MIMICA: A GENERAL FRAMEWORK FOR SELF-LEARNING COMPANION AI BEHAVIOR

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Travis Angevine

June 2016

COMMITTEE MEMBERSHIP

TITLE:	MimicA: A General Framework for Self-Learning Companion AI Behavior
AUTHOR:	Travis Angevine
DATE SUBMITTED:	June 2016
COMMITTEE CHAIR:	Foaad Khosmood, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	Michael Haungs, Ph.D. Associate professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

MimicA: A General Framework for Self-Learning Companion AI Behavior

Travis Angevine

Companion or support characters controlled by Artificial Intelligence (AI) have been a feature of video games for decades. Many Role Playing Games (RPGs) offer a cast of support characters in the player's party that are AI-controlled to various degrees. Many First Person Shooter (FPS) games include semi-autonomous or fully autonomous AI-controlled companions. Real Time Strategy (RTS) games have traditionally featured large numbers of semi-autonomous characters that collectively help accomplish various tasks (build, attack, etc.) for the player. While RPGs tend to focus on a single or a small number of well-developed character companions to accompany a player controlled main character, the RTS games tend to have anonymous and replaceable workers and soldiers to be micromanaged by the player.

In this paper we present the MimicA framework, designed to govern AI companion behavior based on mimicking that of the player. Several features set this system apart from existing practices in AI-managed companions in contemporary RPG or RTS games. First, the behavior generated is designed to be fully autonomous, not partially autonomous as in most RTS games. Second, the solution is general. No specific prior behavior specifications are modeled. As a result, little to no genre, story or technical assumptions are necessary to implement this solution. Even the list of possible actions required is generalized. The system is designed to work independently of game representation. We further demonstrate, analyze and discuss MimicA by using it in *Lord of Towers*, a novel tower defense game featuring a player avatar. Through our user study we show that a majority of participants found the companions useful to them and liked the idea of this type of framework.

ACKNOWLEDGMENTS

Many thanks to Foaad Khosmood for his continued support in design and development of this project. Additional thanks to Michael Haungs and Franz Kurfess for taking the time to evaluate this project. Further thanks to Andrew Guenther and Corrigan Johnson in providing the template for this document. Lastly, thanks to everyone who took the time to take part in, and provide feedback for, the study done as part of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

As video games have developed from the early days of Pong [2] and Tetris [31] to 21st century hits like World of Warcraft [9] and Call of Duty: Modern Warfare [17], they have evolved in their style, depth, and difficulty. As different types of games have developed, so has the range of artificial intelligence (AI) used by the non-player characters (NPCs) in the games. This includes enemy characters that oppose the player, neutral characters that may support the player in their interactions with shops or quests, and companion characters that work alongside the character.

1.1 Description of the Problem

In most games support characters don't require any advanced, player-like AI because they have fixed behavior. They are there to sell the player items, provide quests, or other similar actions. These actions can be easily scripted in order to provide the level of interaction needed for these types of NPCs. So while research has been done to make sure these types of characters are believable [21], not as much effort needs to be made to make them player-like.

Aside from support NPCs, while much work has gone into developing highly sophisticated AI for enemy characters, less has been done for companion characters [3][23]. This lack of sophistication when it comes to companion characters can lead to frustration on the part of the player, especially if the companion is a required part of the game, because the player now has to attempt to work with this character that has strange, unintuitive behaviors. Companions are intended to be present in a game to aid the player in various ways. However, if the companions do not do what the player expects, or even inhibits the player from accomplishing goals in a desired way, the companion can quickly become an annoyance rather than a boon [33].

An example of this is seen in critiques of the companions in Skyrim [4], where the behavior of the player companions have led to reviews saying "... all they (the companions) really do is serve as a beast of burden for carrying your spare loot, ruining your stealth, activating every trap in a given area, or getting themselves killed" [10] and "Companion AI ... frequently steps in front of you to take friendly fire and just die" [25]. While some have tried to remedy this problem with player made mods [26], these issues present room for improvement in this area.

1.2 Overview of the Solution

Major contemporary trends in companion AI development are towards either creating fully autonomous companions, or creating companions still controlled by the player to some degree [33]. This work falls into the first category by focusing on developing a character that will behave completely autonomously from the player.

Good AI companions will aid in increasing the fun and immersion of a game [33], as well as allowing games to feel more life-like by providing more realistic player-NPC interactions. Additionally, they will allow for more complex strategies to be used both by game developers and the players because the NPCs working with the players will be closer in level of competence to the current state of enemy NPCs, as well minimizing the gap between player skill level and companion skill level. Finally, constructing good AI companions will improve the game experience for players by causing fewer situations similar to the problems mentioned previously in the reviews of Skyrim companions. MimicA aims to provide this through the creation of a fully autonomous companion.

1.3 Outline of the Thesis

Chapters 2 and 3 of this thesis discuss some of the background and related works of this project. Chapter 4 presents the design of the MimicA framework, while chapter 5 discusses the use of the framework in a game developed for this project, *Lord of Towers*. Chapter 6 outlines the user study performed in order to validate the MimicA framework, as well as the results of the study. Lastly, chapter 7 concludes with a summary of the contribution of this work, as well as some of the challenges faced during its development, and chapter 8 presents possible future work to this project.

Chapter 2

BACKGROUND

This chapter presents background research performed in areas involved in games and game AI. It provides a brief discussion of different types of agents, planning and learning techniques, and introduces three classifiers which the MimicA system uses. It also provides a discussion of adaptive gameplay and teammate AI.

2.1 Agent and Multi-Agent Environments

The companion developed in this thesis is a form of an automated agent designed to assist the player in progressing through the game. As described by Panait and Luke, "An agent is a computational mechanism that exhibits a high degree of autonomy, performing actions in its environment based on information (sensors, feedback) received from the environment. [28]" In a video game, NPCs are all agents inside the environment of the game. These NPCs are automated to perform some behavior, whether that is to attack the player's base in the case of the enemy characters, or to build walls and towers in the case of the player companions. Additionally, while a human player is not necessarily a "computational mechanism" they do still fit into the previous definition and can be considered an agent as well, just not an automated one. As such, we will differentiate between human agents and automated or AI agents if a distinction is needed.

Additionally, Panait and Luke define a multi-agent environment as one, "in which there is more than one agent, where they interact with each other. [28]" This is important to consider, as many video games are examples of multi-agent environments. Specifically, since MimicA aims to develop a companion AI that would work alongside the player, all games that MimicA could be used in would be multi-agent environments. Panait and Luke additionally discuss such an environment where one agent may not have the same knowledge about the environment that another agent does. This is important to consider as we determine how companions using MimicA gain knowledge about the environment they are present in. Ultimately, it is left to the game developer to decide how much information should be passed to MimicA, the details of which are discussed later in the paper. The game we have developed for the sake of testing MimicA opts to provide all agents present in the game with the same amount of information about the current state of the game.

2.2 Goal Based Agents and Planning

A video game contains, at its core, a series of goals for the player. Many games have a set of conditions that must be met for the player to win. These conditions provide a set of goals for the player to accomplish in order to win the game. Similarly, AI agents can operate based on a set of goals instead of just a predefined set of actions. These goal based AI agents can effectively consider both the consequences of their actions, as well as how much those actions and consequences align with their goals [36].

Goal based agents and goal oriented planning is discussed by Yue and de Byl [37]. They discuss goal oriented action planning, a “decision-making architecture that defines the conditions necessary to satisfy a goal, as well as steps to satisfy this goal in real time.” This can provide direction for automated agents in how they go about satisfying the goals that they have. The automated agents can be programmed such that, for a given goal, the automated agent would know the steps it takes to complete that goal, as well as any preconditions necessary to complete those steps. As such, the agent is able to come up with a sequence of actions that will lead to the desired goal.

Once the automated agent has a plan, it will then follow the plan until it is completed, or until it no longer needs to be completed. However, the agent can also be designed to continuously assess the current game state and interrupt a current plan if a more relevant or necessary goal is recognized. According to Yue and de Byl, goal oriented action planning provides an advantage, in that every goal that is created does not have a hard coded plan [37]. Instead, the plan to achieve the goal is created dynamically based on changes in the current environment. This dynamic plan creation also provides the advantage that agent behaviors can be formed through the creation of actions and preconditions for those actions, instead of having to program a separate behavior for every agent.

2.3 Learning

Learning is a key part of an advanced artificial intelligence. It is a part of what allows the AI to change and react to the environment. In a game, having an agent capable of learning would allow for more advanced behaviors and possible interactions. Several characteristics of agent learning are discussed by Yildirim and Stene [36]. These include learning that something exists or can be done, learning how much something should be done, learning how to do something, and learning what should be done in a specific situation.

The characteristics of agent learning each have varying degrees of complexity [36]. Learning that something exists can be easy, as all that is needed is for the agent to become aware of it, either by experiencing it or by being told that it exists. Learning how to do something can also be easy, as it can also be accomplished through observation or direct order. Learning how much something should be done can be a more difficult problem, as the same action might need to be done more or less depending on what the action is accomplishing and what the state of the rest of the environment is. Lastly, learning what should be done in a specific situation is similarly difficult for much the same reason. Situational dependency is the key, and accomplishing that can be more difficult, as is discussed in more detail in section 2.3.1.

In addition to these characteristics, Yildirim and Stene discuss four ways learning can be initiated [36]. Learning can occur from feedback, from a command, from observation, and from reflection. Feedback usually comes from the player; the learner is either rewarded or punished based on the action performed. Similarly, commands are also usually from the player. The learner is explicitly told what to do, and as such learns what behavior is expected of it. Learning from observation can come from observing anything similar to the learner, be they the player or other similar automated agents.

To learn expected behavior through observation is more complex in that the learner must distinguish between agents it should be observing and agents it should not, as well as determining what is good or bad without explicit feedback [36]. MimicA makes no assumptions as to which agents it should observe and which it should not. Instead, it relies on the game developer to take any actions that should be observed and pass them to the framework. Lastly, learning from reflection can tie in with the previous section on goal oriented planning. The learner is able to reflect on the goal it had and the action it took. The learner can then determine how well the goal was satisfied based on that action, and determine how useful that action was. This does, however, imply that goals have more than a boolean success or failure state.

2.3.1 Case Based Learning from Observation

Of particular interest for this thesis is learning from observation, as MimicA aims to learn its behavior by observing the performance of the player. In learning from observation, the observed expert behavior is represented by a vector of learning traces which contain a game state paired with an action. We refer to these later as vector-action pairs. Case based learning from observation approaches learning from observation through case based reasoning. Multiple case acquisition strategies for learning from observation are presented by Ontañón and Floyd's [14]. These include reactive learning, monolithic sequential learning, temporal backtracking learning, and similarity-based "chunking" learning.

In reactive learning, the system generates a case for each learning trace [14]. These cases contain the same game state and action as the trace that generated them. These can then be used by the learning agent to determine what action should be taken based on a specific game state. This approach, however, can have issues when it comes to ensuring that certain actions happen after each other, as no action order or temporal information is stored unless it is a part of the game state. Monolithic sequential learning is an approach that attempts to solve that problem by learning a single case for an entire learning trace set. The case contains a game state and a sequence of actions that will be executed in the same order as was in the learning trace. These two approaches have opposite problems. While reactive learning does not maintain any order to actions performed, sequential learning does not have the ability to change based on current situations. As such, neither is ideal for a good learning from observation system.

Temporal backtracking and similarity-based "chunking" both attempt to be the best of both worlds [14]. Temporal backtracking creates cases in almost the same way as reactive learning, with the exception that it adds a link to the previous case.

Instead of retrieving one case to perform, the system retrieves multiple cases based on their similarity to the target state. If they all correspond to the same action, then that action is performed. Otherwise, the system starts comparing previous cases through temporal backtracking with the previous action of the current state, going as far back in time as necessary to determine what action to perform. This can, however, have the drawback of taking more time to find the appropriate action.

While temporal backtracking ties every case to the previous one, similarity-based "chunking" instead attempts to group cases based on how similar their corresponding game states are [14]. Chunks are created for cases where the similarity between their game states is above a certain threshold. Then, when the system queries for an action to perform, the chunk determined to be optimal is returned, and every action in that chunk is executed. This provides a similar benefit to temporal backtracking where actions are more likely to be performed in the same order as they were learned, while at the same time avoiding the longer runtime of retrieving an action. However, "chunking" can have the same, albeit reduced, downside as monolithic sequential learning, in that it is possible not all actions in a chunk need to be performed, even though they were performed in sequence at one point.

2.4 Classifiers

This project makes use of a Decision Tree classifier and a Naive Bayes classifier as two of the three methods for determining which action the companion AI should take. The basics of these classifiers are discussed below, while the specific details for their use in this project are discussed in section 4.2.

2.4.1 Decision Trees

Decision Trees make use of a branching tree like data structure in order to determine what action an AI should make at a given time. Each node in the tree represents some state variable to be examined, while each edge coming off of a node represents a specific value or set of values that the state of that variable can be in. The leaves of the tree are actions that can be taken by the AI.

In order to make use of a Decision Tree, it must first be trained. The training step is what constructs the tree that will be used later, creating the nodes, branches, and leaves. This can either be done before the program is run, if the programmer knows the states that should be examined and actions that can be taken, or at runtime, if the programmer does not know what to include in the tree ahead of time. If done at runtime, the tree may be retrained after more time has passed or more knowledge has been gained, as is the case for this project. This has the advantage of being able to update the tree as new information is gained, as we discuss in section 4.2, however it also has to possible downside of causing delays as the tree is retrained.

After the tree has been trained, a current state can then be classified in order to find the action to perform. This is done by starting at the root of the tree and traversing it, following the branches that correspond with the current values of the different state variables held in the nodes of the tree, until an action is reached. This is the action that the current state has been classified into, and the AI will then perform.

2.4.2 Naive Bayes

The Naive Bayes classifier uses probabilities to determine what action an AI should make. It works by examining action-feature vector pairs. Feature vectors are a collection of state data at a given time, in this case the time that the paired action was performed.

Like Decision Trees, Naive Bayes requires training before it can be used. This takes the form of a collection of action-feature vector pairs that will be examined in order to classify a current state. As with Decision Trees, the Naive Bayes implementation can be retrained as more pairs are generated, in order to have more data to examine and work with.

To perform classification, Naive Bayes looks to find the maximum probability of some action given the current state. This is done by multiplying the probability of the action with the probability of each individual feature of the current state given that action. These probabilities are found using the training data. The probability of an action A is the number of times that action A occurred out of all of the actions which have occurred. The probability of an individual feature given action A is the number of times that the feature occurred in the action-feature vector pair out of every pair containing action A. Following this classification, which generates probabilities for each possible action, the highest of these probabilities can be used to determine the best action to perform next.

2.5 Adaptive Gameplay

The idea of adapting some aspect of a game to fit the player's needs can occur in more ways than just a well done AI companion. A common approach is through dynamic difficulty adjustment. Although the means of performing dynamic difficulty adjustment can be varied, the process is ultimately some variation of monitoring a player's performance and changing some aspect of gameplay accordingly. One such type of dynamic difficulty adjustment is through negative feedback [30]. In games using this approach, the game gets harder as the player does better, and then gets easier again when the player makes a mistake. This is done with the intent of keeping a game at a more stable state. An example might be a game where, as the player gets more points, the game speeds up, thereby making it more difficult for the player to continue getting points. When the player hits an obstacle and loses points, the game slows back down.

While negative feedback is generally seen to increase the difficulty of games, dynamic difficulty adjustment can also be used to decrease the difficulty of games, making them easier for players. An example of this can be seen in the Hamlet system, presented by Hunicke [16]. This system, integrated into the game Half-Life, is designed to examine the current state of the player and the game and possibly offer aid to the player or make it harder for them by reducing health and ammo drops. This could take the form of an increased chance of a health drop if the player is low on health, or an increased chance of an ammo drop if they are low on ammo. This was shown to help reduce the number of times that players died in the game.

Additionally, Hunicke showed that the addition of the Hamlet system to Half-Life increased the enjoyment of players that were previously experienced with the game. This supports the findings of a survey on game adaptivity, which found that current work in game adaptivity produced good results in adapting towards an optimal skill level, as well as positively impacting fun, frustration, predictability, anxiety and boredom [22]. This helps emphasise that creating forms of adaptive gameplay, either through a method such as dynamic difficulty adjustment similar to the Hamlet system, or through a companion AI such as MimicA, can have a positive impact on the games that make use of these methods.

2.6 Real-Time Teammate AI

Real-time teammate AI in video games involves agents that can accomplish a variety of team oriented behaviors, while also allowing for player participation. These include taking into account the behavior, needs, goals, plans, or intentions of other agents on the same team, acting as part of coordinated behaviors, performing actions relevant to shared goals, and prioritize for player participation when possible [23]. It is important that the agent not only works towards the goals of the team, but also allows for player focused gameplay in order to provide more enjoyment for the player. While it is possible to develop agents that complete team objectives, if they do so without involving the player then it doesn't allow for much of a team based game.

