IMPROVING COMPANION AI BEHAVIOR IN MIMICA

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

Daniel Toy

October 2017

COMMITTEE MEMBERSHIP

TITLE:	Improving Companion AI Behavior in MimicA
AUTHOR:	Daniel Toy
DATE SUBMITTED:	October 2017
COMMITTEE CHAIR:	Foaad Khosmood, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Alexander Dekhtyar, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Improving Companion AI Behavior in MimicA

Daniel Toy

Companion characters are an important aspect of video games and appear in many different genres. Their role is typically to support the player as they progress through the game by helping to complete tasks or assisting in combat. However, oftentimes, these companion characters are limited in their ability to dynamically react to new situations and fail to properly assist the player.

In this paper, we present a solution by improving upon the MimicA framework, which allows companion characters to emulate the human player. The framework takes a learn by observation approach by storing the game state when the player performs an action. This is then used by machine learning classifiers to determine what action to take and where it should be done. Because the framework makes little assumptions about the rules of the game and focuses on a single session experience, it is flexible enough to apply to a variety of different games and requires no prior training data. We have further improved the original MimicA framework by adding feature selection, n-gram analysis, an improved feedback system, random forest classifier, and a new system for picking a location for actions. In addition, we refactored and updated the original framework to make it easier to use for game developers and the game, Lord of Towers, which was used as a proof of concept. Further, we create another game, Lord of Caves, to demonstrate the flexibility of the new version of the framework. We validated our work using automated simulations and a user study. In our automated simulations, we found random forest was a consistently strong performer. Our user study found that the our implementation of n-grams was successful and 19 of 26 believed our framework would be useful to a game developer.

ACKNOWLEDGMENTS

Special thanks to Foaad Khosmood for his support and guidance in the development of this thesis. I would also like to thank Franz Kurfess and Alex Dekhtyar for reviewing the work. Further thanks to Travis Angevine for his work on the original MimicA framework and for providing advice. Additionally, I would like to thank Gavin Scott for his assistance as we worked in parallel on separate projects regarding MimicA. Lastly, thank you to everyone who participated in the survey and provided feedback as well as those listed in the art credits for supplying art and preventing my poor drawing skills from coming to light.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Artificial intelligence is an integral component in video games and has grown to encompass many different aspects of video games as they have continued to grow more complex [11]. While early games such as Pong [25] may not have made use of AI in many areas, today's modern games use AI for tasks from controlling characters to generating content. This is evident in games such as Skyrim [26] and Gears of War [29] where players can immerse themselves in massive worlds populated by legions of characters that are hostile, neutral, or allied to the player. AI is essential in these games for creating character behaviors including path planning and determining what actions to take in response to the game state. However, many of these strategies for determining what action to take are only able to create the illusion of intelligence and can fail in different situations [6, 8].

1.1 Companion characters

Companion characters, in video games, are the characters in a game which provide support to the player as they accomplish different tasks. They fall under the category of non-playable characters (NPCs) which also includes enemies who typically fight the player and neutral characters such as store keepers or quest givers that the player interacts with. The goal of a companion character is to be the player's ally throughout the game and provide assistance [18]. Ideally, they are able to emulate a human who follows the player and is their trusted sidekick. To do this without pre-planned or hard-coded behavior, they need to be able to observe the player and determine the players goals. Using this knowledge, the companion can then attempt to take the

best action. These actions can come in several forms ranging from passive actions such as healing to more aggressive actions such as participating in combat [37].

These companions also serve roles in the game, other than providing assistance. Firstly, they can serve as a form of dynamic difficulty adjustment in which they give more assistance to the player if they are struggling or become more passive if the player is doing well. This can help the game maintain balance and keep players progressing smoothly [39]. Companion characters can also provide a social aspect to games, particularly in single player missions. In these cases, they can make players feel as if they are part of a squad or larger team. In short, they can become something that the player trusts. Lastly, companions can be used to direct the player's focus and help tell the story of the game. They can warn the player of impending danger or point out interesting clues hidden in the game's flora and fauna. Thus, companions can serve several other roles in addition to helping the player progress through the game [40].

Another important trait of a companion and, on a broader scale, all NPCs is being believable. Believability is vital to keeping players immersed in the game and is determined by the player's perception [17]. Characters need to behave in a manner that the player expects and exhibit human traits such as social and strategic skills. They also need to display human levels of ability [40]. For example, an NPC which has superhuman reaction speeds or aiming skills is not believable and can often make the game less enjoyable [15, 40].

If not implemented correctly, companions can have varying or negative effects on gameplay. Oftentimes, their decision making can actually detract from the gaming experience as they often fail to properly cooperate with the player as a normal human would. Instead, they get in the player's way and become more of a hindrance. This can ruin the game experience by breaking the suspension of disbelief and frustrating

the player [39]. This is due in part to game companies placing more focus, in terms of time and resources, on graphics, storytelling, and level design. This takes away from the available resources to work on companion AI [21]. In addition, the use of static scripts or rule-based systems make the companions unable to react to new events that were never accounted for or make their actions predictable and less human-like [11, 20, 36].

