester, P. (2005.) "A* Pathfinding for Beginners." policyalmanac.org. 17 July 2008. http://www.policyalmanac.org/games/aStarTutorial.htm

[4] Will Wright (Maxis Software), Ken Forbus (Northwestern University) 2001, "Under the hood of The Sims", Presentation of The Sims at Northwestern University. (03/03). http://www.cs.northwestern.edu/~forbus/c95-gd/lectures/The_Sims_Under_the_Hood.htm

[5] Bates, J., Loyall, A. B., and Reilly, W. S. 1998. An architecture for action, emotion, and social behavior. In *Readings in Agents*, M. N. Huhns and M. P. Singh, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 225-231.

[6] Davis, D.N. (2000). Minds have personalities - Emotion is the core. In Symposium on How to Design a Functioning Mind, AISB2000, University of Birmingham.

[7] El Rhalibi, A., Baker, N., and Merabti, M. 2005. Emotional agent model and architecture for NPCs group control and interaction to facilitate leadership roles in computer entertainment. In *Proceedings of the 2005 ACM SIGCHI international Conference on Advances in Computer Entertainment Technology* (Valencia, Spain, June 15 - 17, 2005). ACE '05, vol. 265. ACM, New York, NY, 156-163.

[8] Survival: The Last Laugh. Chapter 1- The Beginning. (1998). In *Survival.com*. Retrieved April 29, 2008, from http://www.survival.com/bookch1a.htm

[9] Pac-Man. (2006). In *The Dot Eaters*. Retrieved April 29, 2008, from http://www.thedoteaters.com/p2_stage4.php

[10] World of Warcraft. (2008, April 28). In *Wikipedia, the free encyclopedia*. Retrieved April 29, 2008, from http://en.wikipedia.org/wiki/World_of_Warcraft

[11] Brothers in Arms: Road to Hill 30. (2008, March 29). In *Wikipedia, the free encyclopedia*. Retrieved April 29, 2008, from http://en.wikipedia.org/wiki/Brothers_in_Arms:_Road_to_Hill_30

[12] Grand Theft Auto series. (2008, April 29). In *Wikipedia, the free encyclopedia*. Retrieved April 29, 2008, from http://en.wikipedia.org/wiki/Grand_Theft_Auto_(series)

[13] The Legend of Zelda: Twilight Princess. (2008, April 28). In *Wikipedia, the free encyclopedia*. Retrieved April 29, 2008, from http://en.wikipedia.org/wiki/The_Legend_of_Zelda:_Twilight_Princess

[14] Fable (video game). (2008, April 28). In *Wikipedia, the free encyclopedia*. Retrieved April 29, 2008, from http://en.wikipedia.org/wiki/Fable_(video_game)

[15] Online Dictionary of Social Sciences. (2008, April 17). In *The Online Dictionary of Social Sciences*. Retrieved April 29, 2008, from http://bitbucket.icaap.org/dict.pl?term=NORMS

[16] Busse, C. and Hamilton, W.J. 1981. Infant Carrying by Male Chacma Baboons. In *Science: New Series, June 12, 1981.* Science, vol. 212. AAAS, Washington D.C., 1281-1283.

[17] Hart, P.E., Nilsson, N.J., and Raphael, B. 1972. Correction to "A formal basis for the heuristic determination of minimum cost paths." *SIGART Newsletter*, no. 37, December, pp. 28-29.

[18] Hinchey, Michael G.; Sterritt, Roy; Rouff, Chris, "Swarms and Swarm Intelligence," *Computer* , vol.40, no.4, pp.111-113, April 2007.

APPENDIX A

PROGRAM SOURCE CODE

DrawingFrame.java

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class DrawingFrame extends JFrame implements ActionListener {
   Simulation s = new Simulation();
   DrawingPanel mapPanel = new DrawingPanel();
   private long speed;
   private boolean go;
   private boolean step;
   private boolean done;

   //info panel stuff
   JPanel infoPanel = new JPanel();
   JTextArea foodText = new JTextArea(1,5);
   JTextArea waterText = new JTextArea(1,5);
   JTextArea turnText = new JTextArea(1,5);
   JTextArea speedText = new JTextArea(1,5);
   JTextArea popText = new JTextArea(1,5);
   JTextArea sText = new JTextArea(1,5);
   JTextArea vText = new JTextArea(1,5);

   //log panel stuff
   JScrollPane logPanel = new JScrollPane();
   JTextArea logText = new JTextArea(5,30);

   //toolbar stuff
   JToolBar toolBar = new JToolBar();

   //panel for the info and the toolbar
   JPanel eastPanel = new JPanel();

   static final private String SPEEDUP = "Speed Up";
   static final private String SPEEDDOWN = "Slow Down";
   static final private String START = "Start";
   static final private String STOP = "Stop";
   static final private String STEP = "Step Forward";
   static final private String INCFOOD = "Add food";
   static final private String DECFOOD = "Remove food";
   static final private String INCWATER = "Add Water";
   static final private String DECWATER = "Remove Water";
   static final private String QUIT = "Quit";

   public DrawingFrame(Simulation sim) {
      go = true;
      step = false;
```

```
      speed = 100;
      done = false;
      s = sim;
      try {
         myInit();
      }catch(Exception ex) {
         ex.printStackTrace();
      }
   }

   void myInit() throws Exception {

      this.setResizable(false);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.setTitle("Thesis");
      this.getContentPane().setLayout(new BorderLayout());

      mapPanel.setSimulation(s);
      mapPanel.setDoubleBuffered(true);
      mapPanel.setPreferredSize(new Dimension(605,605));
      mapPanel.setBackground(Color.BLACK);

      infoPanel.setPreferredSize(new Dimension(150,150));
      infoPanel.setLayout(new GridLayout(7,2,1,1));
      infoPanel.add(new JLabel("Food: "));
      infoPanel.add(foodText);
      infoPanel.add(new JLabel("Water: "));
      infoPanel.add(waterText);
      infoPanel.add(new JLabel("Turn: "));
      infoPanel.add(turnText);
      infoPanel.add(new JLabel("Speed: "));
      infoPanel.add(speedText);
      infoPanel.add(new JLabel("Population: "));
      infoPanel.add(popText);
      infoPanel.add(new JLabel("Stealing: "));
      infoPanel.add(sText);
      infoPanel.add(new JLabel("Violence: "));
      infoPanel.add(vText);
      foodText.setEditable(false);
      waterText.setEditable(false);
      turnText.setEditable(false);
      speedText.setEditable(false);
      popText.setEditable(false);
      sText.setEditable(false);
      vText.setEditable(false);

      foodText.setText(Integer.toString(s.getBoxQuantity(0)));
      waterText.setText(Integer.toString(s.getBoxQuantity(1)));
      turnText.setText(Integer.toString(s.getTurn()));
      speedText.setText(Integer.toString((int)speed));
      popText.setText(Integer.toString(s.getPopulation()));
      sText.setText(Integer.toString(s.getNorms().getS()));
      vText.setText(Integer.toString(s.getNorms().getV()));
      toolBar.setPreferredSize(new Dimension(150,455));
      toolBar.setBorderPainted(false);
      toolBar.setFloatable(false);
      toolBar.setLayout(new GridLayout(11,0,0,0));
```

```
   addButtons(toolBar);

   logText.setEditable(false);
   logText.setText("Information will be displayed here");
   logText.setTabSize(8);
   logPanel.getViewport().add(logText, null);

   eastPanel.setPreferredSize(new Dimension(150,605));
   eastPanel.add(infoPanel);
   eastPanel.add(toolBar);

   this.getContentPane().add(mapPanel, BorderLayout.CENTER);
   this.getContentPane().add(eastPanel, BorderLayout.EAST);
   this.getContentPane().add(logPanel, BorderLayout.SOUTH);

   this.pack();
   this.setVisible(true);
}

public void setSimulation(Simulation s1) {
   mapPanel.setSimulation(s1);
}

public void repaintStuff() {
   mapPanel.repaint();
   foodText.setText(Integer.toString(s.getBoxQuantity(0)));
   waterText.setText(Integer.toString(s.getBoxQuantity(1)));
   turnText.setText(Integer.toString(s.getTurn()));
   logText.setText(s.getLog());
   speedText.setText(Integer.toString((int)speed));
   popText.setText(Integer.toString(s.getPopulation()));
   sText.setText(Integer.toString(s.getNorms().getS()));
   vText.setText(Integer.toString(s.getNorms().getV()));
}

protected void addButtons(JToolBar toolBar) {
   //Add all the tool bar buttons
   JButton button = null;
   button = makeNavigationButton(SPEEDUP, SPEEDUP, SPEEDUP);
   toolBar.add(button);
   button = makeNavigationButton(SPEEDDOWN, SPEEDDOWN, SPEEDDOWN);
   toolBar.add(button);
   button = makeNavigationButton(START, START, START);
   toolBar.add(button);
   button = makeNavigationButton(STOP, STOP, STOP);
   toolBar.add(button);
   button = makeNavigationButton(STEP, STEP, STEP);
   toolBar.add(button);
   toolBar.add(new JLabel(""));
   button = makeNavigationButton(INCFOOD, INCFOOD, INCFOOD);
   toolBar.add(button);
   button = makeNavigationButton(DECFOOD, DECFOOD, DECFOOD);
   toolBar.add(button);
   button = makeNavigationButton(INCWATER, INCWATER, INCWATER);
   toolBar.add(button);
   button = makeNavigationButton(DECWATER, DECWATER, DECWATER);
   toolBar.add(button);
```