Player focused teammate AI can be difficult to accomplish because each player is different [23]. This is the benefit of the real-time component. It allows the AI to develop and adapt to each player's preferences and playstyles. This can be done through the variety of learning and observation methods that were discussed in previous sections. MimicA seeks to do this through a learning by observation method discussed later in this paper.

Chapter 3

RELATED WORK

In this chapter we present a number of pieces of related work to this project. Most prominent among these is the discussion of jLOAF and Darmok 2. However, we also provide a brief discussion of offline learning as well as presenting a few examples of companion AI in previously published games.

3.1 Offline Training

In their paper on learning policies for first person shooter (FPS) games, Tastan and Sukthankar present an approach to improve the performance of bots in FPS games using inverse reinforcement learning [32]. They utilize a finite state machine that causes their bot to switch between one of three different modes, at which point the bot performs a policy lookup based on the current game state.

The policies examined are trained into the program by human players beforehand. As players play the game, the system records sets of states, actions, and rewards, compiling a collection of player demonstrations. These demonstrations are then used in offline training to create a set of policies that the bot will access in game.

While Tastan and Sukthankar attempt to create a more intelligent bot through evaluation of player demonstration, doing so through offline learning of a training set gathered ahead of time inhibits the possible uses. While this approach may work for a FPS game where the number of states a player and the world can have at any given time may be smaller, it may not if applied to a modern role playing game (RPG) or real-time strategy game (RTS). The number of possible player and game states in those types of games is significantly larger, making it so offline training would need to be significantly more extensive. MimicA attempts to avoid this problem through online learning, while the player is playing the game. This aims to avoid missing possible usecases, as well as tailoring the experience more to a single person as opposed to a general audience. And while the large number of possible and ever-changing game states may also be a problem for online learning, previous work such as the TEAM and TEAM2 mechanisms presented by Bakkes, Spronck, and Postma has shown for online learning to still be effective [3].

3.2 jLOAF

A case-based reasoning framework, the Java learning by observation framework, jLOAF, is presented by Floyd and Esfandiari [13]. Their framework aims to aid in the development of agents in different environments, where the agents learn the behaviors they will perform without explicitly being told about necessary tasks or goals. They use case-based reasoning for action determination, and the framework breaks actions and inputs into atomic and complex parts in order to better represent possible inputs to the system and actions to perform.

As a part of the jLOAF framework, preprocessing steps are performed on the cases retrieved thus far. This preprocessing comes in four steps, feature selection, redundancy removal, case base analysis, and case base restructuring. In feature selection, the framework attempts to identify important features in order to optimize analysis and retrieval of cases. Redundancy removal, as it sounds, works to remove duplicate or highly similar cases in order to free up computational or storage space.

Case base analysis doesn't explicitly change the case base like the previous two steps. Instead, it examines the cases retrieved so far and attempts to find areas of the problem space under or over represented to modify what is recorded in future observation sessions. Lastly, case base restructuring simply modifies the way in which the case base is structured in order to expedite case retrieval.

The premise of jLOAF is very similar to the purpose of MimicA. However, while both attempt to create a framework for a general agent that can operate without prior knowledge of the domain, the preprocessing steps that jLOAF has seem to conflict with this. While Floyd and Esfandiari do not discuss how the preprocessing steps are performed, the feature selection and case base analysis steps described seem to require knowledge about the current domain in order to operate effectively or accurately. This could potentially be gained from the user of the framework, however that would require them to put more effort towards the use of the framework. MimicA attempts to avoid this and to require as little from the game developer as possible, in order to provide the developer with a useful framework that doesn't require significant overhead to learn and use.

A more detailed example of how an automated agent using jLOAF would interact with the environment around it is provided in a second paper by Floyd and Esfandiari [12]. They discuss creating an agent with three distinct modules, a perception module, a reasoning module, and a motor control module. The reasoning module is the heart of jLOAF. The module is designed to be used in a wide variety of domains without being altered. The reasoning module will receive input from the perception module and provide output to the motor control module. These other two modules will be domain specific, modified to interface between the specific environment and the generic perception module. This is similar to our approach with MimicA. While we don't have specific modules in the same way jLOAF does, MimicA acts in much the same way as the reasoning portion of jLOAF, taking domain specific information from the game developer, processing it in a generic way, and providing a domain action back to the game developer.

3.3 Darmok 2

Darmok 2 (D2) is a real-time case based planning system for RTS games developed by Ontañón et al. [27]. D2 is a planning system designed to be domain independent, capable of learning how to play RTS games through human demonstration. It combines many of the key concepts already discussed in sections 2.2 and 2.3 of this thesis. D2 uses demonstrations, plans, and cases in order to operate effectively.

Demonstrations in D2 are represented as time, state, action triples, similar to the state-action pairs discussed in section 2.3.1. A key difference is the representation of actions in D2. Since actions in RTS games are not always successful, D2 adds more than just preconditions and postconditions to the actions, including success conditions, failure conditions, and pre-failure conditions. Demonstrations can then be combined into plans consisting of transitions and states. These plans are then stored as cases. Cases also contain episodes, which is an object containing the outcome of a plan when executed at a specific game state. In addition to human demonstrations, D2 requires a set of goals, preset on a per domain basis. It looks for these goals in the plans obtained from the human demonstrations. After D2 has a case base which it will operate off of, when it retrieves a plan from the case base it attempts to modify the plan to fit the specific situation before acting on that plan.

While MimicA does not make use of planning in its current iteration, a system like Darmok 2 can provide good insight into possible future work. More discussion of the possible extension of planning into the MimicA system is discussed in section 8.

3.4 Published Titles

Typically, the avatar presence of a player in a game is a feature of FPS or RPG genre. Most RTS games, including the tower defense genre that *Lord of Towers* is based on, do not have a player avatar. We point to two well-known titles that exhibit some genre-mixing to demonstrate existing use cases: Battlezone [1] and Brütal Legend [11].

Based on the arcade game of the same name, Battlezone is an influential game that experiments with mixing the FPS and RTS genres. The player has an avatar that can enter vehicles and engage in FPS style battle, but the player also controls a base and can give commands, build orders and upgrade orders to the units there. The game received generally positive reviews [24].

Similarly, the 2008 game Brütal Legend, had elements of RPG and RTS mixed together for some of the battle scenes. The player controls the main character, but can also give commands to a number of companions who are partly AI-controlled. Interestingly, it was this mix of genres that is generally considered to be the weakest part of Brütal Legend, leading one reviewer to write:

> But before you know it, you're doing much more managerial work. The on-foot dungeons and one-on-one boss battles disappear, and the rest of the game's big story beats are played out strategically... Your job, instead, is to shuffle like crazy through a host of menus: Send your units to control a tower. Play a guitar solo to buff up your warriors. Load in more units from another menu. Level up your base so you can bring in better units. All I could think was, "This is not what I bargained for" [18].

Kohler's frustration is in part that the RPG+RTS gameplay is too difficult to manage, precisely because the RTS control of the companion units is too much of a distraction from the role-playing battle experience. With the MimicA framework, we can create NPCs with mimicking AI behavior that could eliminate the need to micromanage within the rest of the RTS subsystem.

In addition to these mixed genre games, there are many notable games with one or more companions, for better or worse. These are most commonly seen in RPGs like Elder Scrolls V: Skyrim [4], the Dragon Age series [6], the Mass Effect series [5], or the Dark Souls series [15]. While not always seen as a companion AI, automated teammates in RTS games such as Black and White 2 [20], the Starcraft series [8] and the Warcraft series [7] are also important to pay attention to, as they have much the same purpose. That is, to provide support for the player in their accomplishment of the game's goals.

The companions in the two types of games commonly differ in the amount of interaction the player has with them. In RPGs it is more common for the player to be able to directly give orders to their companions, instructing them to do a variety of things in the game. While the player is able to instruct their companions at times, sometimes the method in which the companion carries out those instructions is not what is desired by the player. MimicA aims to address that problem by developing a companion that performs in the same way as the player, thereby doing what the player desires. However, it is important to note that mimicking behavior may not always be desired. It may be better for the companion to perform a different, complementary, set of actions to what the player can perform. While we acknowledge this, we focus specifically on those types of games where the companion will be performing the same actions as the player and therefore the mimicking behavior would be useful.

RTS games, on the other hand, usually do not allow for players to give instructions to the AI teammates, even if they are on the same team working towards the same goal. On occasion, in games such as Starcraft, the player can request resources from their AI teammates, however they aren't guaranteed to receive them when needed, or at all. MimicA could be used in these types of games to create a better teammate AI that would work with the player to accomplish the goal, while at the same time supporting the player if the situation is right.

Chapter 4

DESIGN

MimicA and *Lord of Towers* are built using the Unity game engine [35] and C#. Specifically we use Unity 2D to create *Lord of Towers*. We use Unity due to the previous familiarity we had with the engine, as well as the initial development overhead handled by the engine. This allows us to spend more time focusing on the development of the MimicA framework. MimicA is built as a series of C# scripts which are then added to Unity objects. *Lord of Towers* then references these scripts in order to integrate with MimicA as described in section 4.3.

Figure 4.1 shows the general flow of MimicA. A player action and current game state are combined into a vector-action pair, which is then stored in the vector-action pair dictionary. This dictionary is then used to create a model, as we discuss in section 4.2. When a companion needs an action to perform, the current game state is provided to the model and the current best action or set of actions is produced. This flow is discussed in more detail in the following sections.

4.1 Action Observation

MimicA is built to interact with a game through observation of actions performed by the player. These can be any action, or possible inaction, a player of the game could make through the normal course of gameplay using intended interfaces. These actions can be anything a developer wants to have in their game. Specific state information about the action (such as where it was triggered) is maintained in order to provide the AI with context. However, details about the exact object that the action was performed on are not maintained for two reasons.

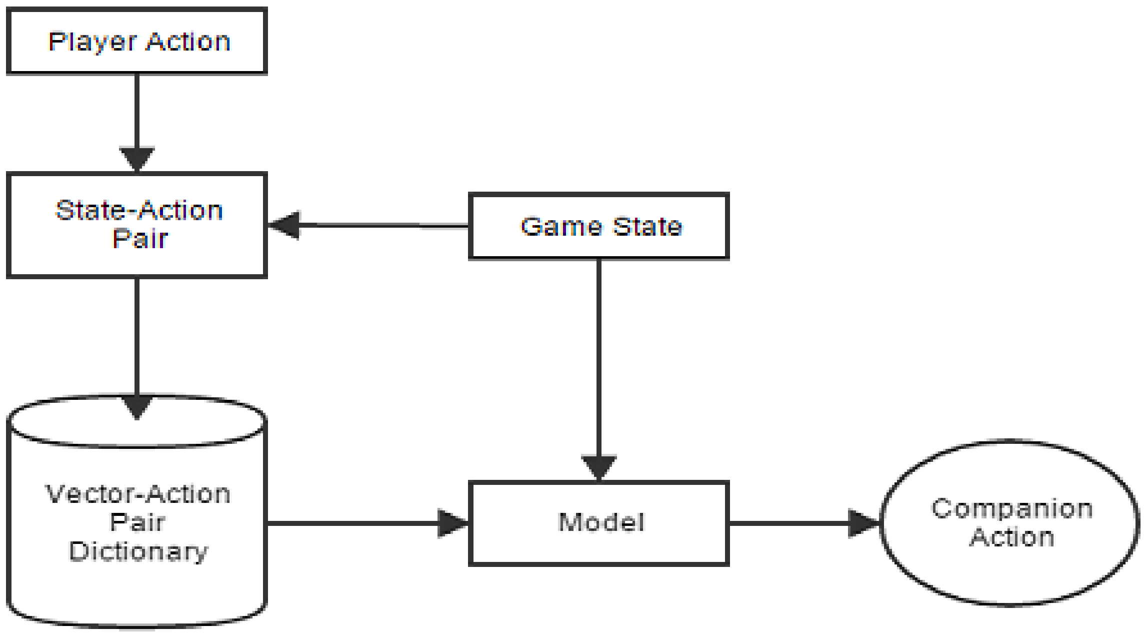


Figure 4.1: General flow for MimicA

First, the exact object may change in the future of the game. For example, an attack action could be performed; however storing the specific enemy attacked is not useful because that exact enemy may not exist the next time an attack action needs to be performed. Secondly, we want the AI to be as generic as possible. It should determine through gameplay what needs to be done and where. So using the attack example again, while the same enemy might still be in the game when the AI determines what to do next, it is better for the AI to attack a different enemy, based on the current state of the game.

Any time an action is performed by the player it is paired with the state of the game at that moment in time, and recorded. The game state is represented by a vector of features designed to capture any and all important aspects of the game at any given point. The game designer, through an interface with the MimicA library, provides this game state, or feature vector. It is left up to the designer to decide what features are important in consideration of actions. The more features that are present in the game state the more data the companion will have in order to make a decision of which action to perform, however this will also potentially increase how long the system takes to retrieve and use the vector. Once the feature vector has been created, and it has been paired with the action just performed, this vector-action pair is stored in a dictionary for later retrieval and comparison.

4.2 Action Determination

When the AI companion needs to determine what action to perform next, it once again creates a vector for the current game state. MimicA then offers three different ways of determining what action to take based on the created vector.

4.2.1 K-Nearest Neighbor

The first, and possibly simplest, way MimicA provides for determining what action to take is through a K-Nearest Neighbor algorithm. MimicA takes the current feature vector and compares it to each of the other vectors stored in the vector-action pair dictionary, generating a list of vectors and corresponding actions most similar to the current vector.

In order to perform this comparison, MimicA first converts the value of every feature in each vector into a number. For features which are already numbers, their value is added to a list. For features that are booleans a one or zero is added to the list depending on whether the boolean is true or false respectively. For enumerations, MimicA takes the integer value of the enumeration and adds that to the list. This means if certain values of an enumeration are not equally different from each other, the game developer must assign non-default values to the enumeration when creating it.

For strings, MimicA either adds a zero or some maximum value to the list. The system compares the value of the string for the current state vector and the compared vector to determine which value to add. If the two strings are the same then a zero is added to the number list for both vectors. If they are different, a zero is added to the list for the current state vector, while a maximum value is added to the list for the compared vector. This maximum value is equal to the largest number in both lists after all other numbers have been set. MimicA performs a similar process for any other non-primitive objects in the vector, using the equals method of the object to determine equality, and adding a zero or a maximum value to the number lists in the same manner as is done for strings. Although it could be useful to require developers to provide a method that returns the value to be used instead of just a zero or maximum value, we opt not to do this for the sake of simplicity and ease of use by the developer.

After the system generates the number lists and finds a maximum value, each of the numbers in both lists are divided by the maximum value. This normalizes the data so that features that naturally are larger numbers because of what they represent in the game do not impact the action determination more than features that naturally are smaller numbers. After this, a third list of numbers is generated where each value in the third list is the difference between the corresponding values in the original two lists. This is the normalized difference for each feature of the two state vectors. Lastly, a root mean squared operation is performed on the list of normalized difference numbers in order to determine a final, single value for the difference between the two state vectors.

This is done for every vector-action pair that is stored in the dictionary. When it has examined all of the stored pairs, MimicA returns an ordered list of the five best vector-action pairs. It is then left up to the game developer to determine how to proceed and what to do with the information. This is done in order to generalize the MimicA framework as much as possible, avoiding imposing restrictions on how actions are implemented. Instead, it is left up to the game developer to determine how to use the list of best actions as they see fit.

4.2.2 Decision Tree

Another method for action determination MimicA has is using Decision Tress. Decision Trees, as previously mentioned, require training before they can be used as a classifier. While decision trees can be trained prior to runtime, this would require knowledge of the features that make up the feature vectors, what possible values those could have, and what possible actions could be performed. This knowledge would be impossible to have however from the perspective of the MimicA framework, as we wanted the framework to be as general as possible, and would have no way of knowing in advance the necessary information for the different games the framework could aid in.

In order to solve this problem, MimicA uses entropy and information gain in order to dynamically build a tree based on the data in the vector-action pair dictionary at the time of training. Entropy is a measure of the purity of a node in terms of number of possible actions, and information gain is the entropy of a parent node minus the average entropy of its children.

For a Decision Tree, each node is a specific feature to compare on. To dynamically determine which feature to use at any given node, we pick the feature that gives the most information gain. Entropy is calculated as the sum over every action of the negative probability of an action multiplied by the log base two of the probability of the action, see equation 4.1, where p_i is the probability of action i.

$$entropy = \sum -p_i * log_2 p_i \tag{4.1}$$

After we calculate the entropy of the current node, we pick a feature and create a set of child nodes based on the possible values of the feature.

For features with discrete values, such as booleans or enumerations, this is easy, where each path to a child node is a specific, discrete value. For features with continuous values, such as numbers and objects this is more difficult. For features that are primitive numbers, MimicA creates children based on the z-score of the value, using the values in the training set to perform the calculation. For non-primitive objects, MimicA requires that game developers implement an interface containing a "decisionTreeBin" method that returns a discrete numerical value then used as the possible children. This is an unfortunate limitation in that it adds additional work for the game developer that might otherwise be avoided.