As a solution to this, games can use artificial intelligence to learn and gather information [41] to dynamically decide what action to perform. Dynamic decision making has many benefits [16, 23]. However, it brings up several issues that make it an unattractive option for game companies, particularly in terms of testability and the unpredictability that is incurred from using it. Unfortunately, most companies in modern games are unwilling to take this risk and stick to reliable and better-tested, hard-coded behaviors [36].

An example of allies that failed to be good companions are the ally soldiers from Gears of War [29], a first-person shooter game where the player fights a hostile subterranean species. Many of the complaints about these characters focus on how poorly they play the game including trying to melee a boss (instead of using their gun) or standing on a fire pit. Many players also complain about Dom, the main companion that plays alongside the player for a majority of the game, who continuously gets himself killed by running head first into the fight [3, 24]. In these situations, players will become more annoyed, especially in the case of Gears of War [29] where players must revert to the last checkpoint if Dom dies.

On the other hand, good companion AI make a game more enjoyable for players by creating companions who respond accordingly to the player's intent. In combat situations, they attack the right targets and in other situations they respond as the player would expect them to [38]. One example of such a character is Ellie from Last

of Us [30] who is an adolescent girl that the player escorts across a zombie infested United States. The developers at Naughty Dog focused on trying to build a connection between the player and Ellie, so she finds items and stays close to the player, especially during stealth portions of the game. Most importantly, they ensure that she is less of a burden on the player by having her be invisible to enemies [9]. While this reduces the realism of the game, it prevents her from becoming a hindrance throughout the game and shifts the player's focus to killing enemies instead of constantly having to worry about her dying.

1.2 Approach

The approach taken in this thesis is to create an adaptive autonomous AI that will mimic the player's actions. This is done by building upon MimicA [4], a framework for developers to create companion AIs who attempt to copy the player's actions. To improve the design of these AIs, we added several components such as a random forest classifier, feature selection, and n-gram analysis that try to improve AI usefulness and ability to make better decisions. In addition, we improve the feedback system so

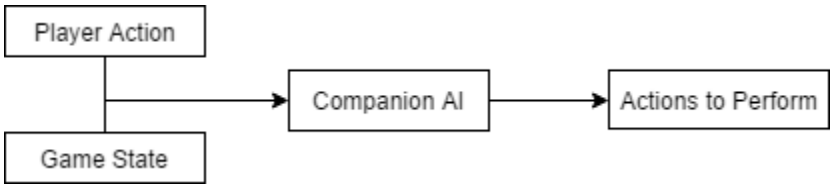


Figure 1.1: Basic Flow of Framework

players can give some guidance to the companion as well as a dynamic region system to determine location of actions. The framework has also been refactored to make it easier to use by making a clearer API and classes that need to be integrated. This ideally provides developers with the means to create games at a faster pace by allowing them to focus on designing gameplay instead of creating behavior for a companion.

Lastly, we present two sample applications: Lord of Towers and Lord of Caves. Lord of Towers was the sample game presented with the original MimicA and has also been updated with new mechanics and visuals. Lord of Caves is a brand new game in an entirely different genre.

It is important to note here that this framework focuses on using single session methods primarily because we do not have access to thousands of samples of gameplay to analyze data. In reality and especially for new games, getting a hold of this amount data is very difficult. While being able to analyze this does allow for the creation of more precise models, the system becomes useless if it needs but does not have access to this much information. In addition, our goal in this framework is to create a companion tailored to the player currently playing and so other player data becomes much less important and maybe even useless. For these reasons, we need to be able to perform our computations and training during run-time. This is why the methods we select need to be quick and efficient so we stray from more computationally expensive but also more accurate approaches.

Chapter 2

BACKGROUND

This section discusses background on the areas used in our approach. In particular, we discuss learning by observation, MimicA (the original framework), and some of the concepts behind our improvements including feature selection and n-grams.

2.1 Mimicking Behavior

We define mimicking behavior as performing actions that the player would typically perform. In particular, the companions aim to choose actions that the player would do, given the current game state. This type of behavior is useful because it tailors the companion's actions to those of the player and allows the companion to choose actions that are in line with the player's. Ideally, it also ensure that the companion is able to mimic the player's strategy and thereby increase its usefulness as opposed to those companions whose behavior is static and more reactive. The addition of dynamic learning allows these companions to further tailor this behavior to individual players to make the experience vary by player.

2.2 Learn by Observation

Learn by observation is an approach to creating behavior in AI agents that are able to learn from humans. The idea involves having AIs typically observing a human expert performing some task. A common approach is to save these observations at some point or interval. Then, as the need arises, perform a search through the observations using a decision making process to find an action most suitable. To do this, the current state of the game world is taken as input to find the closest matching state

in the observations. Then a corresponding action is chosen [19, 20].