```
      button = makeNavigationButton(QUIT, QUIT, QUIT);
      toolBar.add(button);
   }

   protected JButton makeNavigationButton(String actionCommand, String
toolTipText, String altText) {
      //Create and initialize the button.
      JButton button = new JButton();
      button.setActionCommand(actionCommand);
      button.setToolTipText(toolTipText);
      button.addActionListener(this);
      button.setText(altText);
      return button;
   }

   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      // Handle each button.
      if (SPEEDUP.equals(cmd)) {
         speed -= 25;
         if(speed < 0) {
            speed = 0;
         }
      }else if(SPEEDDOWN.equals(cmd)) {
         speed += 25;
         if(speed > 1000) {
            speed = 1000;
         }
      }else if(START.equals(cmd)) {
         go = true;
      }else if(STOP.equals(cmd)) {
         go = false;
      }else if(STEP.equals(cmd)) {
         if(!go) {
            step = true;
         }
      }else if(INCFOOD.equals(cmd)) {
         s.editBoxQuantity(0, true);
      }else if(DECFOOD.equals(cmd)) {
         s.editBoxQuantity(0, false);
      }else if(INCWATER.equals(cmd)) {
         s.editBoxQuantity(1, true);
      }else if(DECWATER.equals(cmd)) {
         s.editBoxQuantity(1, false);
      }else if(QUIT.equals(cmd)) {
         done = true;
      }
   }

   long getspeed() {
      return speed;
   }

   boolean going() {
      return go;
   }
```

```
   boolean done() {
      return done;
   }

   boolean step() {
      return step;
   }

   void stepped() {
      step = false;
   }
}
```

DrawingPanel.java

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class DrawingPanel extends JPanel {
   Simulation s;
   public DrawingPanel () {
      s = new Simulation();
   }

   public void paintComponent(Graphics g)   {
      // Paint background
      super.paintComponent(g);

      // Get the drawing area
      //OBSOLETE
      //int[] xpos = s.getallxpos();
      //int[] ypos = s.getallypos();
      for(int i=0; i<s.getDenziensCount(); i++) {
         if(s.isEntityAlive(i)) {

drawUnit(g,i,s.getEntityXpos(i),s.getEntityYpos(i),s.getEntityWidth(i),
s.getEntityColor(i));
         }
      }
      drawBox(g, s.getBoxXpos(0), s.getBoxYpos(0), s.getBoxWidth(0), 0);
      drawBox(g, s.getBoxXpos(1), s.getBoxYpos(1), s.getBoxWidth(1), 1);

   } // paintComponent

   private void drawUnit(Graphics g, int i, int x, int y, int width, Color
thecolor) {
      g.setColor(thecolor);
      g.fillOval(x, y, width, width);
      g.setColor(Color.BLACK);
      g.drawString(Integer.toString(i), x+5, y+15);
   }

   private void drawBox(Graphics g, int x, int y, int width, int type) {
      String label = "";
```

```
      switch(type) {
         case 0:
            g.setColor(Color.GRAY);
            label = "F";
            break;
         case 1:
            g.setColor(Color.BLUE);
            label = "W";
            break;
      }
      g.fillRect(x, y, width, width);
      g.setColor(Color.BLACK);
      g.drawString(label, x+10, y+20);
   }

   public void setSimulation(Simulation s1) {
      s = s1;
   }

} // class DrawingPanel


import java.awt.Color;
import java.util.Random;
import java.util.ArrayList;

public class Entity {
   private int xpos;
   private int ypos;
   private int xvel;
   private int yvel;
   private int width;
   private int dob;
   private boolean alive;
   private Color mycolor;
   private int hunger;
   private int thirst;
   private int health;
   private int foodcount;
   private int watercount;
   private int minpos;
   private int maxpos;
   private int interactiontarget;
   private int[] positiontarget;
   private int action;
   //private int number;
   //0:   idle
   //1:   eat food
   //2:   drink water
   //3:   obtain food
   //4:   obtain water
   //5:   move randomly
   //6:   interact
   private int resourceaction;
   //0:   get food
   //1:   request food
```

```
    //2:   steal food
    private int interaction;
    //0:   talk
    //1:   hug
    //2:   slap
    //3:   kiss
    //4:   punch
    //5:   mate
    //6:   kill
    private int temperment;
    private ArrayList<int[]> opinions;
    private boolean gotTarget;

    public Entity(Random r, int max, int min, int turn, int grid, int num, int
 mt) {
       minpos = min;
       maxpos = max;
       xpos = minpos+r.nextInt(maxpos);
       ypos = minpos+r.nextInt(maxpos);
       xvel = 0;
       yvel = 0;
       health = 100;
       thirst = r.nextInt(100);
       hunger = r.nextInt(100);
       foodcount = 0;
       watercount = 0;
       width = grid;
       dob = turn;
       alive = true;
       gotTarget = false;
       temperment = -50+r.nextInt(101)+mt;
       if(temperment<-50) {
          temperment = -50;
       }
       if(temperment>50) {
          temperment = 50;
       }
       if(temperment<=-25) {
          mycolor = Color.red;

       }else if(temperment>=25) {
          mycolor = Color.green;
       }else{
          mycolor = Color.yellow;
       }
       action = 0;
       resourceaction = 0;
       positiontarget = new int[2];
       interactiontarget = -1;
       opinions = new ArrayList<int[]>();
    }

    void setxpos(int pos) {
       xpos = pos;
    }
    void setypos(int pos) {
       ypos = pos;
```

```
}
int getxpos() {
   return xpos;
}
int getypos() {
   return ypos;
}
int getwidth() {
   return width;
}

Color getColor() {
   return mycolor;
}

String move() {
   String temp = "";
   temp = " moved from " + xpos + ":" + ypos + " to ";
   xpos += xvel;
   ypos += yvel;
   temp += xpos + ":" + ypos;
   return temp;
}

int[] premove() {
   int[] temp = new int[2];
   temp[0] = xpos+xvel;
   temp[1] = ypos+yvel;
   return temp;
}

void incFood() {
   if(foodcount<3) {
      foodcount++;
   }
}

void incWater() {
   if(watercount<3) {
      watercount++;
   }
}

void decFood() {
   if(foodcount>0) {
      foodcount--;
   }
}

void decWater() {
   if(watercount>0) {
      watercount--;
   }
}

void updateVitals(int t, int m) {
   hunger++;
```

```
    thirst++;
    if(hunger>99) {
       health--;
    }
    if(thirst>99) {
       health -= 2;
    }
    if(health<=0) {
       alive = false;
    }
    if(t-dob>m) {
       alive = false;
    }
 }

 boolean isAlive() {
    return alive;
 }

 int getFoodCount() {
    return foodcount;
 }

 int getWaterCount() {
    return watercount;
 }

 void zeroVel() {
    xvel = 0;
    yvel = 0;
 }

 void reversevel() {
    xvel = -xvel;
    yvel = -yvel;
 }

 void setVel(int[] v) {
    xvel = v[0]-xpos;
    yvel = v[1]-ypos;
 }

 int getage(int turn) {
    return (turn-dob);
 }

 void chooseAction(Random r, int p) {
    if(hunger>49&&thirst<=49) {
       if(foodcount>0) {
          action = 1;
       }else{
          action = 3;
       }
    }else if(hunger<=49&&thirst>49) {
       if(watercount>0) {
          action = 2;
       }else{
```

```
         action = 4;
      }
   }else if(hunger>49&&thirst>49) {
      if((foodcount>0)&&(watercount==0)) {
         action = 1;
      }else if((foodcount==0)&&(watercount>0)) {
         action = 2;
      }else if((foodcount>0)&&(watercount>0)){
         if(hunger>thirst) {
            action = 1;
         }else{
            action = 2;
         }
      }else{
         if(hunger>thirst) {
            action = 3;
         }else{
            action = 4;
         }
      }
   }else{
      if(p>1) {
         action = 1+r.nextInt(6);
      }else{
         action = 1+r.nextInt(5);
      }
   }
}

int getAction() {
   return action;
}

void randomVel(Random r) {
   xvel = -1+r.nextInt(3);
   yvel = -1+r.nextInt(3);
}

void eat() {
   foodcount--;
   hunger = 0;
}

void drink() {
   watercount--;
   thirst = 0;
}

void setTarget(int x, int y) {
   positiontarget[0] = x;
   positiontarget[1] = y;
}

int[] getTarget() {
   return positiontarget;
}
```

```
void setInteractionTarget(int t) {
   interactiontarget = t;
}

void setHasTarget() {
   gotTarget = true;
}

int getInteractionTarget() {
   return interactiontarget;
}

void setInteraction(Norms n) {
   int o = 0;
   for(int i=0; i<opinions.size(); i++) {
      if(opinions.get(i)[0] == interactiontarget) {
         o = opinions.get(i)[1];
      }
   }
   //System.out.println("Opinion of " + interactiontarget + ": " + o);
   if(o>15) {
      interaction = 5;
   }else if((temperment+n.getV()+o) < -25) {
      if((temperment+n.getV()+o) < -45) {
         if((temperment+n.getV()+o) < -65) {
            interaction = 6;
         }else{
            interaction = 4;
         }
      }else{
         interaction = 2;
      }
   }else if((temperment+n.getV()+o) > 25) {
      if((temperment+n.getV()+o) > 45) {
         interaction = 3;
      }else{
         interaction = 1;
      }
   }else{
      interaction = 0;
   }
}

int getInteraction() {
   return interaction;
}

boolean atTarget() {
   if(positiontarget[0]==xpos&&positiontarget[1]==ypos) {
      return true;
   }else{
      return false;
   }
}

int getResourceAction() {
   return resourceaction;
```

```
}

void setResourceAction(Norms n, ResourceBox f, int p) {
   if(f.getQuantity()>0) {
      resourceaction = 0;
   }else{
      if(p>1) {
         if(interactiontarget == -1) {
            resourceaction = -1;
            action = 0;
         }else{
            if(temperment+n.getS()+getOpinion(interactiontarget)>=0) {
               resourceaction = 1;
            }else{
               resourceaction = 2;
            }
         }
      }else{
         resourceaction = -1;
         action = 0;
      }
   }
}

void setAction(int i) {
   action = i;
}

void noTarget() {
   positiontarget[0] = -1;
   positiontarget[1] = -1;
   interactiontarget = -1;
   gotTarget = false;
}

boolean hasTarget() {
   return gotTarget;
}

int getTemperment() {
   return temperment;
}

void die() {
   alive = false;
}

void updateOpinion(int e, int a) {
   boolean done = false;
   int[] t = new int[2];
   for(int i=0; i<opinions.size(); i++) {
      if(opinions.get(i)[0] == e) {
         opinions.get(i)[1] += a;
         if(opinions.get(i)[1]>50) {
            opinions.get(i)[1] = 50;
         }
         if(opinions.get(i)[1]<-50) {
```

```
               opinions.get(i)[1] = -50;
            }
            done = true;
         }
      }
      if(!done) {
         t[0] = e;
         t[1] = a;
         opinions.add(t);
      }
   }

   int getOpinion(int e) {
      for(int i=0; i<opinions.size(); i++) {
         if(opinions.get(i)[0] == e) {
            return opinions.get(i)[1];
         }
      }
      return 0;
   }

   ArrayList<int[]> getOpinions() {
      return opinions;
   }

   void punched() {
      health -= 20;
   }

   void slapped() {
      health -= 10;
   }

   void hugged() {
      health += 10;
      if(health>100) {
         health = 100;
      }
   }

   void kissed() {
      health += 20;
      if(health>100) {
         health = 100;
      }
   }
}
```