After the different bins have been created for a specific feature, we place vector-action pairs into each bin corresponding to the value of the feature for each vector. Once each pair has been placed in a bin, we are again able to calculate the entropy of each of the child nodes, and using that information we then calculate the information gain for our current feature. Doing this process for every feature in our feature vector, we find the feature that gives us the most information gain and assign that feature to the current node before recursively performing the same process for each of the children. A stopping point is reached when either the entropy of a node is below a specific threshold, or the current node is a specific number of levels down the tree. At this point, a leaf node is generated by selecting the highest occurring action out of those in the current node.

When the companion AI needs a new action to perform MimicA obtains the current game state vector and then traverses the decision tree, comparing the value of features in the current vector with those stored in the nodes of the tree to reach a leaf node containing an action to perform. However, since the training process involves stepping over all unused features at every node in the tree, this can result in potentially long runtimes in order to construct the tree. Due to this, the decision tree must be manually ordered to train with what is currently in the vector-state dictionary. This means it is the responsibility of the game developer to determine when it is best or how often to train the tree, and to make sure the tree has been trained before attempting to determine a best action. While we do not currently have the data, it would be beneficial to provide the game developer with some form of heuristic in order to aid in determining when training should be performed.

4.2.3 Naive Bayes

The final method MimicA provides for action determination is with the Naive Bayes algorithm. Naive Bayes uses probabilities in order to determine which action is best to perform. Using this algorithm, the probability of an action given some state vector is equal to the probability of the action multiplied by the probability of a feature given the specified action, for each feature in the state vector, as shown in equation 4.2.

$$p(action|vector) = p(action) * p(feature_1|action) * ... * p(feature_n|action) \tag{4.2}$$

This algorithm requires use of a training set, similar to the Decision Tree method. While MimicA utilizes a “train” method that must again be called by the developer to create the training set, the runtime of this algorithm is short. It will generate a probability for every action in the training set and perform a calculation for every feature, so while it will take longer as more actions and features are introduced it will not take a long as the Decision Tree classifier. For the number of actions and features that we had for *Lord of Towers*, the training of the model for Naive Bayes was fast enough that it would not have been noticeable if the system had been trained every time a new action was needed.

MimicA calculates the probability of every action it knows through the current training set given the current state vector. In order to perform this calculation MimicA requires the probability of an action and the probability of each feature given the same action. It finds the probability of the action A as the number of times action A has occurred out of the number of total actions in the training set.

In order to find the probability of a feature F given the action A, MimicA gets the value of feature F from the current state vector and then compares it to the value of feature F for every vector in the training set whose corresponding action is A. The probability of feature F given action A is then the number of times feature F is equal for both vectors, divided by the number of occurrences of action A in the training set. This process is done for every feature in the vector, then the values are multiplied together and multiplied with the probability of the action. The resulting value is the probability of the action given the current state vector.

Due to the possibly large number of features and the possibly large number of vector-action pairs in the training set, it is possible the probabilities that would be generated would be incredibly small, possibly hindering comparison. In order to help alleviate this, we used the product rule of natural logarithms. With this we were able to sum the natural log of each of the probabilities in place of multiplying them, and determine the probability using that sum, as shown in equation 4.3.

$$\begin{split} ln(p(action|vector)) = ln(p(action)) + ln(p(feature_1|action)) + ... \\ + ln(p(feature_n|action)) \end{split} \tag{4.3}$$

After each of the probabilities has been found, the Naive Bayes implementation performs similarly to the Nearest Neighbor method and returns the five best actions

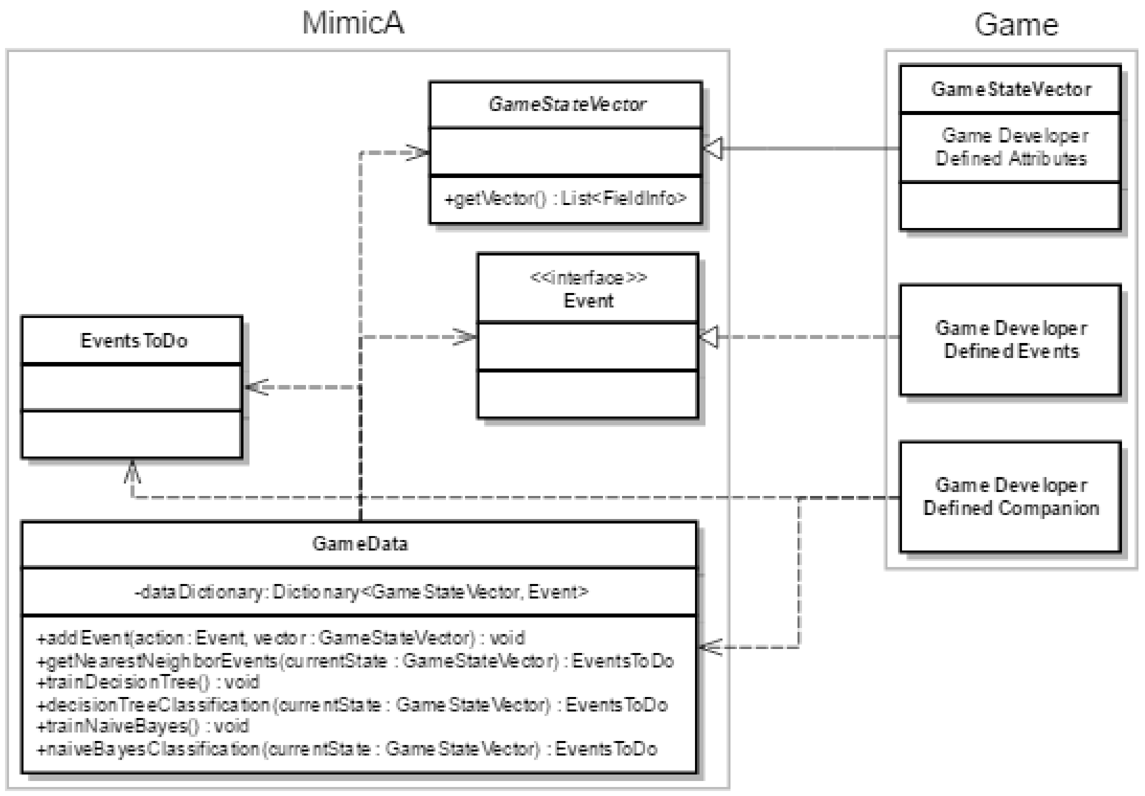

MimicA
Game
GameStateVector
+getVector() : List<FieldInfo>
<<interface>>
Event
EventsToDo
GameData
-dataDictionary: Dictionary<GameStateVector, Event>
+addEvent(action : Event, vector : GameStateVector) : void
+getNearestNeighborEvents(currentState : GameStateVector) : EventsToDo
+trainDecisionTree() : void
+decisionTreeClassification(currentState : GameStateVector) : EventsToDo
+trainNaiveBayes() : void
+naiveBayesClassification(currentState : GameStateVector) : EventsToDo
GameStateVector
Game Developer Defined Attributes
Game Developer Defined Events
Game Developer Defined Companion

Figure 4.2: A class diagram for MimicA and its basic interaction with a game that uses it

to the game developer. Again, it is then up to the developer to determine how to handle those actions.

4.3 API

This section will provide further details into the MimicA API presented to external developers. The main parts that allow MimicA to work are the observation of performed actions and the state of the game. A basic class diagram can be seen in figure 4.2, while more details on the interaction of the classes in MimicA and in implementing games can be found in the rest of this section.

In order to allow MimicA to observe the current state of the game, developers are required to extend the abstract GameStateVector class with their own class containing

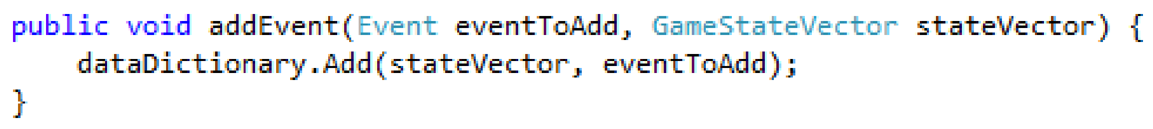
Table 4.1: Sample game state data and values

Parameter Name	Possible Value
lastAction	Build
timeSinceLastAction	10
currentResources	250
closestEnemyDistanceToBase	Distance.Faraway

any relevant information about the game. Each piece of game data should be stored in private instance variables in the developer created class. MimicA is then able to use C# reflection to obtain and use the data stored in these private instance variables. Only important game data should be stored in private instance variables. Any information needed by the developer to gather the data should be left in local variables. This is due to the use of reflection on the part of MimicA.

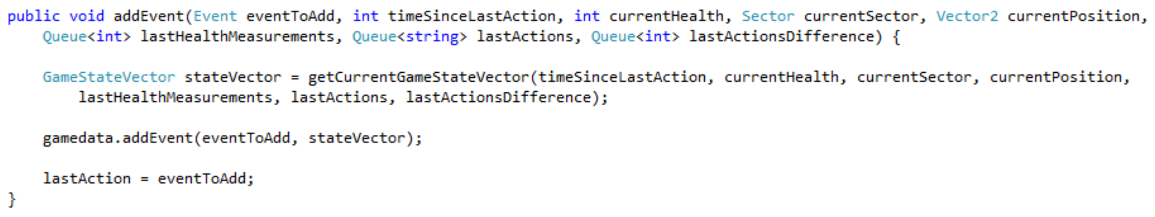
By using reflection, MimicA is able to gather all of the data stored in the created vector class, without having to rely on getting a list of data from the developer. This is also beneficial because some games may have hundreds or more pieces of game data, making it very possible to forget to include one in a returned list. Using reflection makes sure none of the data is missed. Table 4.1 shows an example of some of the data gathered for our Lord of Towers game for use in the game state vector and sample values. It is important to note the values can be anything the developer wants. They simply need to be able to be compared as discussed in the action determination section of this paper.

In order to complete the action side of the vector-action pair, MimicA requires game developers to tie in with the GameData class. This class provides an addEvent method developers are required to call any time an action is performed. This method, shown in figure 4.3, takes in a copy of the action performed and the current GameStateVector that is generated at the time of the actions, adding the pair to the



```
public void addEvent(Event eventToAdd, GameStateVector stateVector) {
    dataDictionary.Add(stateVector, eventToAdd);
}
```

Figure 4.3: The addEvent method MimicA uses



```
public void addEvent(Event eventToAdd, int timeSinceLastAction, int currentHealth, Sector currentSector, Vector2 currentPosition,
    Queue<int> lastHealthMeasurements, Queue<string> lastActions, Queue<int> lastActionsDifference) {

    GameStateVector stateVector = getCurrentGameStateVector(timeSinceLastAction, currentHealth, currentSector, currentPosition,
        lastHealthMeasurements, lastActions, lastActionsDifference);

    gamedata.addEvent(eventToAdd, stateVector);

    lastAction = eventToAdd;
}
```

Figure 4.4: The addEvent method Lord of Towers uses

vector-action pair dictionary. An example of how this is handled in *Lord of Towers* is shown in figure 4.4. The addEvent method in the *Lord of Towers* gathers a variety of information, creates a new GameStateVector, and passes that vector as well as the Event performed to MimicA.

When the game reaches a point where a companion character has been introduced, the developer can request, through the GameData class, an action for the companion to perform. It is important to note that the developer should not attempt to request an action to perform until MimicA has been provided with some previous actions to learn from, in order to make sure that the companion has some information to base its decisions on. The method used depends on the classification method being used. If using the Nearest Neighbor method, the developer makes a call to the getNearestNeighborEvents method, passing the current game state. MimicA then uses the current game state and returns an EventsToDo object containing the five best actions. The details of this action are discussed in the K-Nearest Neighbor section above.

If the Decision Tree or Naive Bayes methods are used instead, the developer must first train the classifier by making calls to the trainDecisionTree or trainNaiveBayes methods accordingly. As mentioned above this isn't something done every time an action is needed, only at certain intervals. The decision regarding how often to train is left up to the developer. Once the classifier has been trained, the developer can make a call to the decisionTreeClassification method or the naiveBayesClassification method, again passing the current game state vector, in order to retrieve the best actions to perform. It should be noted that because of how decision tree classification works, only one action is returned from the decisionTreeClassification method as opposed to the five returned from the other methods.

Chapter 5

CASE STUDY: LORD OF TOWERS

As part of this thesis we developed a tower defense game, *Lord of Towers*, to go along with MimicA and aid in validating the features of the system. As shown in figure 5.1, the player has a physical presence in the game. Although this is abnormal for most tower defense games, it is not unique, and can be seen in games like Dungeon Defenders [34] and Defender's Quest [19]. Another notable difference about the game is the lack of a pre-defined path for the enemies to follow. Instead, the enemies come in from the right side of the screen and proceed to attack the player, moving around anything the player has built. Again, while abnormal, there are other tower defense games that exhibit this same behavior, such as Desktop Tower Defense [29]. A final, notable, difference is after six to ten minutes into the game, the player controlled character will die. While this removes any additional training or information that the companion characters would receive, this is done in order to receive better feedback on how the companions behave without the player around.

The player can select to build and upgrade towers and to build walls and trenches in support of the defense of their base. They can also repair walls, trenches, and towers if they become damaged at any point during the game. Additionally, the player character will automatically attack enemies that come into range as long as no other action is being performed, and they can go heal if they take damage. The actions that are conveyed from the game to MimicA are the build wall, build trench, build tower, upgrade tower damage, upgrade tower speed, repair, go heal, and move actions. The player starts the game with limited resources and more are gained upon defeating enemies. Once the player feels they are sufficiently prepared to start defending they press the start waves button to begin the enemy attack, similar to

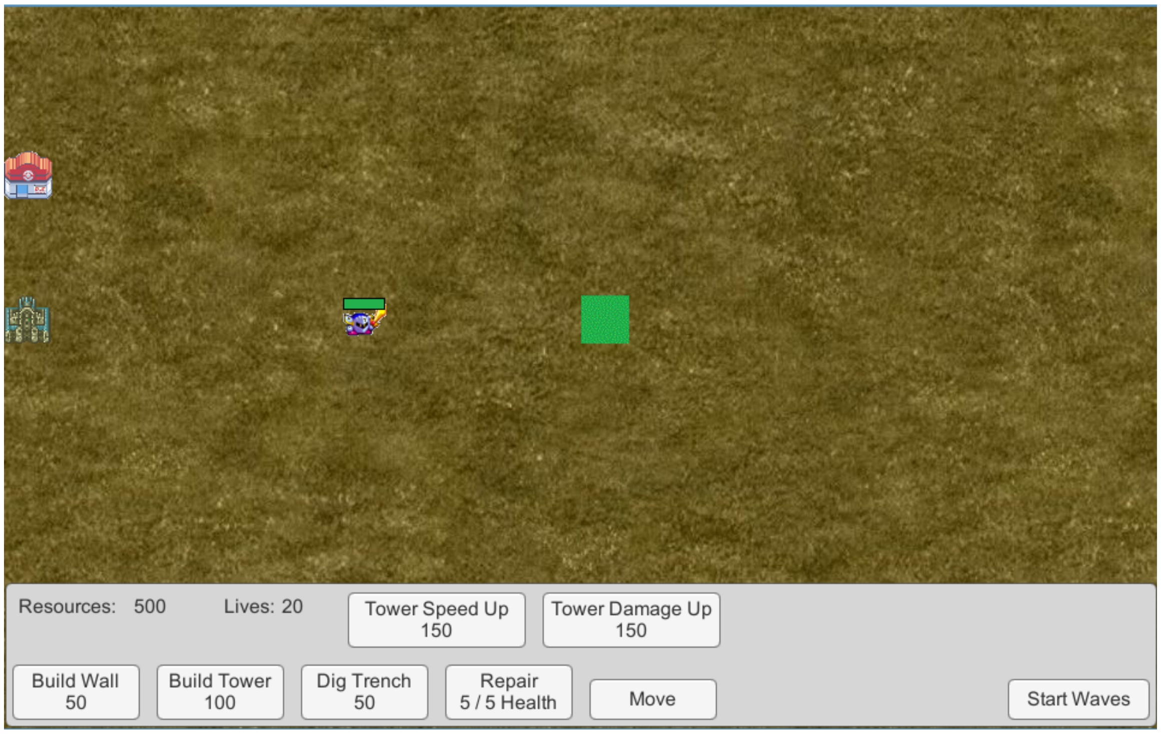


Figure 5.1: The start of gameplay for Lord of Towers

what is shown in figure 5.2.

After the game proceeds for a time, the first companion is introduced, as shown in figure 5.3, and will proceed to assist the player in any tasks the player has performed previously. In this game, building a structure inherently has two actions for the player, the initial build, and then a repair action until the tower is at full health. These two actions are performed back-to-back by the player controlled character as a result of a build request. This sequence will allow the companion to repair buildings even if the player hasn't explicitly used the repair command before. As can be seen in figure 5.3, at the time the first companion is introduced, the countdown before the player dies starts, and a timer appears.