This approach is powerful because it can be abstracted such that the learning is generalized. This allows the agent to learn a wider variety of tasks. Learning by observation can also be done dynamically and performed at run-time which is ideal for video games. In addition to this, planning can be used to choose a better action. However, one weakness of this approach is that it may be difficult to combine actions into a sequence depending on how observations are stored [19].

2.3 MimicA

MimicA is a framework that makes use of learn by observation to allow companion characters to emulate players by making similar decisions. To do this, it saves an event with the game state at that moment into a data dictionary whenever the player makes an action. To decide on which actions to take, the framework implements three different classifiers: K-Nearest-Neighbor, a decision tree, or Naive Bayes [4, 5].

2.3.1 Classifiers

In order to make decisions, MimicA takes the current game state and uses one of three classifiers to find an action that is most appropriate.

K-Nearest Neighbor (KNN) is a machine learning algorithm that can be used for classification and regression. As seen in Algorithm 1, it picks actions from the dictionary by comparing the current game state to every other game state currently saved in the dictionary. It determines the game states that most closely match the current one, meaning that these game states have the most values in common. After finding the top five matches, it outputs the corresponding events paired with these game states in the dictionary and returns them in descending order of relevance. Thus

it provides five different options [4, 5].

foreach *Game State Action Pair in Data Dictionary* **do**
    Calculate difference between current game state and game state;
    if *difference is among 5 lowest differences* **then**
        add corresponding event to list and remove highest difference;
    end
end

Algorithm 1: MimicA's KNN Algorithm

Decision trees are more complicated than KNN and require more overhead before classification can begin. Specifically, decision trees require prior training in which it builds a tree data structure such that it contains nodes connected by branches and eventually end in leaf nodes which have no more branches. In MimicA, decision tree nodes hold a feature from the game state which it uses to divide the dictionary. To pick this feature, MimicA utilizes entropy and information gain. Entropy is the uncertainty of a piece of information and can be used to measure the purity of a particular feature. We calculate entropy using the equation in 2.1, which sums the negative probability of each action, p_i, multiplied by the log base two of each action's probability [4, 5].

$$entropy = \sum -p_i * log_2 p_i \tag{2.1}$$

Information gain is derived from entropy and can be used to determine which features are more important. In decision trees, this is calculated by taking the entropy of the parent subtracted from the average entropy of its children. Thus, in MimicA we can use the feature in the game state that has the highest information gain to split the dictionary at each stage. This process continues until either the node is of a certain depth or the calculated entropy is below a threshold. At this point, an event

is placed into the node which, in MimicA, is determined by finding the most common event in the dictionary. After training, the decision tree can then take a game state, traverse the tree, and output an event very quickly. However, it only returns one event [4, 5].

Naive Bayes is a classifier that uses probabilities to determine an action for the companion. This probability is calculated by multiplying the probability of the event by the probability of each feature in the game state that matches that event. The events with the highest probability are then chosen by the classifier. Similar to decision trees, this process also requires prior training in which it determines the count of every event currently stored in the dictionary. These are then used to calculate the action probabilities when passed a game state to classify. To calculate the probability of a feature, a count of the number of times the current game state feature's value matches a game state feature's value in the dictionary with the same event is calculated and then divided by the total number of events. Note that multiplying these probabilities would result in a very small number so the product rule of natural logarithms is used to sum the probabilities' natural logs. The final formula for this can be seen in Equation 2.2. As output, MimicA will return the five events with the highest probability [4, 5].

$$\begin{aligned} ln(p(action|vector)) = ln(p(action)) \ + \ ln(p_1(feature_1|action)) \ + \ ... \\ + ln(p_n(feature_n|action)) \end{aligned} \tag{2.2}$$

Lastly, it should be noted that while decision trees and Naive Bayes only use the data they are passed when trained, they can be retrained at any point. This, however, can take some time and can cause notable lags in the game's framerate, especially when a lot of data must be processed.

2.4 Feature Selection

Feature selection is a process commonly used in machine learning to choose a subset of the input data and discard the rest. This subset of data only contains those that contribute most to classification. There are many benefits to using feature selection but most important to this framework are the increased speed for learning algorithms and improved data quality which should improve the performance of the algorithm [14].

There are several different types of feature selection including filter methods, wrapper methods, and embedded schemes. Filter based feature selection focuses on reducing the size of the data by determining which contribute less and removing them. This is done by analyzing a single or combination of features contribution to classification and then determining which contributes the least. These features are then culled and the process is repeated until the data set is of a certain size. In our approach we make use of an univariate filter based approach because it is faster and independent of the classifier being used. However, the disadvantages of this is that we do not take the interactions between features into account since we only analyze them in isolation. While wrapper methods and embedded schemes may create better subsets of data, we require speed in this case because we wish to perform this feature selection in real time when the companion is instantiated in the game. In addition, these other methods are typically classifier dependant [14]. We provide a more in-depth discussion of our use of feature selection in section 4.3.