Entity.java

```
import java.awt.Color;
import java.util.Random;
import java.util.ArrayList;

public class Entity {
```

```
    private int xpos;
    private int ypos;
    private int xvel;
    private int yvel;
    private int width;
    private int dob;
    private boolean alive;
    private Color mycolor;
    private int hunger;
    private int thirst;
    private int health;
    private int foodcount;
    private int watercount;
    private int minpos;
    private int maxpos;
    private int interactiontarget;
    private int[] positiontarget;
    private int action;
    //private int number;
    //0:   idle
    //1:   eat food
    //2:   drink water
    //3:   obtain food
    //4:   obtain water
    //5:   move randomly
    //6:   interact
    private int resourceaction;
    //0:   get food
    //1:   request food
    //2:   steal food
    private int interaction;
    //0:   talk
    //1:   hug
    //2:   slap
    //3:   kiss
    //4:   punch
    //5:   mate
    //6:   kill
    private int temperment;
    private ArrayList<int[]> opinions;
    private boolean gotTarget;

    public Entity(Random r, int max, int min, int turn, int grid, int num, int
mt) {
       minpos = min;
       maxpos = max;
       xpos = minpos+r.nextInt(maxpos);
       ypos = minpos+r.nextInt(maxpos);
       xvel = 0;
       yvel = 0;
       health = 100;
       thirst = r.nextInt(100);
       hunger = r.nextInt(100);
       foodcount = 0;
       watercount = 0;
       width = grid;
       dob = turn;
```

```
      alive = true;
      gotTarget = false;
      temperment = -50+r.nextInt(101)+mt;
      if(temperment<-50) {
         temperment = -50;
      }
      if(temperment>50) {
         temperment = 50;
      }
      if(temperment<=-25) {
         mycolor = Color.red;

      }else if(temperment>=25) {
         mycolor = Color.green;
      }else{
         mycolor = Color.yellow;
      }
      action = 0;
      resourceaction = 0;
      positiontarget = new int[2];
      interactiontarget = -1;
      opinions = new ArrayList<int[]>();
   }

   void setxpos(int pos) {
      xpos = pos;
   }
   void setypos(int pos) {
      ypos = pos;
   }
   int getxpos() {
      return xpos;
   }
   int getypos() {
      return ypos;
   }
   int getwidth() {
      return width;
   }

   Color getColor() {
      return mycolor;
   }

   String move() {
      String temp = "";
      temp = " moved from " + xpos + ":" + ypos + " to ";
      xpos += xvel;
      ypos += yvel;
      temp += xpos + ":" + ypos;
      return temp;
   }

   int[] premove() {
      int[] temp = new int[2];
      temp[0] = xpos+xvel;
      temp[1] = ypos+yvel;
```

```
   return temp;
}

void incFood() {
   if(foodcount<3) {
      foodcount++;
   }
}

void incWater() {
   if(watercount<3) {
      watercount++;
   }
}

void decFood() {
   if(foodcount>0) {
      foodcount--;
   }
}

void decWater() {
   if(watercount>0) {
      watercount--;
   }
}

void updateVitals(int t, int m) {
   hunger++;
   thirst++;
   if(hunger>99) {
      health--;
   }
   if(thirst>99) {
      health -= 2;
   }
   if(health<=0) {
      alive = false;
   }
   if(t-dob>m) {
      alive = false;
   }
}

boolean isAlive() {
   return alive;
}

int getFoodCount() {
   return foodcount;
}

int getWaterCount() {
   return watercount;
}

void zeroVel() {
```

```
      xvel = 0;
      yvel = 0;
   }

   void reversevel() {
      xvel = -xvel;
      yvel = -yvel;
   }

   void setVel(int[] v) {
      xvel = v[0]-xpos;
      yvel = v[1]-ypos;
   }

   int getage(int turn) {
      return (turn-dob);
   }

   void chooseAction(Random r, int p) {
      if(hunger>49&&thirst<=49) {
         if(foodcount>0) {
            action = 1;
         }else{
            action = 3;
         }
      }else if(hunger<=49&&thirst>49) {
         if(watercount>0) {
            action = 2;
         }else{
            action = 4;
         }
      }else if(hunger>49&&thirst>49) {
         if((foodcount>0)&&(watercount==0)) {
            action = 1;
         }else if((foodcount==0)&&(watercount>0)) {
            action = 2;
         }else if((foodcount>0)&&(watercount>0)){
            if(hunger>thirst) {
               action = 1;
            }else{
               action = 2;
            }
         }else{
            if(hunger>thirst) {
               action = 3;
            }else{
               action = 4;
            }
         }
      }else{
         if(p>1) {
            action = 1+r.nextInt(6);
         }else{
            action = 1+r.nextInt(5);
         }
      }
   }
```

```
int getAction() {
   return action;
}

void randomVel(Random r) {
   xvel = -1+r.nextInt(3);
   yvel = -1+r.nextInt(3);
}

void eat() {
   foodcount--;
   hunger = 0;
}

void drink() {
   watercount--;
   thirst = 0;
}

void setTarget(int x, int y) {
   positiontarget[0] = x;
   positiontarget[1] = y;
}

int[] getTarget() {
   return positiontarget;
}

void setInteractionTarget(int t) {
   interactiontarget = t;
}

void setHasTarget() {
   gotTarget = true;
}

int getInteractionTarget() {
   return interactiontarget;
}

void setInteraction(Norms n) {
   int o = 0;
   for(int i=0; i<opinions.size(); i++) {
      if(opinions.get(i)[0] == interactiontarget) {
         o = opinions.get(i)[1];
      }
   }
   //System.out.println("Opinion of " + interactiontarget + ": " + o);
   if(o>15) {
      interaction = 5;
   }else if((temperment+n.getV()+o) < -25) {
      if((temperment+n.getV()+o) < -45) {
         if((temperment+n.getV()+o) < -65) {
            interaction = 6;
         }else{
            interaction = 4;
```

```
            }
         }else{
            interaction = 2;
         }
      }else if((temperment+n.getV()+o) > 25) {
         if((temperment+n.getV()+o) > 45) {
            interaction = 3;
         }else{
            interaction = 1;
         }
      }else{
         interaction = 0;
      }
   }

   int getInteraction() {
      return interaction;
   }

   boolean atTarget() {
      if(positiontarget[0]==xpos&&positiontarget[1]==ypos) {
         return true;
      }else{
         return false;
      }
   }

   int getResourceAction() {
      return resourceaction;
   }

   void setResourceAction(Norms n, ResourceBox f, int p) {
      if(f.getQuantity()>0) {
         resourceaction = 0;
      }else{
         if(p>1) {
            if(interactiontarget == -1) {
               resourceaction = -1;
               action = 0;
            }else{
               if(temperment+n.getS()+getOpinion(interactiontarget)>=0) {
                  resourceaction = 1;
               }else{
                  resourceaction = 2;
               }
            }
         }else{
            resourceaction = -1;
            action = 0;
         }
      }
   }

   void setAction(int i) {
      action = i;
   }
```

```
void noTarget() {
   positiontarget[0] = -1;
   positiontarget[1] = -1;
   interactiontarget = -1;
   gotTarget = false;
}

boolean hasTarget() {
   return gotTarget;
}

int getTemperment() {
   return temperment;
}

void die() {
   alive = false;
}

void updateOpinion(int e, int a) {
   boolean done = false;
   int[] t = new int[2];
   for(int i=0; i<opinions.size(); i++) {
      if(opinions.get(i)[0] == e) {
         opinions.get(i)[1] += a;
         if(opinions.get(i)[1]>50) {
            opinions.get(i)[1] = 50;
         }
         if(opinions.get(i)[1]<-50) {
            opinions.get(i)[1] = -50;
         }
         done = true;
      }
   }
   if(!done) {
      t[0] = e;
      t[1] = a;
      opinions.add(t);
   }
}

int getOpinion(int e) {
   for(int i=0; i<opinions.size(); i++) {
      if(opinions.get(i)[0] == e) {
         return opinions.get(i)[1];
      }
   }
   return 0;
}

ArrayList<int[]> getOpinions() {
   return opinions;
}

void punched() {
   health -= 20;
}
```

```
   void slapped() {
      health -= 10;
   }

   void hugged() {
      health += 10;
      if(health>100) {
         health = 100;
      }
   }

   void kissed() {
      health += 20;
      if(health>100) {
         health = 100;
      }
   }
}
```