Additionally, in figure 5.3 a prompt is shown of the companion asking the player before performing an action. This is to avoid the companion spending all of the player's resources if the player intends to use the resources for something. This does,

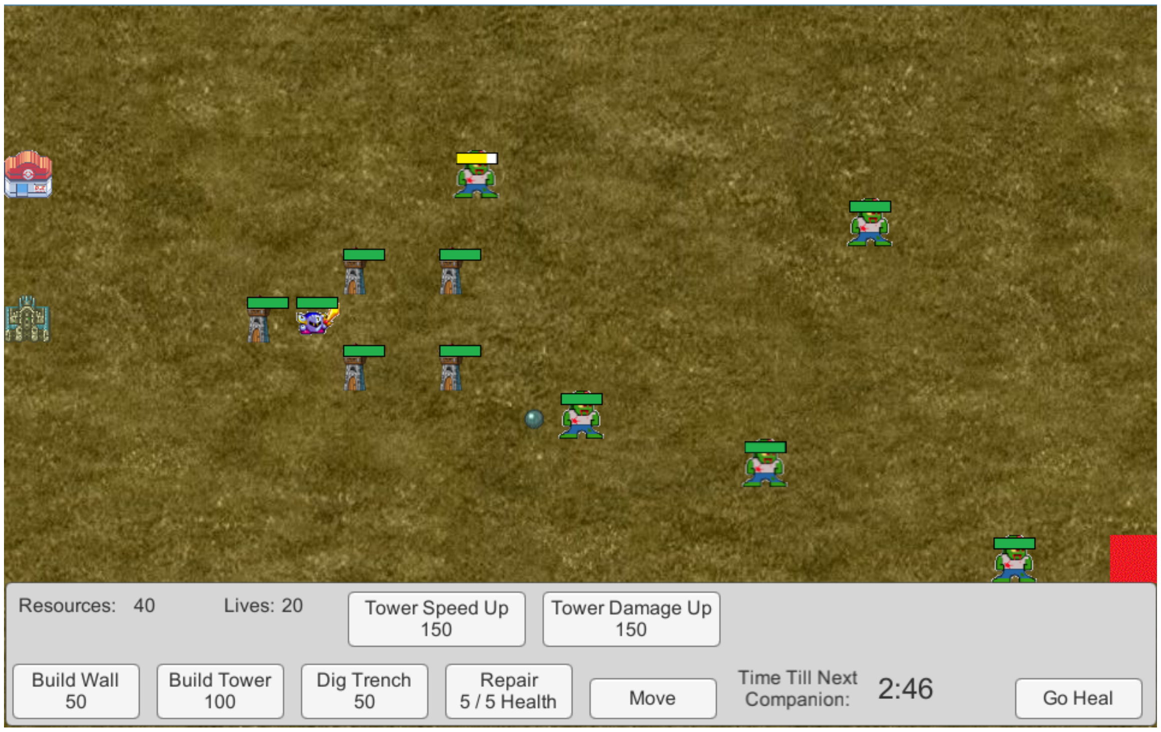
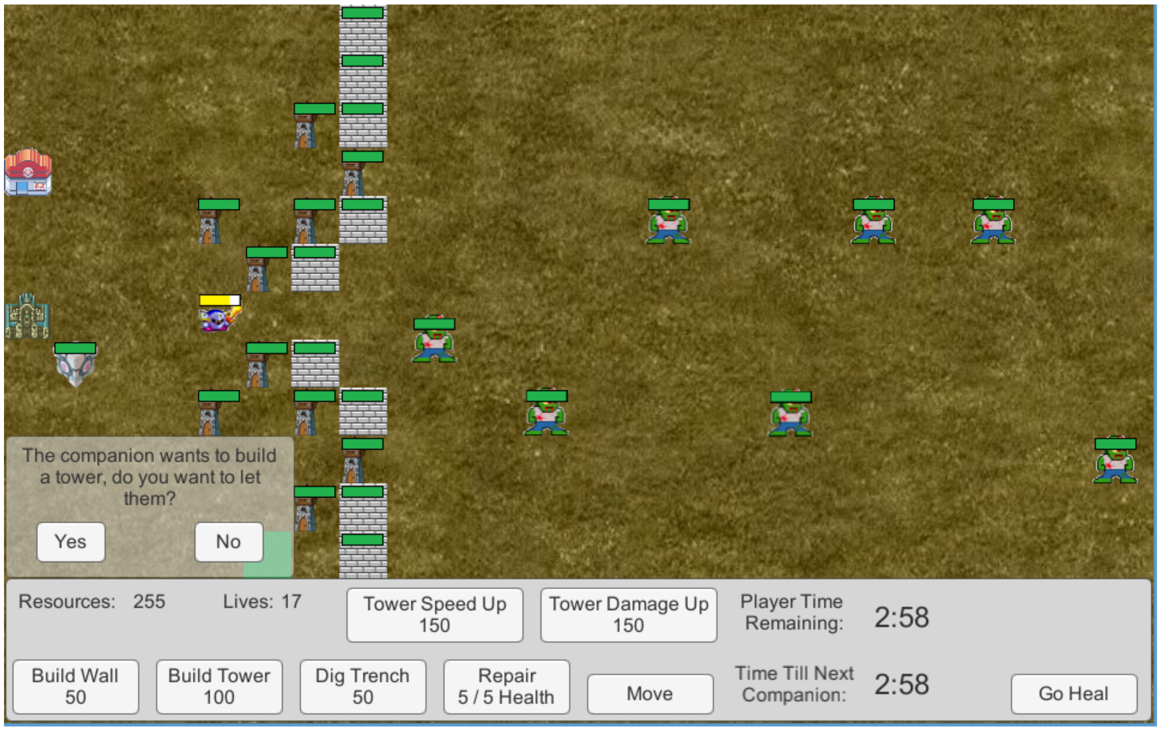


Figure 5.2: The first wave of enemies


The companion wants to build a tower, do you want to let them?

Yes No

Figure 5.3: The first companion is introduced

however, highlight a problem that exists in MimicA. While MimicA is designed to take action based on the actions the player has previously performed, it does not have a way to take into account a player's plan, possibly causing conflict with the player. In the case of *Lord of Towers* this prompt also serves for additional training for the companion. If the player selects that the companion can perform the action they are requesting, a new vector-action pair is generated based on the current game state and the action the companion is performing, and that pair is added to the dictionary, effectively reinforcing that action for the companion.

After three more minutes, the player dies and a second companion will join the game, as shown in figure 5.4. This companion will operate based on the same stored data as the first companion, however the two of them are able to perform independently, acting based on whatever action makes the most sense for them when they need another action to perform. While this may be the same action, such as repairing a tower at the same time, they will also perform independent actions. After three more minutes, a third and final companion will join the game, operating the same as the previous two.

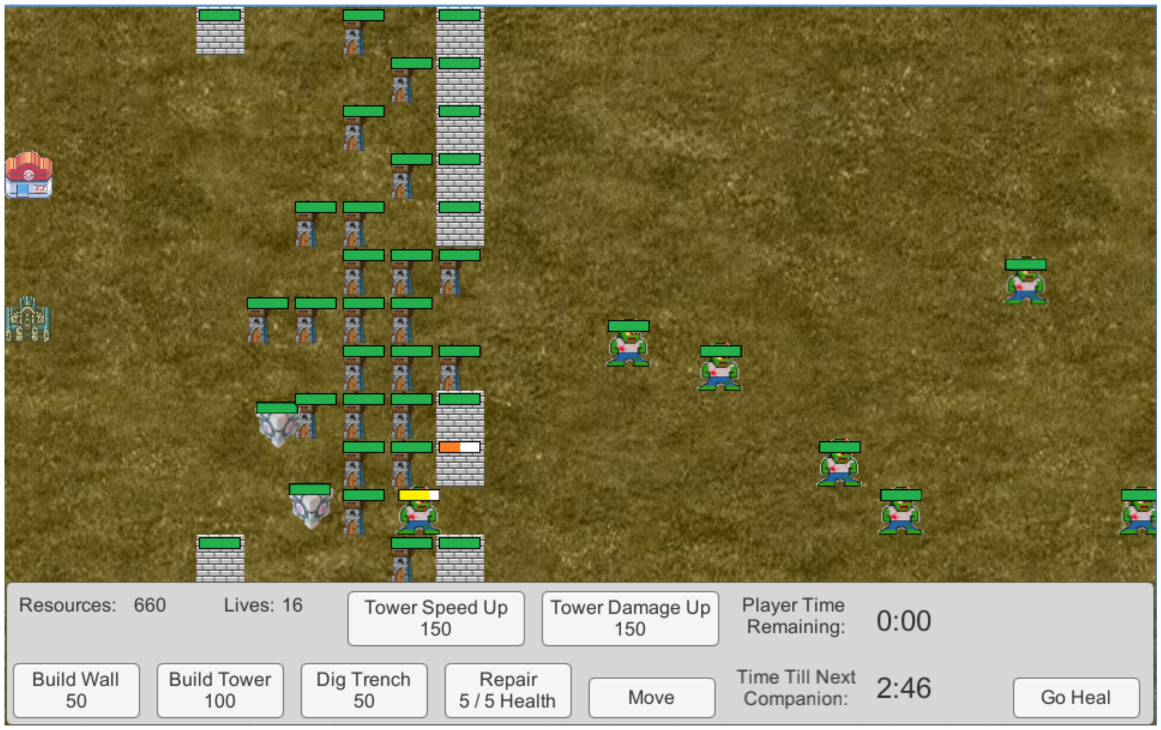


Figure 5.4: The player dies and a second companion takes its place

Chapter 6

USER STUDY AND RESULTS

In this chapter we present the process that we undertake in order to validate the performance of the MimicA system. Additionally, we present the results of the study we performed as well as a discussion of the results.

6.1 User Study

In order to test the effectiveness of the MimicA framework we asked 30 people to play Lord of Towers and answer a survey about their experience. The participants in the study are all in college or graduated from college within the last year. They were found through the graduate program at California Polytechnic State University, through the Study Session program at the same school, or are friends of one of the researchers.

To begin, participants receive a set of instructions on where to obtain and play the game, some details about the game itself, and some general information about the study. Additionally, participants receive instructions on which type of the game to play. While the participants are not told what the types meant, each type corresponds to one of the three possible classification methods MimicA makes use of, as discussed in section 4.2. We evenly test each of the three classification methods in order to determine if one of the methods is perceived to produce better companion behavior. However, as the three classification methods perform the same function, we do not expect there to be a significant difference in results among the three methods. The participants are told nothing about the companion other than it will help them in the game. This is done in order to avoid biasing the participants about what the companion does and receive more accurate feedback about the perceptions of the companion's performance. The full instruction message sent to participants can be seen in appendix A.

After participants finish playing the game three times, we ask them to take an online survey about their experience. The survey includes questions about both the game and the AI companion, and includes both free form responses and multiple choice questions. The full survey can be seen in appendix B, and some of the questions and responses will be discussed in more detail in the following section.

As a part of the analysis of the participants' responses, we code one of the free form answers we receive. The question is "How do you think the companions were programmed?" We ask three coders, individuals familiar with the project, to take the responses given by the participants and code them as one of four possible categories. These categories, as well as a sample response that fell into each category can be seen in table 6.1. If two of the three coders agree on a code for a particular response, we count that as a true response in that category. Out of the 30 responses we received, a category was unanimously agreed upon for 18 of these responses, while a category for each of the other 12 responses was agreed upon by two of the three coders. The three coders never produced three separate codes for the same response.

6.2 Results

One of the main things we hoped to see in our feedback was if people were able to recognize that the companion was performing actions based on what the player had done before. As such, we took great care in making sure little information about the game and companion was given ahead of time, and that the questions of the survey are organized in such a manner as to not reveal the companion's behavior too early.

Towards that end, our first question asks if the participant has played the game

Table 6.1: Coded categories and a corresponding sample response

Code	Code Category	Sample Response
1	They built things regardless of what else was going on	"They appear to move and build at random"
2	They do what is needed based on what else is going on, but don't rely on player behavior	"Finite state machines"
3	They mimic the player or were effected by player behavior in some way	"To replicate what the user is/has been doing"
4	Other	"No idea"

before. As part of an early prototype we had members of the Game Development Club and the Interactive Entertainment Engineering class at Cal Poly playtest the game. This question was present to make sure we could exclude any responses from prior participants. However, it is possible the wording of the question caused possible issues with this response. One of the participants asked if the question was intended to ask if they had played the game *ever*, or rather had they played the game *before taking the survey*. In the first case their answer would be no, but in the second case it would be yes. We later changed the wording of the question to specify that we were asking if they had played the game before this study. Prior to this change being made, five of the 30 participants indicated that they had played the game before. However, we believe that no one who had been given the game up to the point where we changed the wording of the question had in fact played the game as part of our earlier prototype. After we changed the wording of the question none of the participants indicate they had played the game before.

The next few questions of the survey are intended to elicit feedback about how

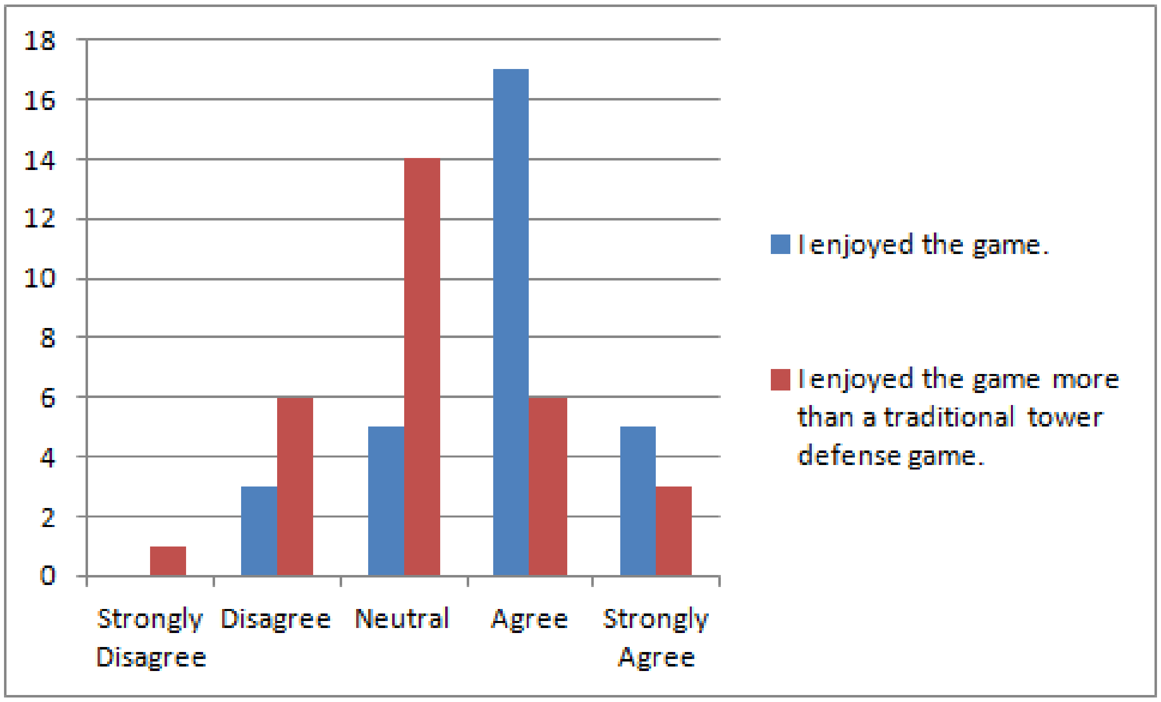


Figure 6.1: Responses for our 30 participants with regards to how much they enjoyed the game

players felt about the game itself, as well as how familiar they were with the tower defense genre. We ask users how much they agree with three statements: "I enjoyed the game", "I enjoyed the game more than a traditional tower defense game", and "I am familiar with other tower defense games." The possible answers range on a five-point Likert scale from "strongly disagree" to "strongly agree." The results are shown in figures 6.1 and 6.2. As figure 6.1 shows, a majority of the participants enjoy the game. However, on average, participants are neutral about enjoying the game more than a traditional tower defense game.