2.5 N-grams

N-grams are a method to analyze a series and try to predict some future action using previous series. It is typically used in computational linguistics and probability. A

sequence of length 1 is called a unigram, one of length 2 is a bigram, and 3 is called a trigram. Beyond this, they are simply referred to as 4-gram, 5-gram, etc. To predict some future term, n-grams analyze previous sequences. This is done by taking the last n-1 sequences and then finding the every corresponding n-1 sequences in the history and picking the term that appears most often. In linguistics, this can be used to try to predict the next word in a sentence [13]. In this paper, we use n-grams to try to find the next action that the companion should make using the entire history of actions the player has made as data. We discuss the specifics of their use in section 4.4.

2.6 Tools

The MimicA framework and our improvements are built in Unity [34] using the C# language. Unity is a popular engine for game developers because it provides many features for free. These features allow for easier and faster setup for games because it does much of the tedious work that would otherwise take a large amount of time such as piecing together animations, creating a physics engine, or catching collisions. Thus, by using Unity, we are able to focus on improving the MimicA framework rather than worry about editing or building a new game engine. However, our use of Unity also imposes some restrictions as it requires most of the work to be done on a main thread and can also be difficult to debug at times.

Chapter 3

RELATED WORK

In this section we discuss work that has also taken a learn through observation approach for creating more dynamic companions including the original MimicA framework. In particular, we describe their approach and also consider the differences in our strategies. We then look at some examples of companions that exist in today's game industry as well to see how mimicking companions are used and how they apply to different game genres.

3.1 MimicA

As this thesis is based on previous work, we discuss the MimicA framework which served as our starting point. The framework focuses on reducing the workload on game developers for creating companion AI. It uses learn by observation and saves the game state whenever the player performs an action. The game state is saved in a dictionary which is then used by a classifier to determine what action a companion should take [4, 5].

To use MimicA, developers must clearly define several parts of the game in order to start using it. Firstly, developers must define the game state class that represents the state of the world at any point in time. This includes all the important variables that the developers feel the companion should know to make decisions. To function most effectively, MimicA needs to save the game state every time the user performs an action which it then saves as a pair in the data dictionary. These actions must also be defined by the developer [4, 5].

As a proof of concept, the framework was implemented in a game called Lord of

Towers which is a unique tower defense game where the player controls an avatar in the world and eventually dies after a set amount of time. However, all is not lost, as the player is joined by companion characters who assist the player and continue playing even after the player's time is up. Thus, this game focuses on being able to train your companions properly in your short time [4, 5].

3.2 Learning from Demonstration

One approach to learn through observation is given by Mehta et al. in which the AI learns from human demonstration. This process consists of four steps: demonstration, annotation, behavior learning, and behavior execution. Demonstration involves having a human play the game and saving all the actions they take in a log file. Annotation is a manual step requiring a human to specify the goals of each action in the log. This could be automated but keeping it manual allows the user to specify which behaviors are learned. Behavior learning then takes this and extracts procedural behavior which is stored in a behavior base. These are translated into behaviors and are stored for later use. Lastly, behavior execution step involves the combination of behaviors from the behavior base which can then be formed into plans [19].

The main difference between our approaches is that this method requires a manual annotation step. This is unfeasible for our framework because we seek to perform this behavior learning in real time and cannot have the player stop the game to describe their different goals before having a companion join them. In addition, our framework seeks to minimize the overhead required for integration into a game engine and as such we would like to avoid this. On the other hand, Mehta et al. point out that by adding the annotation step, developers can exert greater control over the AI which is still possible in our framework but requires the developer hard-code certain behavior.

3.3 Rapidly Adaptive AI

Another approach proposed by Bakkes et al. is a rapidly adaptive game AI where domain knowledge is gathered "automatically by the game AI, and is immediately utilised to evoke effective behaviour" [6]. In this approach, each character in the game sends the game AI its current game state, receives some action to perform, and then, after completing that action, reports back on the results. The character's observations in combination with observations made by other characters, such as teammates, are used to gather cases from a case base. The game AI then uses an evaluation function that incorporates temporal difference learning to dynamically decide on weights to pick the appropriate action. Lastly, an adaption mechanism such as reinforcement learning, is incorporated to allow the AI to better adapt to different situations [6].

The major difference between this approach and our framework is the offline work that Bakkes et al. perform prior to the start of the game. This includes indexing previous games by time step and with a fitness value according to the desirability of that state as well as grouping observations together using k-means clustering. This information is not always available to our framework as we may not have access to previous game data. Often, our framework has to work with minimal data gathered prior to the companion's entrance. Thus we cannot rely on being able to perform this offline work which in this case allows for reduction of the search space and optimization to avoid work later.