InitPanel.java

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

//import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
//import javax.swing.JSlider;
import javax.swing.JSpinner;
import javax.swing.SpinnerListModel;
import javax.swing.SpinnerNumberModel;

public class InitFrame extends JFrame implements ActionListener {
   private boolean initialized;
   private boolean uservars;
   private JPanel initPanel = new JPanel();
   private JPanel buttonPanel = new JPanel();
   private String[] gList = { "10", "12", "15", "20", "24", "25", "30", "40",
"50", "60" };
   private JSpinner gSpin = new JSpinner();
   private JSpinner eSpin = new JSpinner();
   private JSpinner meSpin = new JSpinner();
   private JSpinner fSpin = new JSpinner();
   private JSpinner wSpin = new JSpinner();
   private JSpinner tSpin = new JSpinner();
   private JSpinner sSpin = new JSpinner();
   private JSpinner vSpin = new JSpinner();
   private JSpinner twfSpin = new JSpinner();
   private JSpinner twwSpin = new JSpinner();
```

```
private JSpinner maxtSpin = new JSpinner();
private JCheckBox vCheck= new JCheckBox("Visualize?");
private JCheckBox lCheck = new JCheckBox("Log Results?");
private JButton okbutton = new JButton();
private JButton canbutton = new JButton();

public InitFrame(boolean yn) {
   if(yn) {
      initialized = false;
      uservars = true;
      try {
         this.setTitle("Initialize Variables");
         this.setResizable(false);
         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         this.getContentPane().setLayout(new BorderLayout());

         initPanel.setLayout(new GridLayout(12,2,10,10));
         initPanel.add(new JLabel("Gridsize: "));
         gSpin.setModel(new SpinnerListModel(gList));
         gSpin.setValue("20");
         gSpin.setAlignmentY(1);
         initPanel.add(gSpin);
         initPanel.add(new JLabel("Entities:"));
         eSpin.setModel(new SpinnerNumberModel(10, 1, 20, 1));
         initPanel.add(eSpin);
         initPanel.add(new JLabel("Max Entities:"));
         meSpin.setModel(new SpinnerNumberModel(20, 1, 20, 1));
         initPanel.add(meSpin);
         initPanel.add(new JLabel("Food Quantity:"));
         fSpin.setModel(new SpinnerNumberModel(100, 0, 100000, 1));
         initPanel.add(fSpin);
         initPanel.add(new JLabel("Water Quantity:"));
         wSpin.setModel(new SpinnerNumberModel(100, 0, 100000, 1));
         initPanel.add(wSpin);
         initPanel.add(new JLabel("Median Temperment:"));
         tSpin.setModel(new SpinnerNumberModel(0,-50,50,1));
         initPanel.add(tSpin);
         initPanel.add(new JLabel("Stealing Norm:"));
         sSpin.setModel(new SpinnerNumberModel(0,-50,50,1));
         initPanel.add(sSpin);
         initPanel.add(new JLabel("Violence Norm:"));
         vSpin.setModel(new SpinnerNumberModel(0,-50,50,1));
         initPanel.add(vSpin);
         initPanel.add(new JLabel("Turns W/O Food: "));
         twfSpin.setModel(new SpinnerNumberModel(50,0,100,1));
         initPanel.add(twfSpin);
         initPanel.add(new JLabel("Turns W/O Water: "));
         twwSpin.setModel(new SpinnerNumberModel(50,0,100,1));
         initPanel.add(twwSpin);
         initPanel.add(new JLabel("Max Turns: "));
         maxtSpin.setModel(new SpinnerNumberModel(50000,1,100000,1));
         initPanel.add(maxtSpin);
         vCheck.setSelected(true);
         initPanel.add(vCheck);
         lCheck.setSelected(false);
         initPanel.add(lCheck);
```

```
            buttonPanel.setLayout(new GridLayout(1,2,10,10));
            okbutton.setActionCommand("OK");
            okbutton.setToolTipText("OK");
            okbutton.setText("OK");
            okbutton.addActionListener(this);
            buttonPanel.add(okbutton);
            canbutton.setActionCommand("CANCEL");
            canbutton.setToolTipText("CANCEL");
            canbutton.setText("CANCEL");
            canbutton.addActionListener(this);
            buttonPanel.add(canbutton);

            this.getContentPane().add(initPanel, BorderLayout.NORTH);
            this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
            this.pack();
            this.setVisible(true);

         }catch(Exception ex) {
            ex.printStackTrace();
         }
      }else{
         initialized = true;
         uservars = false;
      }
   }

   public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if(cmd.equals("OK")) {
           initialized = true;
        }
        if(cmd.equals("CANCEL")) {
           System.exit(0);
        }
   }

   public boolean initialized() {
      return initialized;
   }

   public boolean printLog() {
      return lCheck.isSelected();
   }

   public int getGrid() {
      return Integer.parseInt(gSpin.getValue().toString());
   }

   public int getEntity() {
      return Integer.parseInt(eSpin.getValue().toString());
   }

   public int getMaxE() {
      return Integer.parseInt(meSpin.getValue().toString());
   }

   public int getFood() {
```

```
        return Integer.parseInt(fSpin.getValue().toString());
    }

    public int getWater() {
        return Integer.parseInt(wSpin.getValue().toString());
    }

    public int getMT() {
        return Integer.parseInt(tSpin.getValue().toString());
    }

    public int getS() {
        return Integer.parseInt(sSpin.getValue().toString());
    }

    public int getV() {
        return Integer.parseInt(vSpin.getValue().toString());
    }

    public int getTWF() {
        return Integer.parseInt(twfSpin.getValue().toString());
    }

    public int getTWW() {
        return Integer.parseInt(twwSpin.getValue().toString());
    }

    public int getMaxT() {
        return Integer.parseInt(maxtSpin.getValue().toString());
    }

    public boolean getVisualize() {
        return vCheck.isSelected();
    }

    public boolean getVars() {
        return uservars;
    }
}
```

Norms.java

```
public class Norms {
    private int[] norms;
    public Norms() {
        norms = new int[2];
        norms[0] = 0;
        norms[1] = 0;
    }

    public Norms(int violence, int stealing) {
        norms = new int[3];
        norms[0] = violence;
        norms[1] = stealing;
    }
```

```
   public void updateNorms(int i, int a) {
      norms[i] += a;
      if(norms[i]>50) {
         norms[i] = 50;
      }
      if(norms[i]<-50) {
         norms[i] = -50;
      }
   }

   int getV() {
      return norms[0];
   }

   int getS() {
      return norms[1];
   }
}
```

Pathfinder.java

```
import java.util.ArrayList;
import java.io.*;
public class Pathfinder {
   private int sx;
   private int sy;
   private int ex;
   private int ey;
   private int[][] map;
   //new
   private int min;
   private int max;