Additionally, while a majority of participants are familiar with the tower defense genre, some felt they were not, thereby possibly impacting their answers. We also examined this question by sorting the answers by classification method in order to better understand if a particular method might be biased more in regards to familiarity with the genre. The results can be seen in figure 6.3. Of the three methods,

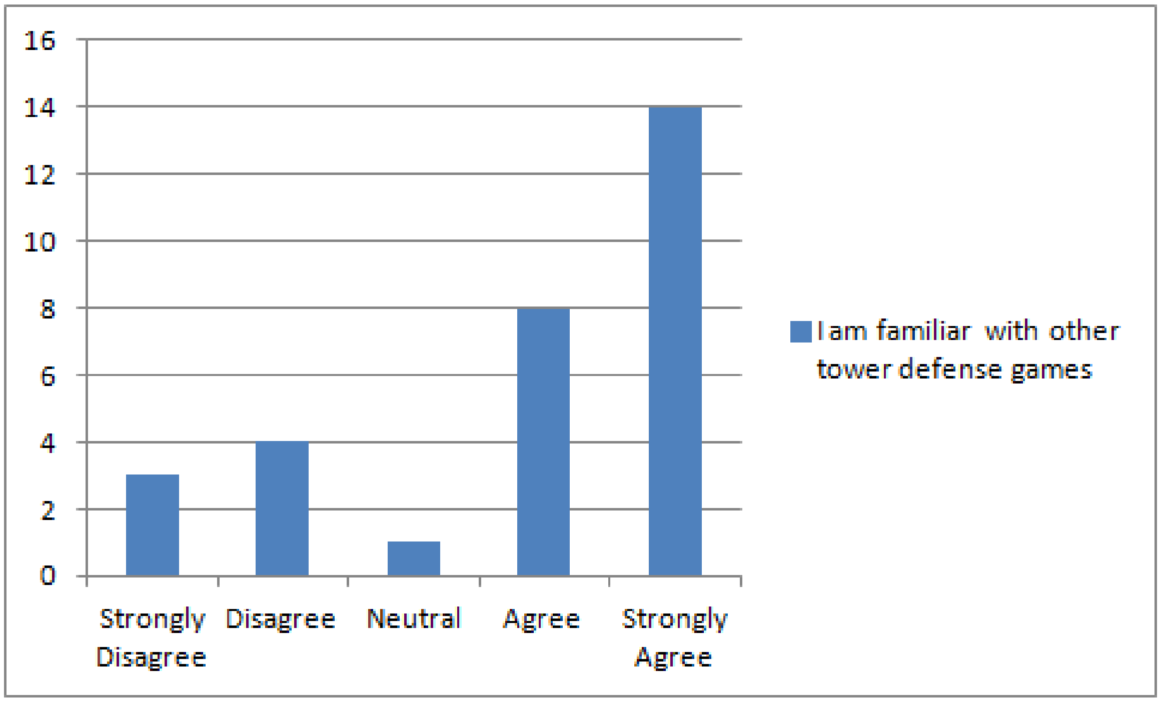


Figure 6.2: Responses for our 30 participants with regards to their familiarity with tower defense games

Decision Tree only had one participant who was not familiar with the genre, while both K-Nearest Neighbor and Naive Bayes had three.

Next, the survey has questions that begin to focus on the companions in the game. First, we ask participants, "How do you think the companions are programmed?" As mentioned in the previous section, this is a freeform question, the answers of which are coded into categories found in table 6.1. The results of this coding can be seen in figure 6.4. This was interesting, because even though a good number of participants recognized that the companion was doing things based on the player's behavior, or at least that it was responding to some part of the game state, an equal number of the responses could not be categorized, usually with answers along the lines of "I don't know." This could be in part due to the participants that were not necessarily game developers or didn't have a programming background.

We further break this question down based on classification method, as shown in

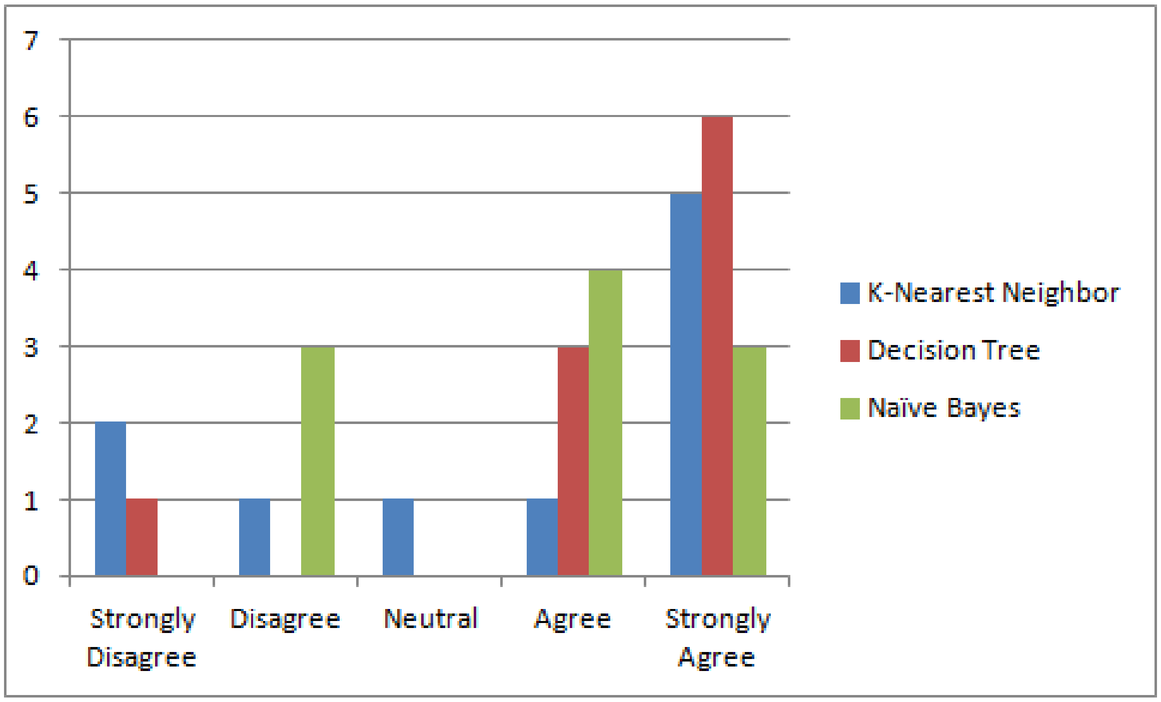


Figure 6.3: Responses for our 30 participants, 10 per classification method, with regards to their familiarity with tower defense games separated by classification method

figure 6.5. The participants who recognized the companion was performing actions based on the player's behavior the most were using the Naive Bayes classification method, however this method also had the most people who couldn't be categorized. K-Nearest Neighbor had the most participants who recognized either the companion was performing actions based on the player's action or based on some other part of the game state, as well as the least number of participants whose responses could not be classified.

The next question on the survey begins to address the actual behavior of the companion, asking participants to indicate if they noticed the companion doing any of a number of things. The possible options, as well as the responses, can be see in figure 6.6. When directly asked, 22 of the 30 participants indicate noticing the companion performing similar actions to themselves. Additionally, 17 of the participants felt the companions were performing actions useful to them. This number is lower than

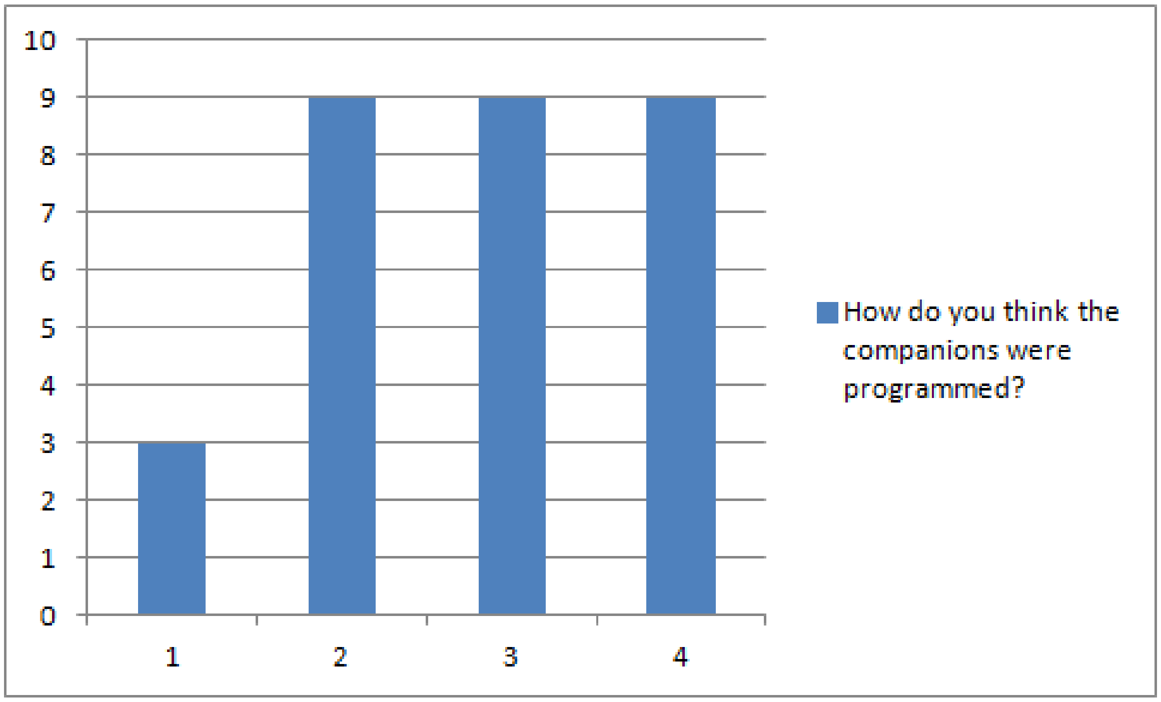


Figure 6.4: Coded responses for freeform question "How do you think the companions are programmed?" (1) They built things regardless of what else was going on. (2) They do what is needed based on what else is going on, but do not rely on player behavior. (3) They mimic the player or were effected by player behavior in some way. (4) Other.

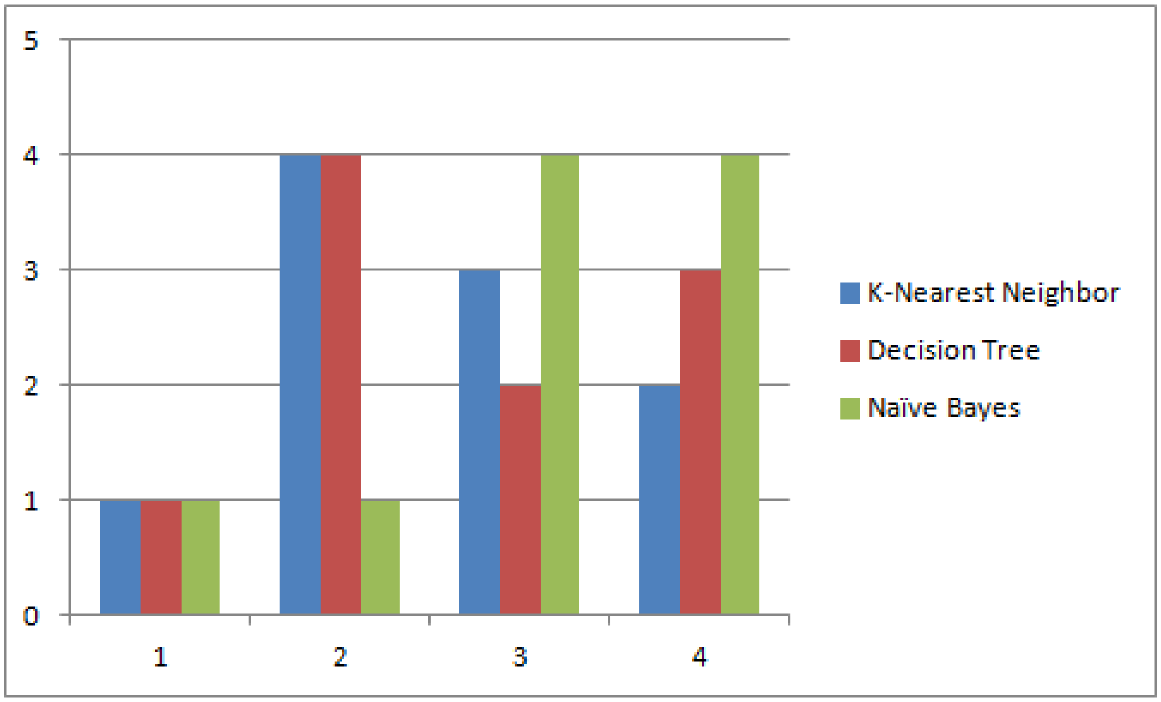


Figure 6.5: Coded responses for freeform question "How do you think the companions are programmed?" separated by classification method

would be desired. Since MimicA aims to follow player behavior, especially in a tower defense game the goal would be for the companion to always perform an action seen as useful to the player because it is an action the player would also do.

The results for this question when separated by classification method can be seen in figure 6.7. All 10 of the participants using the Decision Tree classification method indicated that they noticed the companion performing similar actions to themselves. Naive Bayes had the worst response for this category with only half of the participants noticing the companion performing similar actions to themselves. Both the Decision Tree method and the Naive Bayes method had six participants, and the K-Nearest Neighbor method had five participants, who felt that the companions were performing actions useful to them.

We next ask the participants if they ever wished the companions would do something they were not. If they indicated yes we asked what they wished the companions

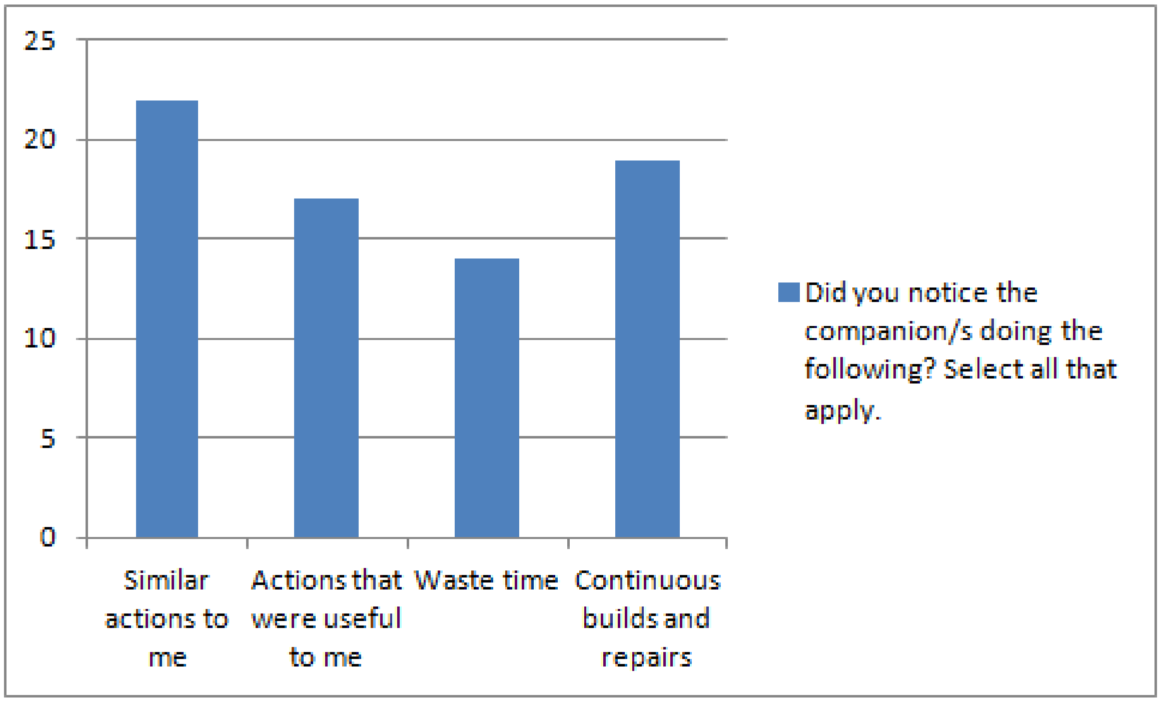


Figure 6.6: Participant responses when directly asked about various companion behavior

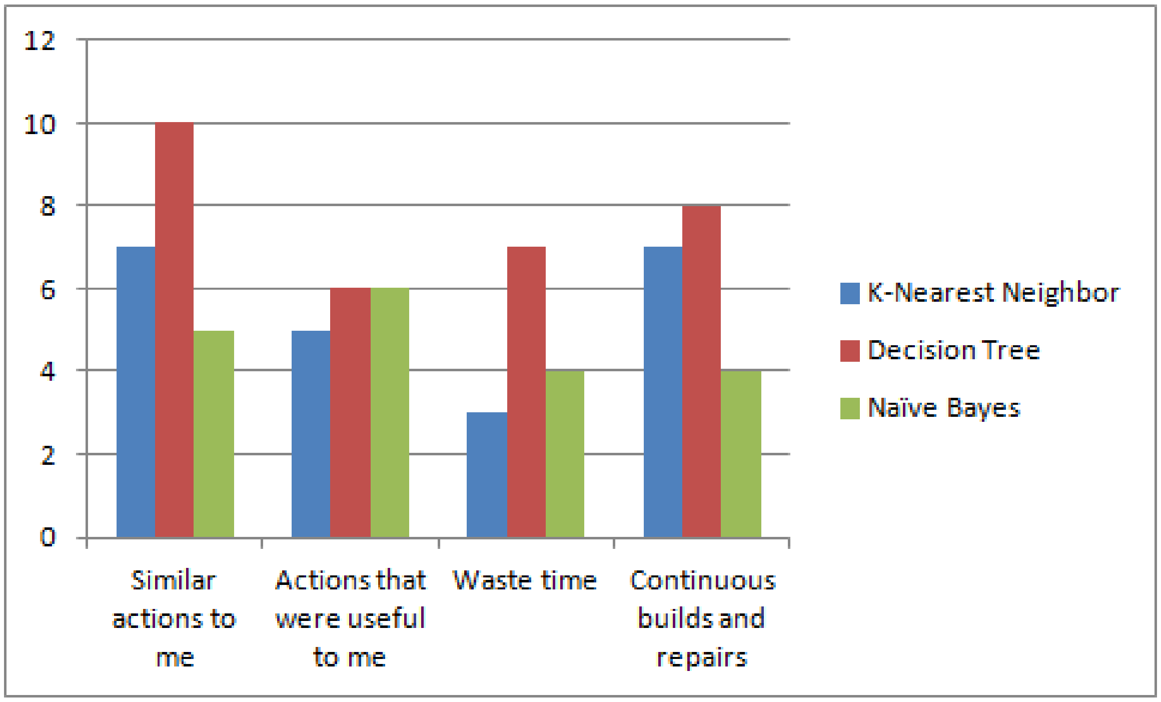


Figure 6.7: Participant responses when directly asked about various companion behavior, separated by classification method

would have done. 23 of the participants indicated yes, that they wish the companions would do something that they were not. While many of the responses to the follow up question don't necessarily indicate whether the problem was with the game or with the framework, a majority of the problems could likely be addressed on the side of the game. Most of the responses were along the lines of "don't disrupt the path that I created", "don't fill in the gaps that I leave to make a maze", or "companions needed to repair buildings that they just built."