3.4 jLOAF

jLOAF is a framework that allows agents to use learn by observation in real time. It observes the expert over time and saves the actions as cases. These cases serve as a log for a specific action and includes time, the action taken, and the entire run

up until that point. This allows the system to represent the entire reasoning process that led to the action. Inputs can be either atomic or collections of atomic inputs to allow modeling of both simple and complex actions. It should be noted that the framework does not keep track of the success of an action which renders techniques such as reinforcement learning ineffective [10].

jLOAF can also make use of optional preprocessing to reduce the burden of computation during run-time. Feature selection finds useful features and optimizes retrieval accordingly. Another preprocessing step, redundancy removal, attempts to reduce the size of the case base by removing or replacing similar cases with a single one. Case base analysis finds areas in the problem space that are underrepresented so as to stop collecting this data. Case base restructuring converts the data into a different format in order to reduce retrieval time [10].

jLOAF performs case-based reasoning in four stages: retrieval, reuse, revision, and retention. Retrieval compares the current cases to stored cases in the case base. Similarity between cases is done by combining similar individual inputs though the framework also allows a custom comparison to be implemented for more flexibility. Reuse directly copies the solution to the case and does not perform any adaptation. Revision and retention both require much more specific domain knowledge and must be implemented by the developer [10].

This general framework is similar to ours in that both seek to create real time AIs which learn through observation. Both choose actions based on similar states, or cases, and then directly apply that action. In our framework, however, the execution of this action is dependent on the developer to define. We also use feature selection to improve our data set by shrinking it and culling less valuable features though we do this step in real time instead of as a preprocessing step.

3.5 Darmok 2

Darmok 2 is another framework that uses a real-time case-based planning system to play Real Time Strategy (RTS) games. It takes an online approach and includes the ability to analyze and acquire cases by observing human demonstration, interleaving planning and execution, real-time plan adaptation, and adversarial case-based planning. The system gathers observations from human demonstration for a set of goals which are defined beforehand. These observations are then stored in a case-base and include the timestamp, game state, and the actions taken prior. These actions however are more than just the event, they include a large amount of other information including goals, preconditions, success conditions, and other parameters required in planning. These observations are then formed into cases. These cases are plans stored with their outcomes which is a success score between 0 and 1. Then, Darmok 2 chooses plans using a hierarchical tree and an adversarial case-based planner to pick a combination of the proposed plans that best fit the goals. In this step, they also use a simulator to better predict outcomes [22].

The largest distinction between Darmok 2's approach and ours is the planning done. Our framework does not make use of planning and as such does not require the same amount of information about each action. In our framework, we store game states and events, which correspond to Darmok 2's actions. However, our events do not include the preconditions, postconditions, success conditions, etc. because these are not possible for our framework to know. Thus we require less work on the developer's side for integration as each action or event requires less work to create. In addition, the simulator that Darmok 2 uses is not something that we can implement in our framework. Particularly, we have developed in Unity which does not support the ability to create copies of a game in another thread. Running this in the background would also require a high overhead in terms of performance and could cause visible

frame drop within the game, something that would definitely detract from the game's experience.

3.6 Commercial Video Games

Lastly, there are many examples of companion AIs in existing commercial video games. While many of these do not feature adaptive companion AI, they do make use of companion characters and provide offline training and testing to ensure they perform their tasks at least adequately. Companion characters tend to be more prominent in certain genres such as role-playing (RPG) and shooter games. In each genre, we examine some games as well as discuss how a mimicking companion would be helpful.

RPG games commonly have companion characters which accompany the player. For example, in Skyrim [26] players can make use of companions as support characters. In these cases, mimicking activity can be used to complement the player with a similar play style. However, it is important to note here that it may not always be good to copy all of the player's actions. Consider the case in a combat scenario where the player is a melee warrior while the companion is a spell-casting wizard. In this case, the companion should not run in and melee attack as the player would and mimicking actions would not apply to their combat pattern [11].

In adventure or action games, the player is often exploring a world while solving some problem that pertains to the story or collecting items along the way. Oftentimes, the player has a companion who joins them and coordinates its actions with the player [18]. In these games, it is often very useful if the companion performs similar actions to the player, especially during stealth portions of the game. A great example of this is Last of Us [30], in which the companion, Ellie, follows the player and helps them in their tasks.

Within shooter games, companion characters typically exist as squad mates or fellow soldiers that join the player as they battle through the game. In these instances, these characters are expected to support the player with cover fire and other combat tactics in order to better simulate a combat environment. As Tremblay stated, the goal is to "help the player accomplish in-game goals, simulating the effect of actual co-operative gameplay" [39]. An example of this is Gears of War [29] where the player usually has a one or more companions at different points throughout the game. In these cases, it would be optimal if the companions would support the player by performing useful actions extracted from the player's play style rather than following their own agenda. This can be accomplished if they perform similar actions but with minor modifications such as flanking instead of simply following behind the player.

Sports games can also have companion characters particularly as fellow team mates. Depending on the sport, it may not always be helpful to have companions mimic the player's actions. Often in sports games, companions fill in the team and take roles not taken by the player. Similar to shooter games, it is vital that companions are able to cooperate with the player [18]. However, a mimicking companion could still apply in games where the player can control parts of a team and switch roles during the game. The companion AI can then take over the teammates and make decisions that are similar to the player's play style. This can be applicable in sports games such as the NBA 2K [31] or Madden NFL [27]. It can also apply to sports where it would be preferable if the player's partner had the same play style such as doubles in tennis.