   //added max
   public Pathfinder(int x1, int y1, int[] target, int[][] m, int min2, int
max2) {
      sx = x1;
      sy = y1;
      ex = target[0];
      ey = target[1];
      map = m;
      //new
      min = min2;
      max = max2;
   }

   public int getDistance(int cx, int cy, int tx, int ty) {
      int x = Math.abs(cx-tx);
      int y = Math.abs(cy-ty);
      int h = 0;
      if(x>y) {
         h = 14*y + 10*(x-y);
      }else{
```

```
      h = 14*x + 10*(y-x);
    }
    return h;
  }

  public void openNodes(Node n, NodeList o, NodeList c) {
    Node n2;
    int g = 0;
    int h = 0;
    for(int i=-1; i<2; i++) {
      for(int j=-1; j<2; j++) {
        g = getDistance(n.x, n.y, n.x+i, n.y+j) + n.g;
        h = getDistance(ex, ey, n.x+i, n.y+j);
        n2 = new Node(n.x+i, n.y+j, n.x, n.y, g+h, g, h);
        //new
        if((n.x+i>=min)&&(n.y+j>=min)&&(n.x+i<max)&&(n.y+j<max)) {
          //changed this to make it work
          if(!(i==0&&j==0)&&(map[n.x+i][n.y+j] == 0)&&(c.exists(n2)==-
1)) {
            if(o.exists(n2)>=0) {
              o.replaceIfBetterG(n2, o.exists(n2));
            }else{
              //System.out.println("Added " + n2.x + "," + n2.y + " to
the open list");
              o.add(n2);
            }
          }
        }
      }
    }
  }

  public int[] findPath() {
    NodeList open = new NodeList();
    NodeList closed = new NodeList();
    NodeList path = new NodeList();
    Node n = new Node(sx, sy, 0, 0, 0, 0, 0);
    Node start = n;
    Node n2;
    Node end = new Node(ex, ey, 0,0,0,0,0);
    int result[] = new int[2];
    open.add(n);
    while((closed.exists(end)<0)&&(open.size()>0)) {
      n = open.getBestF();

      //System.out.println("Checking " + n.x + "," + n.y);
      open.remove(n);
      closed.add(n);
      openNodes(n, open, closed);
      /*
      System.out.println("********************************");
      System.out.println("Open Contents");
      open.printContents();
      System.out.println("Closed Contents");
      closed.printContents();
      /*
      try {
```

```
            System.in.read();
         }catch(IOException e) {

         }
         */
      }
      n2 = closed.get(closed.size()-1);
      while(n2!=start) {
         path.add(n2);
         n2 = closed.get(closed.findParent(n2));
      }
      path.add(n);
      //System.out.println("\n*******************************");
      /*System.out.println("Final Path");
      for(int i=path.size()-2; i>=0; i--) {
         System.out.println(path.get(i).x +","+path.get(i).y);
      }
      */
      if(path.size()>1) {
         result[0] = path.get(path.size()-2).x;
         result[1] = path.get(path.size()-2).y;
      }else{
         result[0] = path.get(path.size()-1).x;
         result[1] = path.get(path.size()-1).y;
      }
      return result;

   }
}

class Node {
   public int x;
   public int y;
   public int px;
   public int py;
   public int f;
   public int g;
   public int h;

   public Node(int x1, int y1, int px1, int py1, int f1, int g1, int h1) {
      x = x1;
      y = y1;
      px = px1;
      py = py1;
      f = f1;
      g = g1;
      h = h1;
   }
}

class NodeList {
   ArrayList<Node> nodes;

   public NodeList() {
      nodes = new ArrayList<Node>();
   }
```

```
   void add(Node n) {
      nodes.add(n);
   }

   void replaceIfBetterG(Node n, int i) {
      if(n.g<nodes.get(i).g) {
         //new
         //System.out.println("Replacing " + n.x + "," + n.y + " G value " +
nodes.get(i).g + " with " + n.g);
         nodes.set(i,n);
      }
   }

   int size() {
      return nodes.size();
   }

   Node get(int i) {
      return nodes.get(i);
   }

   void remove(Node n) {
      nodes.remove(n);
   }

   Node getBestF() {
      Node n = nodes.get(0);
      for(int i=1; i<nodes.size(); i++) {
         if(nodes.get(i).f<n.f) {
            n = nodes.get(i);
         }
      }
      return n;
   }

   int exists(Node n) {
      int i=-1;
      for(int j=0; j<nodes.size(); j++) {
         if((n.x==nodes.get(j).x)&&(n.y==nodes.get(j).y)) {
            i = j;
         }
      }
      return i;
   }

   void printContents() {
      for(int i=0; i<nodes.size(); i++) {
         //System.out.println(i+": "+nodes.get(i).x+","+nodes.get(i).y + "
Parent: " + nodes.get(i).px +"," + nodes.get(i).py + " F: " + nodes.get(i).f
+ " G: " + nodes.get(i).g + " H: " + nodes.get(i).h);
      }
   }

   int findParent(Node n) {
      int h=-1;
      for(int i=0; i<nodes.size(); i++) {
         if((n.px==nodes.get(i).x)&&(n.py==nodes.get(i).y)) {
```

```
            //System.out.println(i);
            h = i;
         }
      }
      return h;
   }
}
```

ResourceBox.java

```
import java.util.Random;

public class ResourceBox {
   private int resetq;
   private int quantity;
   private int xpos;
   private int ypos;
   private int width;
   private int type;

   public ResourceBox(Random r, int q, int maxpos, int minpos, int t, int
grid, int[][] cm) {
      quantity = q;
      resetq = q;
      width = 4*grid;
      xpos = r.nextInt(maxpos-4)+minpos;
      ypos = r.nextInt(maxpos-4)+minpos;
      while(cm[xpos/grid][ypos/grid] != 0) {
         xpos = r.nextInt(maxpos-4)+minpos;
         ypos = r.nextInt(maxpos-4)+minpos;
      }
      type = t;
   }

   public void setQuantity(int n) {
      quantity = n;
   }

   public int getQuantity() {
      return quantity;
   }

   void decQuantity() {
      quantity--;
   }

   public int getxpos() {
      return xpos;
   }

   public int getypos() {
      return ypos;
   }
```

```
   public int getwidth() {
      return width;
   }

   public int getType() {
      return type;
   }

   public void reset() {
      quantity = resetq;
   }
}
```

Simulation.java

```
import java.awt.Color;
import java.util.Random;
import java.util.ArrayList;

public class Simulation {
   static final Random r = new Random();
   private static final int maxage = 10000;
   private ArrayList<Entity> denziens;
   private Norms normsystem;
   private ResourceBox foodbox;
   private ResourceBox waterbox;
   private Pathfinder pf;
   private int[][] collisionmap;
   private int[][] positionmap;
   private int turn;
   private String log;
   private int maxpop;
   private int minpos;
   private int maxpos;
   private int gridsize;
   private int births;
   private int murders;
   private int slaps;
   private int punches;
   private int hugs;
   private int kisses;
   private int donations;
   private int thefts;
   private int twf;
   private int tww;
   private int maxtwf;
   private int maxtww;


   public Simulation(int entitycount, int maxecount, int foodcount, int
watercount, int maxsize, int grid, int mt, int v, int s, int mtwf, int mtww)
{
      r.setSeed(System.currentTimeMillis());
      maxpop = maxecount;
      minpos = 0;
```

```
      maxpos = maxsize;
      gridsize = grid;
      collisionmap = new int [maxpos+1][maxpos+1];
      positionmap = new int[maxpos+1][maxpos+1];
      denziens = new ArrayList<Entity>();
      normsystem = new Norms(v, s);
      foodbox = new ResourceBox(r, foodcount, maxpos, minpos,1, gridsize,
collisionmap);
      loadBoxToMap(foodbox);
      waterbox = new ResourceBox(r, watercount, maxpos, minpos, 2, gridsize,
collisionmap);
      loadBoxToMap(waterbox);
      turn = 0;
      log = "";
      births = 0;
      murders = 0;
      slaps = 0;
      punches = 0;
      hugs = 0;
      kisses = 0;
      donations = 0;
      thefts = 0;
      maxtwf = mtwf;
      maxtww = mtww;
      twf = 0;
      tww = 0;

      for(int i=0; i<entitycount; i++) {
         denziens.add(new Entity(r, maxpos, minpos, turn, gridsize, i, mt));

while(collisionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] !=
0) {
            denziens.set(i, new Entity(r, maxpos, minpos, turn, gridsize, i,
mt));
         }
         collisionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] =
1;
         positionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] =
i;
      }
   }

   public Simulation() {
      denziens = null;
   }

   private int collision(int[] e) {
      if(e[0]<minpos||e[0]>maxpos||e[1]<minpos||e[1]>maxpos) {
         return -10;
      }else{
         return collisionmap[e[0]][e[1]];
      }
   }

   public int distanceTo(int cx, int tx, int cy, int ty) {
      int x = Math.abs(cx-tx);
      int y = Math.abs(cy-ty);
```