The first two types of responses don't necessarily indicate there is anything wrong in terms of what the companion is selecting to do, but rather where it is selecting to do it. This could be solved with better interpretation on the part of the game that, once an action has been determined by the MimicA framework, tries to better understand the strategy the player is using and to follow the same strategy. This could, however, also be addressed inside MimicA with the introduction of more types of planning, which is discussed more in section 8.

The third type of response, indicating the companion is not following up on an action it just performed, is a problem discussed in section 2.3.1. It is a known problem in case based learning that, depending on the method used, actions performed by the agent may not be in the same temporal order as those performed by the expert. However, this could also be solved in the game. As it stands in *Lord of Towers*, build and repair are two separate actions, leading to the observed problem that companions don't always repair a tower to full health right after they build it. While we don't want to remove the repair action altogether, it would make sense from a game development standpoint to immediately follow the build action by a repair action for the companion, just as it works for the player. This would not prevent other companions or the player from helping to "repair" a newly constructed building to full health, but it would result in more fully constructed buildings. So although this is a known issue for case based learning agents, this could likely be solved on the game side, as opposed to relying on a solution on the framework side. However, this would put more of a burden on the game developer to handle how actions like these would interact with each other in a different game. Alternatively, if a solution could be found on the part of the framework, it could possibly open up a wider range of dynamic behaviors where the companion decides it doesn't need to finish building something because there is a more urgent need elsewhere, and instead will come back to finish the building after.

Next, participants are again asked to rate their agreement with a number of statements, this time focusing on the companion. The statements were "The companion/s was/were useful to me", "The companion/s would protect me", "The companion/s was/were performing actions that I would do", and "The companion/s was/were learning from the actions that I was performing", again answering on a Likert scale. While two of these statements aim to gather much the same data as was discussed above and presented in figure 6.6, the final statement is the most important of the group. We are now directly asking participants if they noticed any form of learning behavior based on the player. The results of this question can be seen in figure 6.8, and it is important to note that only 29 of our 30 participants answered this question. When directly asked, just over half of the participants either agreed or strongly agreed that the companions were learning from the actions the player was performing. Only six of the participants felt the companions were not learning from the actions the player was performing.

We focus more on whether participants felt the companions were learning from their actions by separating the results by classification method, as can be seen in figure 6.9. The Decision Tree method and the Naive Bayes method had the best response, each having six participants per method who either agreed or strongly agreed the companions were learning from the actions the player was performing. K-Nearest Neighbor only had four participants who either agreed or strongly agreed, and was on average neutral.

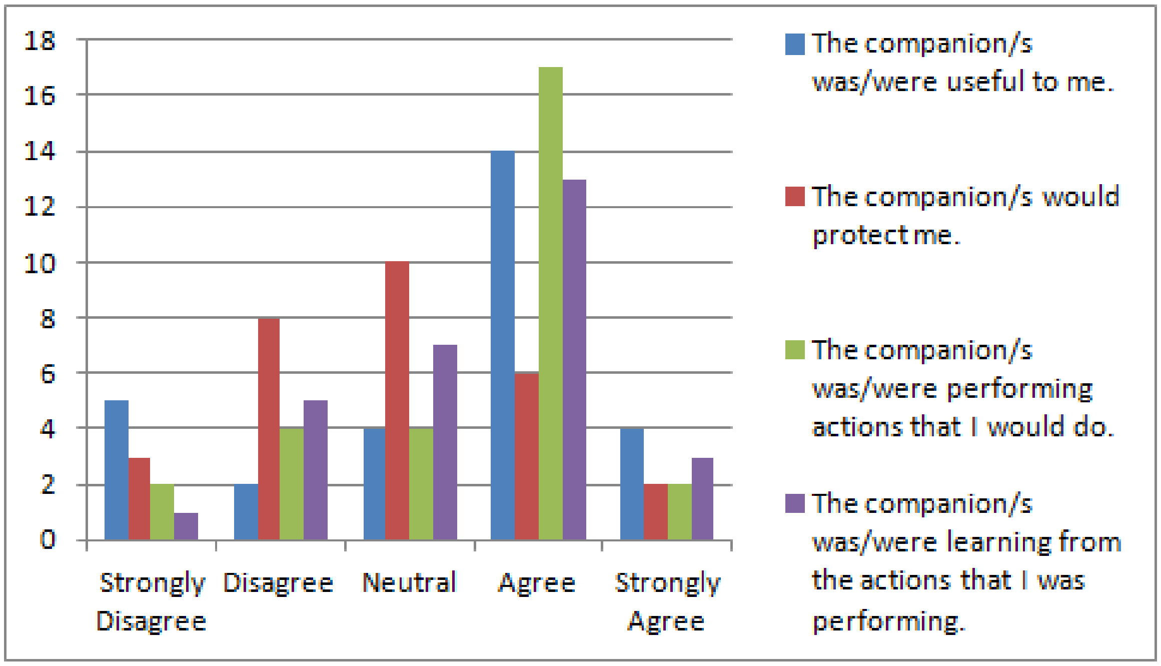


Figure 6.8: Participant responses regarding companion behavior

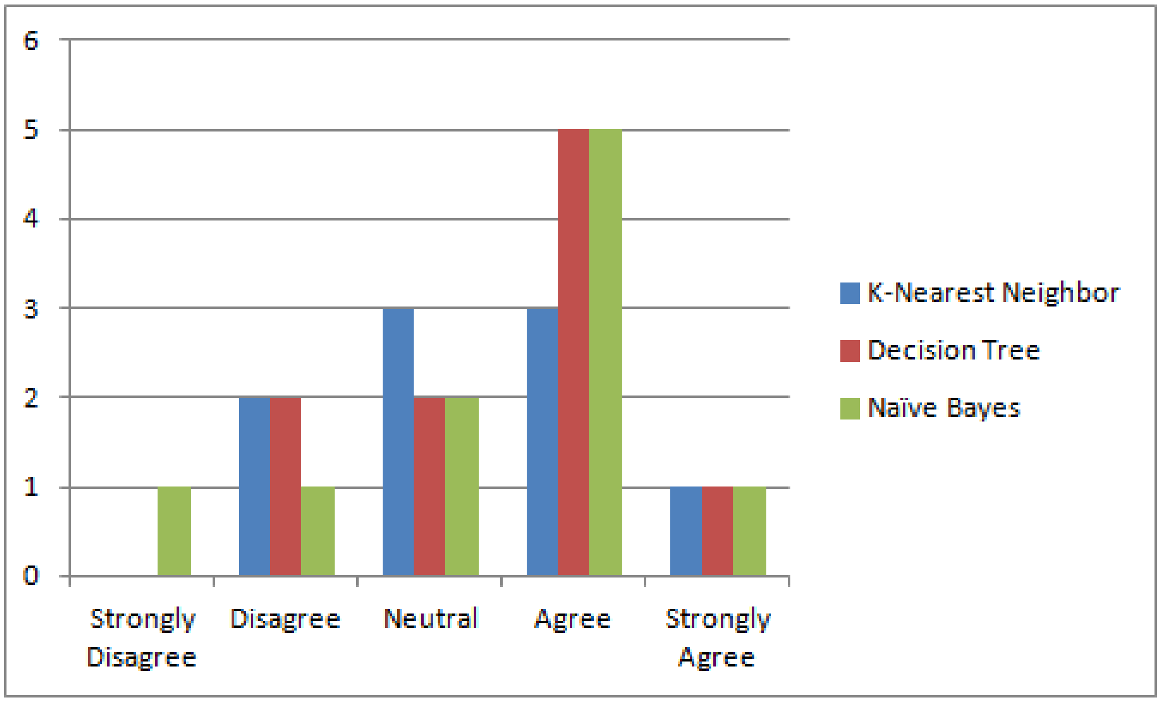


Figure 6.9: Participant responses for agreement on the companion learning from actions they were performing

At this point in the survey, we tell the participants that the companions are programmed to learn from the player's behavior, asking the participants to take the perspective of a game developer to answer the question "if this AI was available as a library/plugin that you could use to aid in development of your game, would you use it," and why or why not? This question garnered mixed results, most likely due to the broad range of participants that were in the study. 21 of the participants said they would use such a plugin, eight said they would not, and one didn't provide an answer. It may be important to note, however, that two of the participants who said they would not use it followed up by saying it was because they were not a game developer.

Of the 21 that responded they would use a plugin like this, many of the follow up responses indicated they would use it because it would take away some of the load for the game developer or it would help expand upon the possible strategies available in the game, both features that MimicA aims to provide. One of the main points of opposition to a companion like this was that a companion which simply mimicked the player isn't always desired. It might be better for the companion to provide a support role, performing actions that aid the player but don't directly copy them. This is a valid concern for a framework like this, and why it might be less useful depending on the game environment.

Chapter 7

CONCLUSION

In conclusion, we present in this chapter a number of challenges that were faced in the development of this project, as well as a summary of the contribution which this project makes.

7.1 Challenges

In this section, we discuss several challenges encountered while developing the MimicA framework. These are: the general issues concerning frame of locality and relative space designation, idle waiting behavior, external requirements expected of a companion AI which are not necessarily learned behavior (example, avatar protection), and finally build/repair overlap.

7.1.1 Frame of Locality and Relative Space

A major problem encountered in the development of MimicA is determining how to tell the system where an action took place. While *Lord of Towers* uses a built in grid to determine where characters can move and where buildings can be placed, requiring MimicA to work with a grid system would be too restrictive. This would especially be seen when a companion utilizing MimicA determines what action needs to be taken next. If MimicA operates by using a specific grid, then the companion would attempt to perform the action in the exact same space every time, which would likely not be helpful for the player.

As an alternative approach, we opted to use a relative space system in the game. Each grid square is located in one of six sectors and sector information is stored as part of the vector-action pair. This allows the companion to know the sector an action is performed in, and then use some knowledge programmed into the game to determine where in that specific sector an action should be performed. This allows for much more general behavior than would be seen otherwise, however, the behavior is handled by the game, not by MimicA. As such, it would be equally possible for a developer to not even include a position in the vector-action pair, and simply determine where to perform an action when needed.

7.1.2 Idle Waiting

Another problem that quickly becomes apparent while developing MimicA is the amount of time companions spent waiting. "Wait" is included as an action in Lord of Towers because the player will not always be doing something. A player could spend time waiting to determine what to do next and we want this behavior to be reflected by the companion.

Unfortunately, the companion performed this wait action much more frequently than we expected. This was alleviated somewhat by increasing the duration of player idle time necessary before generating a wait action, however it still did not completely solve the problem. Another idea we considered was to make it so there wasn't a wait action at all, and instead make it so the companion only waits if all the actions MimicA returned didn't make sense to do at the time (e.g. not enough resources to build, no damaged buildings to repair, etc.) Ultimately, as MimicA attempts to impose as few restrictions as possible on the actions a game can have, the framework provides no limitations to prevent large numbers of wait actions from being performed. This is instead left up to the game developer to handle, if wait actions are even relevant to the game.

7.1.3 External Requirements

While MimicA is designed to provide a game developer with actions for a companion to perform based on the current game state, there may be times when the developer wants the companion to do one thing no matter what. An example of this could be having the companion move to protect the player's character any time they are being attacked. Determining how to handle this and attempt to integrate the behavior into MimicA is a problem during development of the framework.

Certain behaviors like this could be tied into the game state vector. For example, in *Lord of Towers*, if the player were to move to assist a companion being attacked, that action would be recorded and paired with the current game state, and the companion would learn from that and possibly perform similar behavior in the future. However, this is reliant on the player performing the action first. Ultimately we decided this was the behavior we desired from MimicA. The intent behind the framework is to provide actions to perform based on learned behavior from the player, so if the player hasn't performed an action, then the framework won't say that an AI should either. If a game developer wants a companion in their game to act with some default behavior in certain situations, it is up to them to provide that overriding functionality before performing the action suggested by MimicA.

7.1.4 Build/Repair Overlap

Lastly, for *Lord of Towers* we want to separate the creation of a building into two actions, a build and a repair. This means a building is placed at minimal health, and then is "repaired" up to the maximum health for that building. We want this functionality in order to allow for situations where a player or companion can start construction of a building, and other friendly characters could come over and assist with finishing off that building.

While the player is designed to immediately transition from initial construction to repair, the companion is not. Instead, if the companion receives a build order from MimicA it will complete that build order and then request the next action to perform from the framework. We noticed right away that MimicA was not always instructing the companion to repair the building it just constructed, opting instead for some other action deemed more relevant. In an attempt to remedy this, we added more features to our game-state that focus around what actions are more often performed after others. While this did help, it didn't completely solve the problem. However, as discussed in the results section, we feel this problem is not a significant hindrance to MimicA. While it is a problem that exists in many case based learning systems, it can be remedied, if not solved, in the game itself, and therefore we do not attempt to change the observation system in order to compensate.

7.2 Summary of Contribution

In this paper, we present the MimicA framework, a system for governing the behavior of companion AI. We posit that certain games can benefit greatly from an open framework designed to fully automate the companion AI for those games where it makes sense to have companions learn behavior through actions of the player. The challenge is for the task assignment system to intelligently choose the right companion and assign it the right task at the right time. While this study presents three different classification methods, this is done for the sake of testing, in order to see if one method is perceived to be better than the others. Ultimately the framework would likely be composed of only a single classification method.

Our user study on *Lord of Towers* suggests that such a framework can be easily used to showcase games with a new form of companion AI for video games. This companion will perform alongside the player and operate by learning from the player without explicit teaching by the player. Out of 30 participants, a majority agrees that the companions are doing useful things. As expected, there is not a significant difference between the number of participants that find the companion useful when separated by classification method. Further, 16 of the 30 agree that the companion learns from the player while six disagreed with this (the remaining participants were neutral on the matter). When separated by classification method, more participants who use the Decision Tree or Naive Bayes methods indicate that the companion learns from the player. The results act as proof of concept for MimicA. Of the three classification methods, participants who use the Decision Tree method generally have the most positive response. Users generally understand what companions are doing and find them helpful, supporting our belief that this is a useful framework to continue to explore.

Chapter 8

FUTURE WORK

For future work, it would be beneficial to attempt to integrate MimicA with an already functioning game. While *Lord of Towers* was good as a case study for the framework, too many of the problems that arose in development or were brought up in our study could have been the result of the game, not the framework. As such, using the framework with an existing game would be beneficial to clear up some of the possible issues. Additionally, integrating with a previous game would potentially allow for a more objective way of determining companion performance. It would be good to objectively measure companion performance by initially training them and then letting the game run to see how long the companions can last on their own. Doing so in a already balanced game would provide much better feedback than attempting to do so in *Lord of Towers*.

As mentioned in section 3.3, extending MimicA to take advantage of planning would be particularly useful. In its current state, MimicA has no form of planning incorporated with the methods in which it determines what action should be performed next. Adding a planning system to the framework would allow MimicA to perform more advanced action determination, thereby enhancing the performance of the framework.

Additionally, it would be beneficial to detach MimicA from Unity. While developing *Lord of Towers* in Unity made the most sense based on time constraints and prior knowledge, it restricts the possible audience for the framework. Being able to separate MimicA away from Unity into just a C# library, or even be able to implement it in other languages, would be highly beneficial towards expanding possible use cases.

BIBLIOGRAPHY

[1] Activision. Battlezone. [PC Computer], 1998.

[2] Atari Incorporated. Pong. [Arcade Game], 1972.

[3] S. Bakkes, P. Spronck, and E. Postma. Best-response learning of team behaviour in Quake III. In *Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 13–18, 2005.

[4] Bethesda Game Studios. The Elder Scrolls V: Skyrim. [PC Computer, Playstation 3, Xbox 360], 2011.

[5] BioWare. Mass Effect. [PC Computer, Playstation 3, Xbox 360], 2007.

[6] BioWare. Dragon Age. [PC Computer, Playstation 3, Xbox 360], 2009.

[7] Blizzard Entertainment. Warcraft. [PC Computer], 1994.

[8] Blizzard Entertainment. Starcraft. [PC Computer], 1998.