Racing games rarely contain companion characters, but Forza Motorsport 5 [33] introduced an interesting AI that learns from player's driving style. This game makes use of a "Drivatar" which collects data on how players drive to influence how AI opponents will drive during a race. [35] While the effectiveness is disputed in forums [2], it is still considered to be an improvement over the previous version [1]. Forza's

Drivatar also gives an example of another application of mimicking behavior other than to companion characters.

Lastly, RTS games can also make use of companion characters though mimicking characters may be less applicable. At the unit level, it would be interesting to see these units perform useful actions without the player explicitly telling them to avoid idling. For example, this would be helpful in a game such as Age of Empires 2[28] where villagers could automatically collect nearby resources as an option for beginners who are just learning how to play. This type of behavior is not typically seen in RTS games however as the player typically assumes more control over them and may not want their units to move on their own. A companion AI could also be useful as another player on the same team. A cooperative teammate who copies the player's actions may be interesting but one that seeks to perform actions that complement the player's strategy would be much more helpful in this case [12].

Chapter 4

SYSTEM DESIGN

In this section we discuss the major changes and improvements made to the original MimicA framework. These changes were aimed at improving the companions' ability to make decisions as well as the speed with which they make them.

4.1 Overview

Our framework uses learn by observation by saving game states and events that occur as the player progresses through the game. Whenever an action is performed by the player, that action is saved with the corresponding game state when that action was performed as a pair. The game state consists of game variables that encompass all important aspects of the game. These can be anything from player health, distance to closest enemy, or type of the enemy nearest to the player. It is also important to note that the location of the action in terms of world coordinates is also saved.

When companions need an action to perform they use a classifier to get a list actions. To do this, they take the current game state and feed it to the classifier. The resulting list of actions will be sorted in the order of descending fitness to the situation, according to that classifier. Each action will also contain a location in the form of a range in Unity's world coordinates. We will discuss how this is done in the section 4.2. Figure 4.1 shows the general flow when a companion needs to make a decision. The sequence analyzer is our use of n-grams and is used prior to classification to see if we can shortcut the classification if there is an obvious choice. This will be discussed more heavily in section 4.4. To note, the system currently uses a KNN classifier to determine location, but this can easily be switched out for another

classifier.

Lastly, the game can tell the companion that certain actions should not be performed. Actions can be banned permanently or temporarily which allows for the autonomous companions to receive some feedback from the user.

When companions are instantiated, they are given the entire data history of actions game state pairs currently stored. However, this list is not updated constantly as some classifiers require training before they can function. To solve this, companions can be retrained if asked to do so by the game (e.g. calling the retrain function which is described in the chapter 5) or when all their actions are found to be banned.

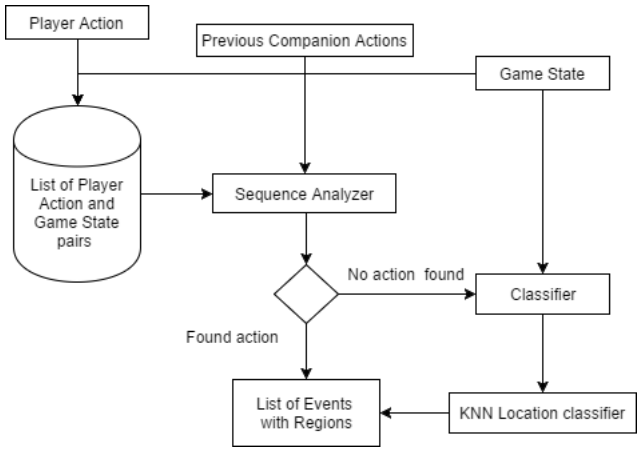


Figure 4.1: Diagram of Decision making

4.2 Regions

The issue of determining a location for action is not a new problem and was something that was previously addressed in MimicA with the introduction of sectors. In MimicA, these sectors evenly divided the world into six parts and gave the framework a way in which to refer to relative locations rather than the world coordinate locations. Using a relative location makes sense because having a companion attempt to perform the same action in the same location is not typically desired behavior. For example, this

would not make sense in a shooter game if the player is moving up but the companion is still covering fire from the player's original position. Thus sectors provided a way to give the game a general location to perform an action. They also move the problem of determining the exact location of an action over to the game's side. However, there are several problems that these sectors fail to solve. The first drawback is that it forces developers to define these sectors. This means the developers must divide up their world into different parts and try to predict where things should more often occur. Secondly, they scale poorly in cases where the a majority of the events in a map are concentrated in a certain area. In these cases, developers would have to manually divide these sectors into smaller components which is tedious and often requires play-testing to expose.