```
      int h = 0;
      if(x>y) {
         h = 14*y + 10*(x-y);
      }else{
         h = 14*x + 10*(y-x);
      }
      return h;
   }

   private void closestBoxSpot(Entity e, ResourceBox r) {
      int ex = e.getxpos();
      int ey = e.getypos();
      int rx = r.getxpos();
      int ry = r.getypos();
      int distance = 100000;

      for(int i=-1; i<5; i++) {
         for(int j=-1; j<5; j++) {
            if((rx+i>=minpos)&&(ry+j>=minpos)&&(rx+i<maxpos)&&(ry+j<maxpos))
{
               if((distanceTo(ex, rx+i, ey,
ry+j)<distance)&&(collisionmap[rx+i][ry+j]==0)) {
                  e.setTarget(rx+i, ry+j);
                  distance = distanceTo(ex, rx+i, ey, ry+j);
               }
            }
         }
      }
   }

   private void closestEntitySpot(Entity e) {
      int ex = e.getxpos();
      int ey = e.getypos();
      Entity t = denziens.get(e.getInteractionTarget());
      int tx = t.getxpos();
      int ty = t.getypos();
      int distance = 1000000;
      for(int i=-1; i<2; i++) {
         for(int j=-1; j<2; j++) {
            if((tx+i>=minpos)&&(ty+j>=minpos)&&(tx+i<maxpos)&&(ty+j<maxpos))
{
               if((distanceTo(ex, tx+i, ey,
ty+j)<distance)&&(collisionmap[tx+i][ty+j]==0)) {
                  e.setTarget(tx+i, ty+j);
                  distance = distanceTo(ex, tx+i, ey, ty+j);
               }
            }
         }
      }
   }

   private int[] freeSurroundingSpot(Entity e) {
      int ex = e.getxpos();
      int ey = e.getypos();
      int a[] = new int[2];
      a[0] = -1;
      a[1] = -1;
```

```
      for(int i=-1; i<2; i++) {
         for(int j=-1; j<2; j++) {
            if((ex+i>=minpos)&&(ey+j>=minpos)&&(ex+i<maxpos)&&(ey+j<maxpos))
{
               if(collisionmap[ex+i][ey+j]==0) {
                  a[0] = ex+i;
                  a[1] = ey+j;
               }
            }
         }
      }
      return a;
   }

   private void combineOpinions(Entity s1, Entity s2, Entity t) {
      ArrayList<int[]> o1 = s1.getOpinions();
      ArrayList<int[]> o2 = s2.getOpinions();
      ArrayList<int[]> opinions = new ArrayList<int[]>();
      int[] temp = new int[2];
      for(int i=0; i<o1.size(); i++) {
         opinions.add(o1.get(i));
      }
      for(int j=0; j<o2.size(); j++) {
         for(int k=0; k<opinions.size(); k++) {
            if(o2.get(j)[0]==opinions.get(k)[0]) {
               temp[0] = o2.get(j)[0];
               temp[1] = (o2.get(j)[1]+opinions.get(k)[1])/2;
               opinions.set(k, temp);
            }else{
               opinions.add(o2.get(j));
            }
         }
      }
   }

   private int randomEntity(int e) {
      int t = r.nextInt(denziens.size());
      while((t == e)||(!denziens.get(t).isAlive())) {
         t = r.nextInt(denziens.size());
      }
      return t;
   }

   private String doTalk(int e, int t) {
      Entity a = denziens.get(e);
      Entity b = denziens.get(t);
      String l = "";
      if(Math.abs(a.getTemperment()-b.getTemperment())<50) {
         l = " succeeded.\n";
         a.updateOpinion(t,1);
         b.updateOpinion(e,1);
      }else{
         l = " failed.\n";
         a.updateOpinion(t,-1);
         b.updateOpinion(e,-1);
      }
      return l;
```

```
}

private void doHug(int e, int t) {
   Entity a = denziens.get(e);
   Entity b = denziens.get(t);
   normsystem.updateNorms(0,1);
   a.updateOpinion(t, 5);
   b.updateOpinion(e, 5);
   hugs++;
}

private void doSlap(int e, int t) {
   Entity a = denziens.get(e);
   Entity b = denziens.get(t);
   normsystem.updateNorms(0,-1);
   a.updateOpinion(t, -5);
   b.updateOpinion(e, -5);
   b.slapped();
   slaps++;
}

private void doKiss(int e, int t) {
   Entity a = denziens.get(e);
   Entity b = denziens.get(t);
   normsystem.updateNorms(0,3);
   a.updateOpinion(t, 10);
   b.updateOpinion(e, 10);
   kisses++;
}

private void doPunch(int e, int t) {
   Entity a = denziens.get(e);
   Entity b = denziens.get(t);
   normsystem.updateNorms(0,-3);
   a.updateOpinion(t, -10);
   b.updateOpinion(e, -10);
   kisses++;
}

private void doKill(int e, int t) {
   normsystem.updateNorms(0,-5);
   denziens.get(t).die();
   murders++;
}

private void doGive(int e, int t, boolean what) {
   Entity a = denziens.get(e);
   Entity b = denziens.get(t);
   if(what) {
      a.incFood();
      b.decFood();
   }else{
      a.incWater();
      b.decWater();
   }
   normsystem.updateNorms(1,1);
   a.updateOpinion(t, 3);
```

```
      b.updateOpinion(e, 3);
      donations++;
   }

   private void doSteal(int e, int t, boolean what) {
      Entity a = denziens.get(e);
      Entity b = denziens.get(t);
      if(what) {
         a.incFood();
         b.decFood();
      }else{
         a.incWater();
         b.decWater();
      }
      normsystem.updateNorms(1,-1);
      a.updateOpinion(t, -7);
      b.updateOpinion(e, -7);
      thefts++;
   }

   private void doMate(int e, int t) {
      Entity a = denziens.get(e);
      Entity b = denziens.get(t);
      int mt = (a.getTemperment()+b.getTemperment())/2;
      Entity c = new Entity(r, maxpos, minpos, turn, gridsize,
denziens.size(), mt);
      int[] pos = freeSurroundingSpot(b);
      c.setxpos(pos[0]);
      c.setypos(pos[1]);
      denziens.add(c);
      a.updateOpinion(t,50);
      b.updateOpinion(e,50);
      combineOpinions(a,b,c);
      //logic behind this:
      //need to balance norm changes so that things don't spiral out of
control
      //births will soften a society and make it less prone to violence?
      normsystem.updateNorms(0,5);
      births++;
   }

   private int closestEntityWith(int type, Entity e) {
      int x = e.getxpos();
      int y = e.getypos();
      int x2;
      int y2;
      int distance = 1000000;
      int target = -1;
      for(int i=0; i<denziens.size(); i++) {
         x2 = denziens.get(i).getxpos();
         y2 = denziens.get(i).getypos();
         if(!((x==x2)&&(y==y2))&&(distanceTo(x,x2,y,y2)<distance)) {
            switch(type) {
               case 0:
                  if(denziens.get(i).getFoodCount()>0) {
                     target = i;
                  }
```

```
                break;
            case 1:
                if(denziens.get(i).getWaterCount()>0) {
                    target = i;
                }
            }
        }
    }
    return target;
}

private void randomTarget(Entity e) {
    int x = minpos + (r.nextInt(maxpos-minpos));
    int y = minpos + (r.nextInt(maxpos-minpos));
    while(collisionmap[x][y] != 0) {
        x = minpos + (r.nextInt(maxpos-minpos));
        y = minpos + (r.nextInt(maxpos-minpos));
    }
    e.setTarget(x, y);
}

private boolean surrounded(Entity e) {
    int x = 0;
    int y = 0;
    boolean result = true;
    for(int i=-1; i<=1; i++) {
        for(int j=-1; j<=1; j++) {
            x = e.getxpos() + i;
            y = e.getypos() + j;
            if(x>=minpos&&y>=minpos&&x<=maxpos&&y<=maxpos) {
                if(collisionmap[x][y] == 0) {
                    result = false;
                }
            }
        }
    }
    return result;
}

void loadBoxToMap(ResourceBox b) {
    int x = b.getxpos();
    int y = b.getypos();
    for(int i=0; i<4; i++) {
        for(int j=0; j<4; j++) {
            collisionmap[x+i][y+j] = 1;
        }
    }
}

void nextTurn() {
    if(maxtwf <= twf) {
        foodbox.reset();
        twf=0;
    }
    if(maxtww <= tww) {
        waterbox.reset();
        tww=0;
```