[9] Blizzard Entertainment. World of Warcraft. [PC Computer, Online Game], 2004.

[10] N. Burgener. Skyrim kinda sucks, actually. `http://thenocturnalrambler.blogspot.com/2012/02/skyrim-actually-kinda-sucks.html`, 2012. Accessed: May 18th, 2016.

[11] Electronic Arts. Brütal Legend. [Playstation 3, Xbox 360], 2009.

[12] M. Floyd and B. Esfandiari. Building learning by observation agents using jloaf. In *Workshop on Case-Based Reasoning for Computer Games: 19th international conference on Case-Based Reasoning*, pages 37–41, 2011.

[13] M. W. Floyd and B. Esfandiari. A case-based reasoning framework for developing agents using learning by observation. In *23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 531–538. IEEE, 2011.

[14] M. W. Floyd and S. Ontañón. A comparison of case acquisition strategies for learning from observations of state-based experts. *FLAIRS*, 2013.

[15] FromSoftware. Dark Souls. [PC Computer, Playstation 3, Xbox 360], 2011.

[16] R. Hunicke. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 429–433. ACM, 2005.

[17] Infinity Ward. Call of Duty: Modern Warfare. [PC Computer, Playstation 3, Xbox 360], 2007.

[18] C. Kohler. Review: Brutal Legend rocks the story, whiffs the gameplay. `http://www.wired.com/2009/10/brutal-legend-review/`, 2009. Accessed: Febuary 20th, 2016.

[19] Level Up Labs. Defender's Quest: Valley of the Forgotten. [PC Computer], 2012.

[20] Lionhead Studios. Black and White 2. [PC Computer], 2005.

[21] D. Livingstone. Turing's test and believable AI in games. *Computers in Entertainment (CIE)*, 4(1):6, 2006.

[22] R. Lopes and R. Bidarra. Adaptivity challenges in games and simulations: a survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(2):85–99, 2011.

[23] K. McGee and A. T. Abraham. Real-time team-mate AI in games: A definition, survey, & critique. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, pages 124–131, New York, NY, USA, 2010. ACM.

[24] Moby Games. Critic review for Battlezone. `http://www.mobygames.com/game/windows/battlezone/mobyrank`, 2016. Accessed: Febuary 20th, 2016.

[25] Neeshka. Skyrim is disappointing : why do reviewers ignore its problems? `http://www.giantbomb.com/the-elder-scrolls-v-skyrim/3030-33394/forums/skyrim-is-disappointing-why-do-reviewers-ignore-it-529212/`, 2012. Accessed: May 18th, 2016.

[26] Nexus Mods. Extensible follower framework. `http://www.nexusmods.com/skyrim/mods/12933/?`, 2016. Accessed: May 18th, 2016.

[27] S. Ontanón, K. Bonnette, P. Mahindrakar, M. A. Gómez-Martín, K. Long, J. Radhakrishnan, R. Shah, and A. Ram. Learning from human demonstrations for real-time case-based planning. *IJCAI*, 2009.

[28] L. Panait and S. Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, Nov. 2005.

[29] P. Preece. Desktop Tower Defense. [PC Computer], 2007.

[30] K. Salen and E. Zimmerman. *Rules of play: Game design fundamentals.* MIT press, 2004.

[31] Sega. Tetris. [Arcade Game], 1989.

[32] B. Tastan and G. R. Sukthankar. Learning policies for first person shooter games using inverse reinforcement learning. AIIDE, 2011.

[33] J. Tremblay and C. Verbrugge. Adaptive companions in FPS games. *FDG*, 13:229–236, 2013.

[34] Trendy Entertainment. Dungeon Defenders. [PC Computer, Playstation 3, Xbox 360, iOS, Android], 2010.

[35] Unity Technologies. Unity game engine. `https://unity3d.com/`, 2016.

[36] S. Yildirim and S. B. Stene. A survey on the need and use of AI in game agents. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 124–131, San Diego, CA, USA, 2008. Society for Computer Simulation International.

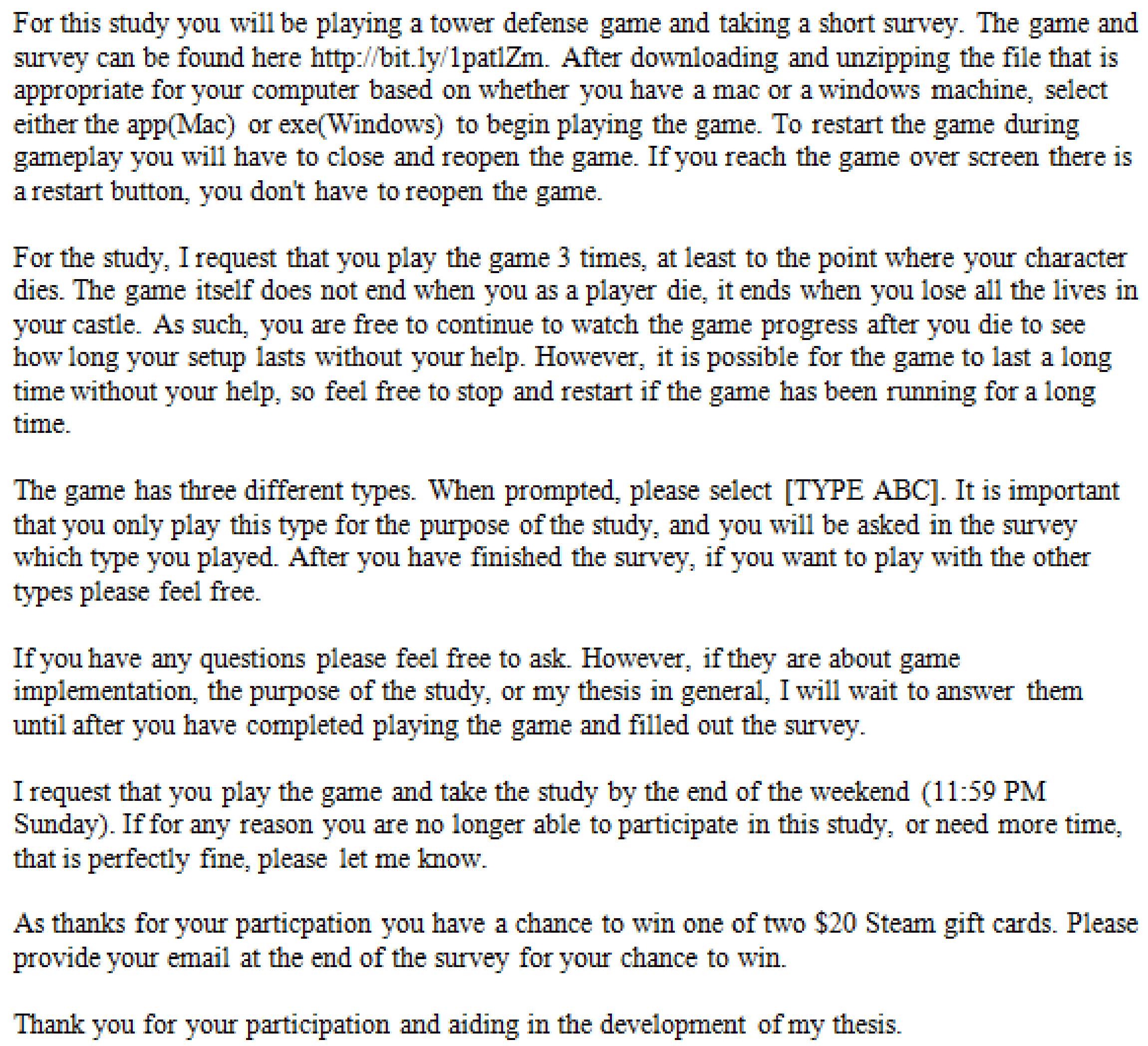
[37] B. Yue and P. de Byl. The state of the art in game AI standardisation. In *Proceedings of the 2006 International Conference on Game Research and Development*, CyberGames '06, pages 41–46, Murdoch University, Australia, Australia, 2006. Murdoch University.

APPENDICES

Appendix A

USER STUDY INSTRUCTIONS

Figure A.1 shows the message that was sent to participants of the user study. The game type was selected before the message was sent.



For this study you will be playing a tower defense game and taking a short survey. The game and survey can be found here http://bit.ly/1patlZm. After downloading and unzipping the file that is appropriate for your computer based on whether you have a mac or a windows machine, select either the app(Mac) or exe(Windows) to begin playing the game. To restart the game during gameplay you will have to close and reopen the game. If you reach the game over screen there is a restart button, you don't have to reopen the game.

For the study, I request that you play the game 3 times, at least to the point where your character dies. The game itself does not end when you as a player die, it ends when you lose all the lives in your castle. As such, you are free to continue to watch the game progress after you die to see how long your setup lasts without your help. However, it is possible for the game to last a long time without your help, so feel free to stop and restart if the game has been running for a long time.

The game has three different types. When prompted, please select [TYPE ABC]. It is important that you only play this type for the purpose of the study, and you will be asked in the survey which type you played. After you have finished the survey, if you want to play with the other types please feel free.

If you have any questions please feel free to ask. However, if they are about game implementation, the purpose of the study, or my thesis in general, I will wait to answer them until after you have completed playing the game and filled out the survey.

I request that you play the game and take the study by the end of the weekend (11:59 PM Sunday). If for any reason you are no longer able to participate in this study, or need more time, that is perfectly fine, please let me know.

As thanks for your particpation you have a chance to win one of two $20 Steam gift cards. Please provide your email at the end of the survey for your chance to win.

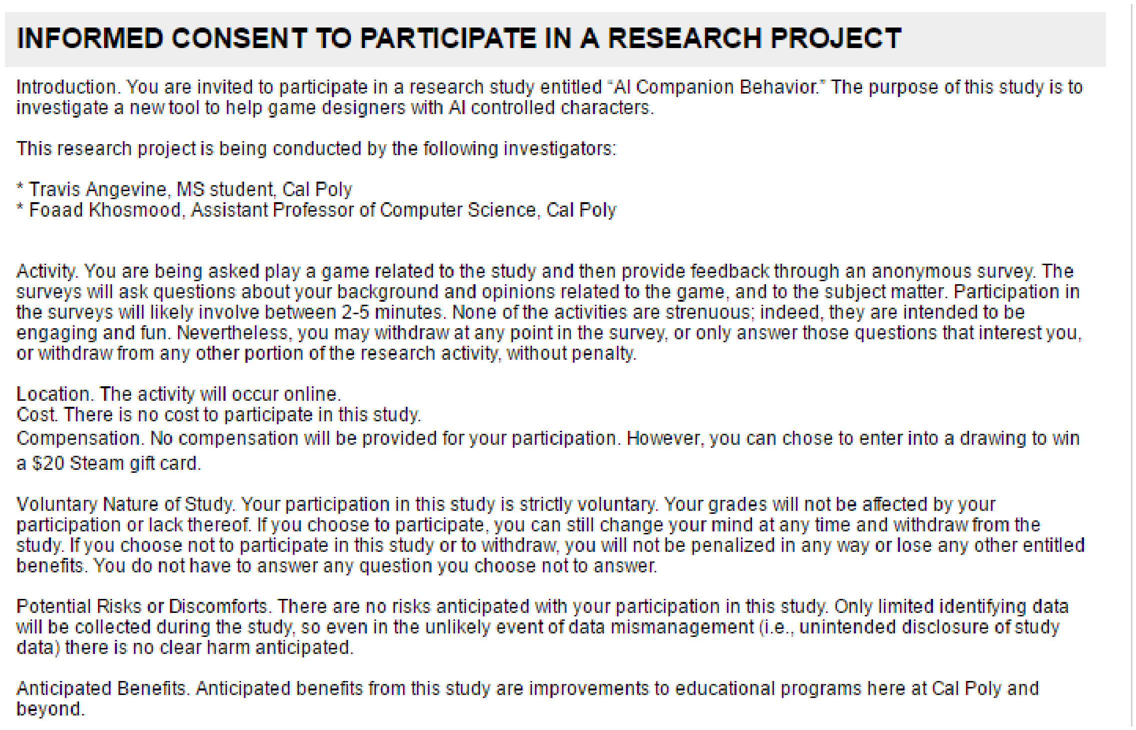
Thank you for your participation and aiding in the development of my thesis.

Figure A.1: The message sent to participants of the user study for instructions

Appendix B

FEEDBACK SURVEY

The following figures show the feedback survey that was given to participants of the user study.



INFORMED CONSENT TO PARTICIPATE IN A RESEARCH PROJECT

Introduction. You are invited to participate in a research study entitled "AI Companion Behavior." The purpose of this study is to investigate a new tool to help game designers with AI controlled characters.

This research project is being conducted by the following investigators:

* Travis Angevine, MS student, Cal Poly
* Foaad Khosmood, Assistant Professor of Computer Science, Cal Poly

Activity. You are being asked play a game related to the study and then provide feedback through an anonymous survey. The surveys will ask questions about your background and opinions related to the game, and to the subject matter. Participation in the surveys will likely involve between 2-5 minutes. None of the activities are strenuous; indeed, they are intended to be engaging and fun. Nevertheless, you may withdraw at any point in the survey, or only answer those questions that interest you, or withdraw from any other portion of the research activity, without penalty.

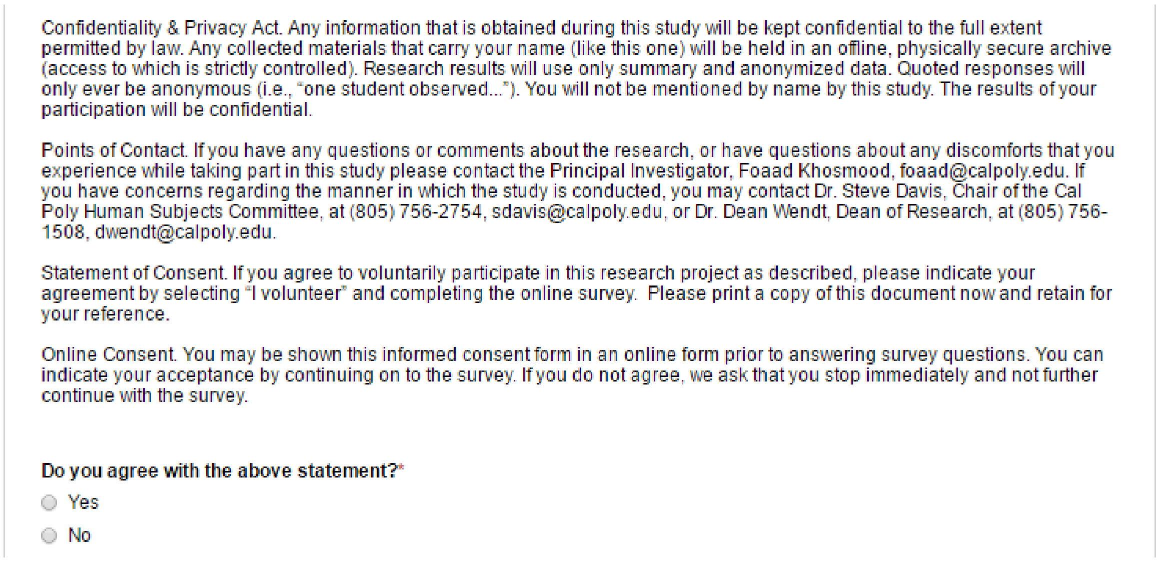
Location. The activity will occur online.
Cost. There is no cost to participate in this study.
Compensation. No compensation will be provided for your participation. However, you can chose to enter into a drawing to win a $20 Steam gift card.

Voluntary Nature of Study. Your participation in this study is strictly voluntary. Your grades will not be affected by your participation or lack thereof. If you choose to participate, you can still change your mind at any time and withdraw from the study. If you choose not to participate in this study or to withdraw, you will not be penalized in any way or lose any other entitled benefits. You do not have to answer any question you choose not to answer.

Potential Risks or Discomforts. There are no risks anticipated with your participation in this study. Only limited identifying data will be collected during the study, so even in the unlikely event of data mismanagement (i.e., unintended disclosure of study data) there is no clear harm anticipated.

Anticipated Benefits. Anticipated benefits from this study are improvements to educational programs here at Cal Poly and beyond.

Figure B.1: Part one of the first question of the feedback survey, providing information to the participants



Confidentiality & Privacy Act. Any information that is obtained during this study will be kept confidential to the full extent permitted by law. Any collected materials that carry your name (like this one) will be held in an offline, physically secure archive (access to which is strictly controlled). Research results will use only summary and anonymized data. Quoted responses will only ever be anonymous (i.e., "one student observed..."). You will not be mentioned by name by this study. The results of your participation will be confidential.

Points of Contact. If you have any questions or comments about the research, or have questions about any discomforts that you experience while taking part in this study please contact the Principal Investigator, Foaad Khosmood, foaad@calpoly.edu. If you have concerns regarding the manner in which the study is conducted, you may contact Dr. Steve Davis, Chair of the Cal Poly Human Subjects Committee, at (805) 756-2754, sdavis@calpoly.edu, or Dr. Dean Wendt, Dean of Research, at (805) 756-1508, dwendt@calpoly.edu.