Our solution to this is to introduce regions. These regions are essentially dynamic sectors which are created as the game progresses and are constantly updated. The entire map begins as one region and is then split in half along its larger axis when an action occurs within it. Regions are constantly split up to a certain threshold. The regions are saved as two points in world coordinates. This means that if the developer defines some other space, such as tiles, they must be able to convert between the world coordinate space and their own locations. This approach requires the developer to do much less work. Particularly, they do not have to define sectors and instead only define the bounds of the world. In addition, these dynamic regions allow for actions to be concentrated in locations much better. For example, if most of the events occur in one area, then it will be subdivided to allow for more precise location picking. A diagram of regions can be seen in Figure 4.2 where the dots represent actions in the world which caused a region to split. In this map, four actions have been performed in the world causing a region to split in each case.

The requirements of adding this type of location determination is that the world coordinates of each action must be saved. This means that in our framework, we

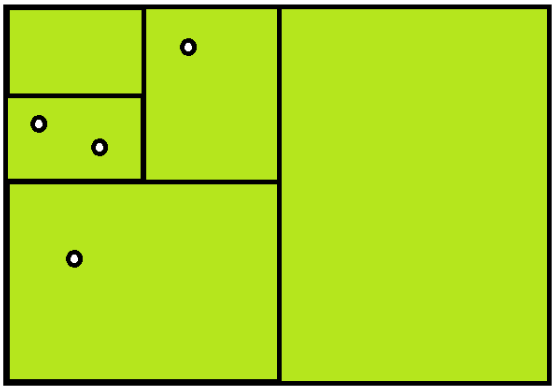


Figure 4.2: Region Splitting with Four actions

introduced a wrapper class that contains a vector for location as well as the event taken. In addition, we add a new type of event, a location event, which is described in chapter 5.

In order to pick a location, we use a KNN classifier. This classifier uses a subset of features from the game state vector which must be passed in by the developer when initializing the companion. The framework uses this additional classifier to pick a world coordinate for every action that the framework is returning which is then converted into a region depending where it is located. Thus it adds regions to every action that is a location event. This will allow for more finely tuned action location determination especially when actions are concentrated in a single area which is an issue mentioned with sectors.

4.3 Feature Selection

Though adding more features to the game state vector will give classifiers more data to work with and ideally become more accurate, it will also slow the companion's decision making process. In order to alleviate this, we use feature selection to shrink and improve the search space. Specifically, we take a filter-based feature selection approach that uses information gain to determine the features that are more valuable.

We also considered principal component analysis which is good for dimensionality reduction; however, we found that this method was too slow to be used in real-time.

We perform our feature selection during training before the classifier is initialized by passing it all the current game state vectors currently held in our data history. After calculating the information gain of each feature, we remove some bottom percentage of lowest scoring features (typically 10%) and output a list of those that should be used in classification. In particular, we return a list of string names of each feature which the classifiers then use to skip over features not on this list.

While adding feature selection saves time by reducing the amount of variables to consider when performing classification, it also has an overhead. For this reason, the percentage of features to be removed can be specified when initializing a companion and can be toggled off by setting this percentage to zero. Lastly, retraining will completely regenerate the list of features to ignore. This allows features whose information gain has changed since the last training to be either removed or added.

4.4 N-grams

Another problem noted in the original MimicA framework included difficulties with performing consecutive actions that would logically follow each other. For example, if building a tower consists of two separate actions: build (to create a new tower) and repair (to bring it to full health) then, the framework may not always return the repair action after the build tower action.

As a solution, we use n-gram analysis to improve the action determination, particularly for recognizing sequences of events. In the framework, we define a class called Sequence Analyzer which takes a parameter, n, to define the length of the sequences to analyze. The class maintains two dictionaries. The first, sequenceMap, maps a sequence of events of length n - 1 to a list of events that occurred next, such as

Build Tower->Repair Tower and Build Tower->Build Tower. The second dictionary, mostCommonEvent, maps each unique event sequence it has recorded of length n - 1 to an event chance pair. This event chance pair contains both the event that was found to occur most often and the number of times that it occurred. This allows the framework to determine the percentage of times that the most common event occurred. The training algorithm for a Sequence Analyzer can be seen in Algorithm 2.

foreach *Sequence of events of length n - 1 in the Data History* **do**

    add the sequence to the sequenceMap dictionary;

    add the event following the sequence to the list of events in that sequence's dictionary entry;

end

foreach *Unique Sequence of Events* **do**

    calculate percentage of times each subsequent event occurs;

    add most sequence and most common event to mostCommonEvent dictionary with its percentage;

end

Algorithm 2: Training a Sequence Analyzer

The Sequence Analyzer is used before the classifiers are called to return a list of actions. It is given a list of the last events which it uses to perform its analysis and must be passed to the framework. The Sequence Analyzer uses the second dictionary to quickly find the event that occurred most often given that sequence. However, it only successfully returns an event if that event is found to have occurred over 90% of the time that sequence was performed. In addition, the sequence must occur at least twice in the entire history. If no event is found by the Sequence Analyzer, the classifiers are used. If an event is found, it determines the location by using the

average relative position when comparing the last location in the event sequence to the event found. For example, for a build and repair tower sequence, the relative position of the repair action will almost always be (0, 0) since it takes place in the exact same position as the build action. Thus, in this example, we would return a location that is the last event location of the sequence.