```
      }
      turn++;
      /*System.out.println(turn);
      for(int k=0; k<maxpos; k++) {
         for(int l=0; l<maxpos; l++) {
            System.out.print(collisionmap[k][l]);
         }
         System.out.println();
      }
      System.out.println("\n");*/
      for(int i=0; i<denziens.size(); i++) {
         denziens.get(i).updateVitals(turn, maxage);
         if(denziens.get(i).isAlive()) {
            if(surrounded(denziens.get(i))) {
               denziens.get(i).zeroVel();
            }else{
               switch(denziens.get(i).getAction()) {
                  case 0:
                     denziens.get(i).chooseAction(r, getPopulation());
                     break;
                  case 1:
                     denziens.get(i).zeroVel();
                     if(denziens.get(i).getFoodCount()>0) {
                        denziens.get(i).eat();
                        log += "Entity " + i + " ate.  Now has " +
denziens.get(i).getFoodCount() + ".\n";
                        denziens.get(i).setAction(0);
                     }else{
                        denziens.get(i).setAction(3);
                     }
                     break;
                  case 2:
                     denziens.get(i).zeroVel();
                     if(denziens.get(i).getWaterCount()>0) {
                        denziens.get(i).drink();
                        log += "Entity " + i + " drank.  Now has " +
denziens.get(i).getWaterCount() + ".\n";
                        denziens.get(i).setAction(0);
                     }else{
                        denziens.get(i).setAction(4);
                     }
                     break;
                  case 3:
                     if(!denziens.get(i).hasTarget()) {

denziens.get(i).setInteractionTarget(closestEntityWith(0,denziens.get(i)));
                        denziens.get(i).setResourceAction(normsystem,
foodbox, getPopulation());

if(denziens.get(i).getResourceAction()!=0&&denziens.get(i).getInteractionTarg
et()==-1){
                           denziens.get(i).setAction(5);
                        }else{
                           denziens.get(i).setHasTarget();
                        }
                     }else{
                        if(denziens.get(i).atTarget()) {
```

```
                              switch(denziens.get(i).getResourceAction()) {
                                 case 0:
                                    if(foodbox.getQuantity()>0) {
                                       denziens.get(i).incFood();
                                       foodbox.decQuantity();
                                       log += "Entity " + i + " got food.  Food
Count: " + denziens.get(i).getFoodCount() + ".\n";
                                    }else{
                                       denziens.get(i).noTarget();
                                    }
                                    break;
                                 case 1:

if(denziens.get(denziens.get(i).getInteractionTarget()).getFoodCount()>0) {

doGive(i,denziens.get(i).getInteractionTarget(), true);
                                    }else{
                                       denziens.get(i).noTarget();
                                    }
                                    break;
                                 case 2:

if(denziens.get(denziens.get(i).getInteractionTarget()).getWaterCount()>0) {

doSteal(i,denziens.get(i).getInteractionTarget(), true);
                                    }else{
                                       denziens.get(i).noTarget();
                                    }
                                    break;
                              }
                              denziens.get(i).noTarget();
                              denziens.get(i).setAction(0);
                           }else{
                              if(denziens.get(i).getResourceAction() == 0) {
                                 if(foodbox.getQuantity()>0) {
                                    closestBoxSpot(denziens.get(i), foodbox);
                                    pf = new
Pathfinder(denziens.get(i).getxpos(), denziens.get(i).getypos(),
denziens.get(i).getTarget(), collisionmap, minpos, maxpos);
                                    denziens.get(i).setVel(pf.findPath());
                                 }else{
                                    denziens.get(i).noTarget();
                                 }
                              }else{

if(denziens.get(denziens.get(i).getInteractionTarget()).getFoodCount()>0) {
                                    closestEntitySpot(denziens.get(i));
                                    pf = new
Pathfinder(denziens.get(i).getxpos(), denziens.get(i).getypos(),
denziens.get(i).getTarget(), collisionmap, minpos, maxpos);
                                    denziens.get(i).setVel(pf.findPath());
                                 }else{
                                    denziens.get(i).noTarget();
                                 }
                              }
                           }
                        }
```

```
                    break;
                 case 4:
                    if(!denziens.get(i).hasTarget()) {

denziens.get(i).setInteractionTarget(closestEntityWith(0,denziens.get(i)));
                       denziens.get(i).setResourceAction(normsystem,
waterbox, getPopulation());

if(denziens.get(i).getResourceAction()!=0&&denziens.get(i).getInteractionTarg
et()==-1){
                          denziens.get(i).setAction(5);
                       }else{
                          denziens.get(i).setHasTarget();
                       }
                    }else{
                       if(denziens.get(i).atTarget()) {
                          switch(denziens.get(i).getResourceAction()) {
                             case 0:
                                if(waterbox.getQuantity()>0) {
                                   denziens.get(i).incWater();
                                   waterbox.decQuantity();
                                   log += "Entity " + i + " got water.
Water Count: " + denziens.get(i).getWaterCount() + ".\n";
                                }else{
                                   denziens.get(i).noTarget();
                                }
                                break;
                             case 1:

if(denziens.get(denziens.get(i).getInteractionTarget()).getWaterCount()>0) {

doGive(i,denziens.get(i).getInteractionTarget(), true);
                                }else{
                                   denziens.get(i).noTarget();
                                }
                                break;
                             case 2:

if(denziens.get(denziens.get(i).getInteractionTarget()).getWaterCount()>0) {

doSteal(i,denziens.get(i).getInteractionTarget(), true);
                                }else{
                                   denziens.get(i).noTarget();
                                }
                                break;
                          }
                          denziens.get(i).noTarget();
                          denziens.get(i).setAction(0);
                       }else{
                          if(denziens.get(i).getResourceAction() == 0) {
                             if(foodbox.getQuantity()>0) {
                                closestBoxSpot(denziens.get(i), waterbox);
                                pf = new
Pathfinder(denziens.get(i).getxpos(), denziens.get(i).getypos(),
denziens.get(i).getTarget(), collisionmap, minpos, maxpos);
                                denziens.get(i).setVel(pf.findPath());
                             }else{
```

```
                                    denziens.get(i).noTarget();
                                 }
                              }else{

if(denziens.get(denziens.get(i).getInteractionTarget()).getWaterCount()>0) {
                                    closestEntitySpot(denziens.get(i));
                                    pf = new
Pathfinder(denziens.get(i).getxpos(), denziens.get(i).getypos(),
denziens.get(i).getTarget(), collisionmap, minpos, maxpos);
                                    denziens.get(i).setVel(pf.findPath());
                                 }else{
                                    denziens.get(i).noTarget();
                                 }
                              }
                           }
                        }
                        break;
                     case 5:
                        if(!denziens.get(i).hasTarget()) {
                           randomTarget(denziens.get(i));
                           denziens.get(i).setHasTarget();
                        }else{
                           if(denziens.get(i).atTarget()) {
                              denziens.get(i).noTarget();
                              denziens.get(i).setAction(0);
                           }else{
                              pf = new Pathfinder(denziens.get(i).getxpos(),
denziens.get(i).getypos(), denziens.get(i).getTarget(), collisionmap, minpos,
maxpos);
                              denziens.get(i).setVel(pf.findPath());
                           }
                        }
                        break;
                     case 6:
                        if(!denziens.get(i).hasTarget()) {

denziens.get(i).setInteractionTarget(randomEntity(i));
                           denziens.get(i).setInteraction(normsystem);
                           denziens.get(i).setHasTarget();
                        }else{

if(denziens.get(denziens.get(i).getInteractionTarget()).isAlive()) {
                              if(denziens.get(i).atTarget()) {
                                 switch(denziens.get(i).getInteraction()) {
                                    case 0:
                                       log += "Entity " + i + " attempted to
talk to Entity " + denziens.get(i).getInteractionTarget() + ".  It";
                                       log += doTalk(i,
denziens.get(i).getInteractionTarget());
                                       denziens.get(i).noTarget();
                                       denziens.get(i).setAction(0);
                                       break;
                                    case 1:
                                       log += "Entity " + i + " attempted to hug
Entity " + denziens.get(i).getInteractionTarget() + ".\n";
                                       doHug(i,
denziens.get(i).getInteractionTarget());
```

```
                                    denziens.get(i).noTarget();
                                    denziens.get(i).setAction(0);
                                    break;
                                 case 2:
                                    log += "Entity " + i + " slapped Entity "
+ denziens.get(i).getInteractionTarget() + ".\n";
                                    doSlap(i,
denziens.get(i).getInteractionTarget());
                                    denziens.get(i).noTarget();
                                    denziens.get(i).setAction(0);
                                    break;
                                 case 3:
                                    log += "Entity " + i + " kissed Entity "
+ denziens.get(i).getInteractionTarget() + ".\n";
                                    doKiss(i,
denziens.get(i).getInteractionTarget());
                                    denziens.get(i).noTarget();
                                    denziens.get(i).setAction(0);
                                    break;
                                 case 4:
                                    log += "Entity " + i + " punched Entity "
+ denziens.get(i).getInteractionTarget() + ".\n";
                                    doPunch(i,
denziens.get(i).getInteractionTarget());
                                    denziens.get(i).noTarget();
                                    denziens.get(i).setAction(0);
                                    break;
                                 case 5:
                                    log += "Entity " + i + " attempted to
mate with Entity " + denziens.get(i).getInteractionTarget();

if(foodbox.getQuantity()<20&&waterbox.getQuantity()<20) {
                                       log += " but decided against it due to
lack of resources.\n";
                                    }else
if(surrounded(denziens.get(denziens.get(i).getInteractionTarget()))||getPopul
ation()>=maxpop) {
                                       log += " but decided against it
because it's too crowded.\n";
                                    }else{
                                       doMate(i,
denziens.get(i).getInteractionTarget());
                                       log += " and a bouncing fully
functional adult was born.\n";
                                    }
                                    denziens.get(i).noTarget();
                                    denziens.get(i).setAction(0);
                                    break;
                                 case 6:
                                    log += "Entity " + i + " killed Entity "
+ denziens.get(i).getInteractionTarget() + ".\n";
                                    doKill(i,
denziens.get(i).getInteractionTarget());
                                    denziens.get(i).noTarget();
                                    denziens.get(i).setAction(0);
                                    break;
                              }
```