Statement of Consent. If you agree to voluntarily participate in this research project as described, please indicate your agreement by selecting "I volunteer" and completing the online survey. Please print a copy of this document now and retain for your reference.

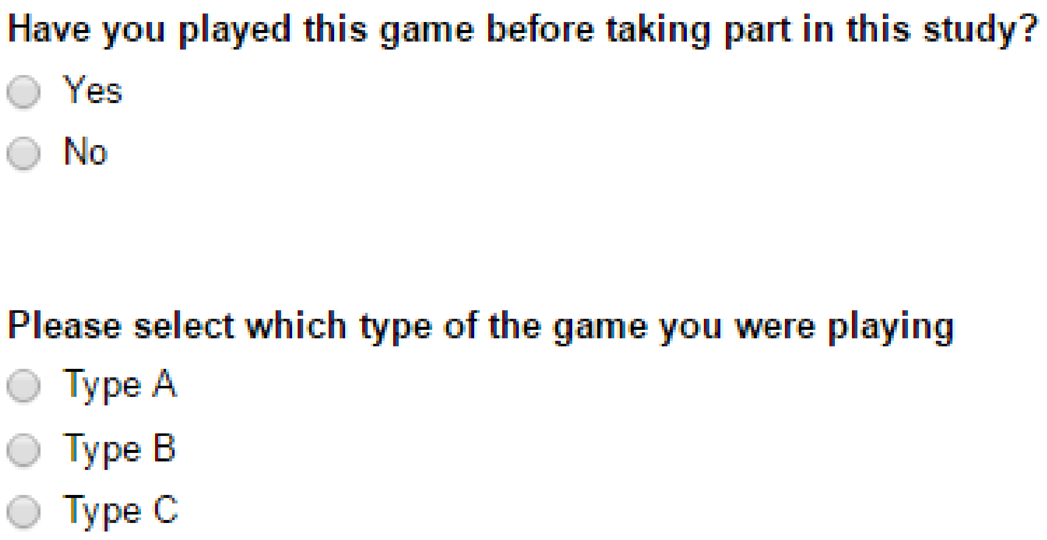
Online Consent. You may be shown this informed consent form in an online form prior to answering survey questions. You can indicate your acceptance by continuing on to the survey. If you do not agree, we ask that you stop immediately and not further continue with the survey.

Do you agree with the above statement?*

○ Yes

○ No

Figure B.2: Part two of the first question of the feedback survey, providing information to the participants



Have you played this game before taking part in this study?

○ Yes

○ No

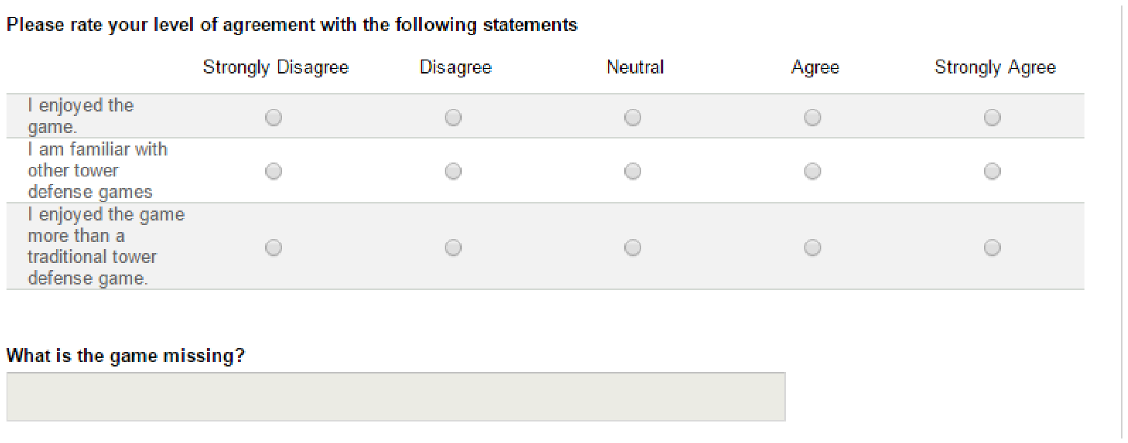
Please select which type of the game you were playing

○ Type A

○ Type B

○ Type C

Figure B.3: Page two of the feedback survey

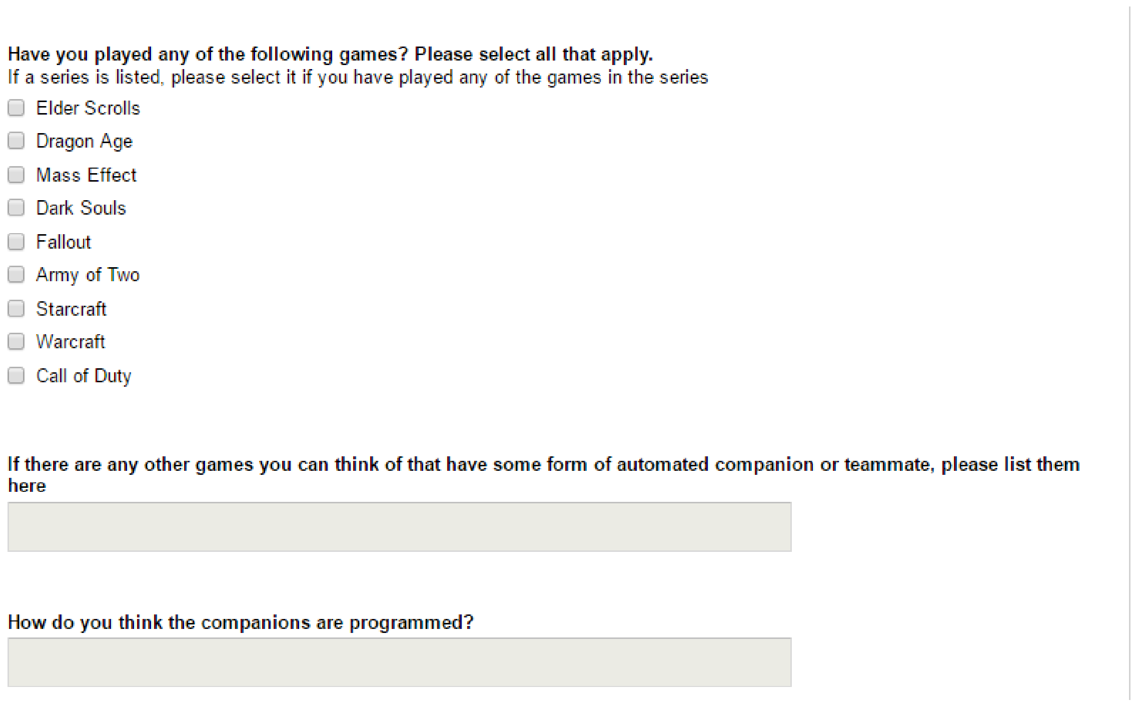


Please rate your level of agreement with the following statements

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I enjoyed the game.	○	○	○	○	○
I am familiar with other tower defense games	○	○	○	○	○
I enjoyed the game more than a traditional tower defense game.	○	○	○	○	○

What is the game missing?

Figure B.4: Page three of the feedback survey



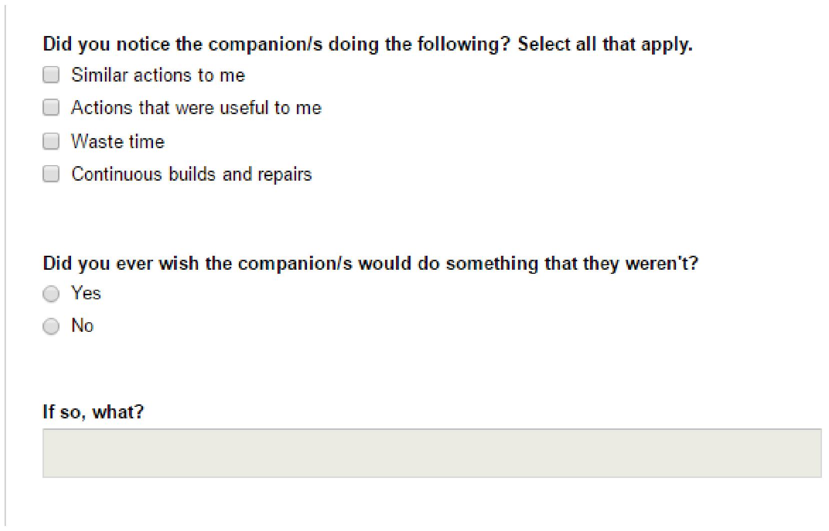
Have you played any of the following games? Please select all that apply.
If a series is listed, please select it if you have played any of the games in the series

- ☐ Elder Scrolls
- ☐ Dragon Age
- ☐ Mass Effect
- ☐ Dark Souls
- ☐ Fallout
- ☐ Army of Two
- ☐ Starcraft
- ☐ Warcraft
- ☐ Call of Duty

If there are any other games you can think of that have some form of automated companion or teammate, please list them here

How do you think the companions are programmed?

Figure B.5: Page four of the feedback survey



Did you notice the companion/s doing the following? Select all that apply.

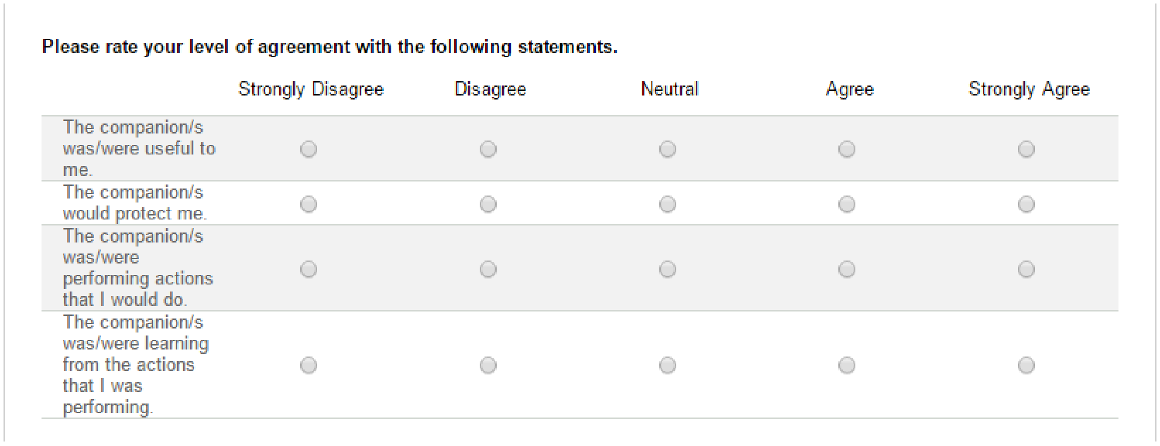
- ☐ Similar actions to me
- ☐ Actions that were useful to me
- ☐ Waste time
- ☐ Continuous builds and repairs

Did you ever wish the companion/s would do something that they weren't?

- ○ Yes
- ○ No

If so, what?

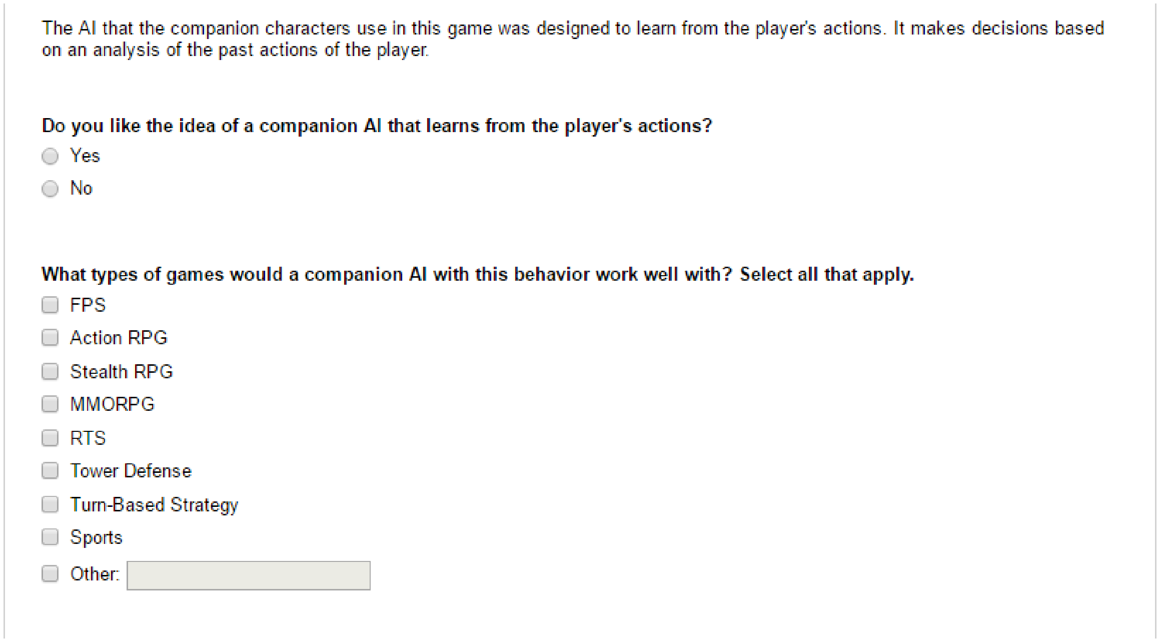
Figure B.6: Page five of the feedback survey



Please rate your level of agreement with the following statements.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The companion/s was/were useful to me.	○	○	○	○	○
The companion/s would protect me.	○	○	○	○	○
The companion/s was/were performing actions that I would do.	○	○	○	○	○
The companion/s was/were learning from the actions that I was performing.	○	○	○	○	○

Figure B.7: Page six of the feedback survey



The AI that the companion characters use in this game was designed to learn from the player's actions. It makes decisions based on an analysis of the past actions of the player.

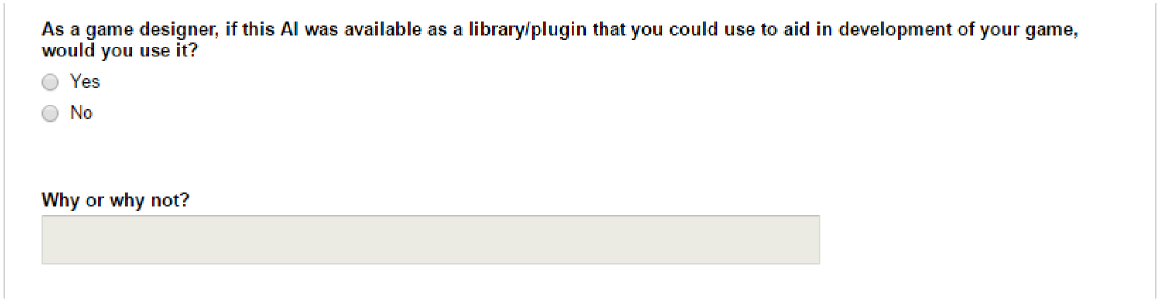
Do you like the idea of a companion AI that learns from the player's actions?

- ○ Yes
- ○ No

What types of games would a companion AI with this behavior work well with? Select all that apply.

- ☐ FPS
- ☐ Action RPG
- ☐ Stealth RPG
- ☐ MMORPG
- ☐ RTS
- ☐ Tower Defense
- ☐ Turn-Based Strategy
- ☐ Sports
- ☐ Other:

Figure B.8: Part one of page seven of the feedback survey

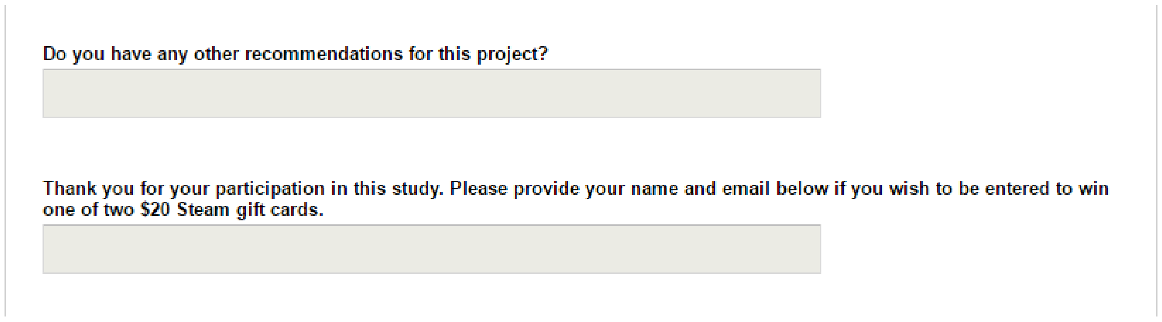


As a game designer, if this AI was available as a library/plugin that you could use to aid in development of your game, would you use it?

- ○ Yes
- ○ No

Why or why not?

Figure B.9: Part two of page seven of the feedback survey



Do you have any other recommendations for this project?

Thank you for your participation in this study. Please provide your name and email below if you wish to be entered to win one of two $20 Steam gift cards.

Figure B.10: Page eight of the feedback survey