```
                              }else{
                                 closestEntitySpot(denziens.get(i));
                                 pf = new Pathfinder(denziens.get(i).getxpos(),
denziens.get(i).getypos(), denziens.get(i).getTarget(), collisionmap, minpos,
maxpos);
                                 denziens.get(i).setVel(pf.findPath());
                              }
                           }else{
                              denziens.get(i).noTarget();
                              denziens.get(i).setAction(0);
                           }
                        }
                        break;
                }

collisionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] = 0;

positionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] = 0;
                if(collision(denziens.get(i).premove())==0) {
                   denziens.get(i).move();
                }

collisionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] = 1;

positionmap[denziens.get(i).getxpos()][denziens.get(i).getypos()] = i;
             }
          }
       }
       if(foodbox.getQuantity()==0) {
          twf++;
       }
       if(waterbox.getQuantity()==0) {
          tww++;
       }
    }

    //Start stuff for drawing
    int getDenziensCount() {
       return denziens.size();
    }

    int getPopulation() {
       int p = 0;
       for(int i=0; i<denziens.size(); i++) {
          if(denziens.get(i).isAlive()) {
             p++;
          }
       }
       return p;
    }

    int getEntityXpos(int i) {
       return denziens.get(i).getxpos()*gridsize;
    }

    int getEntityYpos(int i) {
       return denziens.get(i).getypos()*gridsize;
```

```
}

int getEntityWidth(int i) {
   return denziens.get(i).getwidth();
}

Color getEntityColor(int i) {
   return denziens.get(i).getColor();
}

int getBoxXpos(int i) {
   int n = 0;
   switch(i) {
      case 0:
         n = foodbox.getxpos()*gridsize;
         break;
      case 1:
         n = waterbox.getxpos()*gridsize;
         break;
   }
   return n;
}

int getBoxYpos(int i) {
   int n = 0;
   switch(i) {
      case 0:
         n = foodbox.getypos()*gridsize;
         break;
      case 1:
         n = waterbox.getypos()*gridsize;
         break;
   }
   return n;
}

int getBoxWidth(int i) {
   int n = 0;
   switch(i) {
      case 0:
         n = foodbox.getwidth();
         break;
      case 1:
         n = waterbox.getwidth();
         break;
   }
   return n;
}

boolean isEntityAlive(int i) {
   return denziens.get(i).isAlive();
}

//End Stuff for drawing

//Start Stuff for menus
```

```
int getBoxQuantity(int i) {
   int n = 0;
   switch(i) {
      case 0:
         n = foodbox.getQuantity();
         break;
      case 1:
         n = waterbox.getQuantity();
         break;
   }
   return n;
}

void editBoxQuantity(int i, boolean add) {
   int n = 0;
   if(add) {
      n = 10;
   }else{
      n = -10;
   }
   switch(i) {
      case 0:
         foodbox.setQuantity(foodbox.getQuantity()+n);
         if(add) {
            if(foodbox.getQuantity()>200) {
               foodbox.setQuantity(200);
            }
         }else{
            if(foodbox.getQuantity()<0) {
               foodbox.setQuantity(0);
            }
         }
         break;
      case 1:
         waterbox.setQuantity(waterbox.getQuantity()+n);
         if(add) {
            if(waterbox.getQuantity()>200) {
               waterbox.setQuantity(200);
            }
         }else{
            if(waterbox.getQuantity()<0) {
               waterbox.setQuantity(0);
            }
         }
         break;
   }
}

//End stuff for menus

int getTurn() {
   return turn;
}

String getLog() {
   return log;
}
```

```
   String getResults() {
      String r = "";
      /*
      r += "Number of turns: " + Integer.toString(turn) + "\n";
      r += "Final Population: " + getPopulation() + "\n";
      r += "Give/Steal: " + normsystem.getGS() + "\n";
      r += "Donations: " + donations + "\n";
      r += "Thefts: " + thefts + "\n";
      r += "Hug/Hit: " + normsystem.getHH() + "\n";
      r += "Hugs: " + hugs + "\n";
      r += "Assaults: " + assaults + "\n";
      r += "Kiss/Kill: " + normsystem.getKK() + "\n";
      r += "Kisses: " + kisses + "\n";
      r += "Murders: " + murders + "\n";
      r += "Births: " + births + "\n";
      */
      r = Integer.toString(turn) + "," + getPopulation() + "," +
normsystem.getS() + "," + donations + "," + thefts + "," +normsystem.getV() +
"," + hugs + "," + slaps + "," + kisses + "," + punches + "," + murders + ","
+ births;
      return r;
   }

   Norms getNorms() {
      return normsystem;
   }
}
```

Thesis.java

```
import java.lang.Thread;
import java.io.*;

import javax.swing.*;

public class Thesis {

    public static void main(String[] args) {
       try {
         UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
      } catch (ClassNotFoundException e1) {
         e1.printStackTrace();
      } catch (InstantiationException e1) {
         e1.printStackTrace();
      } catch (IllegalAccessException e1) {
         e1.printStackTrace();
      } catch (UnsupportedLookAndFeelException e1) {
         e1.printStackTrace();
      }
       InitFrame startFrame;
       int gridsize = 20;
       int ecount = 10;
       int maxecount = 20;
       int foodcount = 100;
```

```
int watercount = 100;
int mediantemperment = 0;
int steal = 0;
int violence = 0;
int twf = 0;
int tww = 0;
int maxt = 0;
boolean visualize = true;
boolean showFinal = true;
boolean printLog = false;


Simulation s;
DrawingFrame ThesisFrame;
JFrame finalFrame;
JPanel finalPanel;
JTextArea finalText;

try{
    FileOutputStream fos = new FileOutputStream("output.txt");
    PrintStream ps = new PrintStream(fos);
    System.setOut(ps);
}catch(IOException e) {

}

if(args.length > 0) {
   gridsize = Integer.parseInt(args[0]);
   ecount = Integer.parseInt(args[1]);
   maxecount = Integer.parseInt(args[2]);
   foodcount = Integer.parseInt(args[3]);
   watercount = Integer.parseInt(args[4]);
   mediantemperment = Integer.parseInt(args[5]);
   steal = Integer.parseInt(args[6]);
   violence = Integer.parseInt(args[7]);
   twf = Integer.parseInt(args[8]);
   tww = Integer.parseInt(args[9]);
   maxt = Integer.parseInt(args[10]);
   visualize = Boolean.parseBoolean(args[11]);
   printLog = Boolean.parseBoolean(args[12]);
   showFinal = false;
}else{
   startFrame = new InitFrame(true);
  while(!startFrame.initialized()) {
     startFrame.repaint();
  }
  startFrame.setVisible(false);
  startFrame.removeAll();
  if(startFrame.getVars()) {
     gridsize = startFrame.getGrid();
      ecount = startFrame.getEntity();
      maxecount = startFrame.getMaxE();
      foodcount = startFrame.getFood();
      watercount = startFrame.getWater();
      mediantemperment = startFrame.getMT();
      steal = startFrame.getS();
      violence = startFrame.getV();
```

```
            twf = startFrame.getTWF();
            tww = startFrame.getTWW();
            maxt = startFrame.getMaxT();
            visualize = startFrame.getVisualize();
            printLog = startFrame.printLog();
         }
         showFinal = true;
      }

      s = new Simulation(ecount, maxecount, foodcount, watercount,
600/gridsize-1, gridsize, mediantemperment, steal, violence, twf, tww);

      if(visualize) {
         ThesisFrame = new DrawingFrame(s);
         while(!ThesisFrame.done()&&s.getPopulation()>0&&s.getTurn()<maxt) {
            if(ThesisFrame.going()) {
               s.nextTurn();
               try {
                  Thread.sleep(ThesisFrame.getspeed());
               }catch(InterruptedException e){
                  System.out.println(e);
               }
            }else if(ThesisFrame.step()) {
               s.nextTurn();
               ThesisFrame.stepped();
            }
            ThesisFrame.setSimulation(s);
            ThesisFrame.repaintStuff();
         }
         ThesisFrame.setVisible(false);
      }else{
         while(s.getPopulation()>0&&s.getTurn()<maxt) {
            s.nextTurn();
         }
      }
      if(printLog) {
         System.out.println(s.getResults()+s.getLog());
      }else{
         System.out.println(s.getResults());
      }
      if(showFinal) {
         finalFrame = new JFrame();
         finalPanel = new JPanel();
         finalText = new JTextArea(8,20);
         finalText.setText(s.getResults());
         //finalText.setEditable(false);
         finalPanel.add(finalText);
         finalFrame.getContentPane().add(finalPanel);
         finalFrame.pack();
         finalFrame.setTitle("Results");
         finalFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         finalFrame.setResizable(false);
         finalFrame.setVisible(true);
      }else{
         System.exit(0);
      }
   }
```

```
}
```