One limitation of this approach however is when an action is logged continuously resulting in long chains of the same action. The effect on a companion would be that once it starts performing this action, it never stops. For example, if a move action was logged every time the player takes a step, the companion may get stuck moving indefinitely. The solution to this is to log that event less often, such as a move action every ten steps.

As with feature selection, training also requires some overhead. In order to minimize this, Sequence Analyzers are stored in a static list and used by all companions. In addition, they are constantly kept updated by adding new event sequences whenever an event is logged into the data history. This is done by simply updating the first dictionary in the Sequence Analyzer which in turn updates the second if a new event is found to occur most often given an event sequence. This allows us to avoid initializing a new Sequence Analyzer whenever a new companion is created and instead only create a Sequence Analyzer if one of its length does not already exist.

Lastly, a consequence of keeping track of the order of events requires a modification to the underlying data structure used to keep track of the game state and event pairs that was used in the original MimicA framework. The data dictionary has been converted to a list which holds the pairs of game states, the event, and also the world location. This allows us to track the order of events that took place. It also has little effect on performance as all operations are iterations over all the data. We do not perform any search operations which dictionaries are particularly useful for because

we have removed the ability to remove particular vector event pairs from the data history.

4.5 Feedback

Another improvement includes an improved feedback system. This system is designed to allow the players to stop companions from doing unwanted actions. In the original framework, actions could be banned by deleting the corresponding game state and event pair from the data history and then retraining the companion. Removing data history is not desirable because it could be useful in the future and retraining has an overhead and should be avoided when possible so this has been replaced with two different types of banning. These bans are stored in a list by their names. The first ban is temporary and should be cleared whenever an event is successfully found. Temporary bans do not necessarily cause a retrain and the framework uses this list to filter out banned events when the framework is about to return a list of events to perform. The second type of ban is a permanent ban. Whenever events are added to this list, the companion is retrained on all the current data minus those entries that contain the banned event.

With this type of feedback, players can have companions stop doing some actions if they are particularly malicious. One example is a "delete building" action which can be useful to players who accidentally build towers in incorrect locations or wants to sell buildings to get more immediate money. However, it is very rarely an action players would want autonomous companions to perform, making the ability to ban this ability valuable to players. Temporarily banning actions is also important because it allows players to prevent companions from doing something that they may want done in the near future. An example of this is spending money to build a tower while the player is trying to save up for an expensive upgrade. We make use of these two

lists to implement a better feedback system in Lord of Towers which we discuss in section 6.1.

The introduction of these types of banning, however, creates issues when all actions are currently being denied. In these cases, the companion is retrained and the default action is returned. Default actions are defined when the companion is created and defined by the developer. These should typically be events such as waiting or idling which the companion can do if there are no available actions to perform.

4.6 Random Forest classifier

Our last improvement was the addition of the random forest classifier. The goal of this was to add a strong classifier to see if it could improve the quality of decisions made. The random forest classifier is an ensemble learning method. Ensemble learning methods are different from single classifiers in that it combines multiple classifiers to solve the same problem and are able to boost weak classifiers into a strong classifier making them more accurate. In addition, ensemble learning suffers less from overfitting and has a decent generalization ability, meaning it has lower error when predicting previously unseen data [42].

Random forests when compared to other ensemble learning techniques has comparable accuracy to AdaBoost. In addition, random forest is also particularly resistant to outliers and runs quickly [7]. It works by choosing random subsets from the data with replacement and then training some number of decision trees each with one of these randomly chosen subsets. For classification, the random forest chooses the event that has the highest number of votes among the decision trees. Our algorithm trains ten decision trees by default each with 25% of the data, these however can be changed with input parameters. We then return all the resulting events picked by the forest but in descending order of votes received.

This classifier provides several benefits to our framework. Firstly, it returns a number of choices for events which is something that most of the classifiers we use do with the exception of decision trees. The problem that can occur in decision trees is that if the event returned is temporarily denied by the player or found to be invalid, the game will quickly ask the framework for a new action. However, the decision tree will likely return the same event as the game state has not changed significantly since its prior classification. If this happens too many times, the decision tree may be retrained, which requires some time. Another benefit of the random forest classifier is that it creates companions who will make different decisions. When multiple companions are instantiated at the same time, they will come to the same conclusion on what action to take when given the same game state vector. However, since the trees are trained on different data, the companions using the random forest classifier will not always make the same decision and can perform different actions. This creates the ability to have different companions without having to use different classifiers for each.

4.7 Updating the Original Classifiers

In addition to adding a new classifier, we also updated the original three classifiers: K Nearest Neighbor, Decision trees, and Naive Bayes. This was done as part of the overall refactor of the framework and also required as we changed several of the underlying data structures in the original framework such as the data dictionary.

Our changes to KNN and Naive Bayes mainly consisted of returning a larger number of possible events to return. For KNN, this list is as long as the number of actions but will not necessarily include every possible action as KNN will return similar actions but with different locations. For Naive Bayes, every possible action is now returned instead of the top five. This has the benefit of reducing the chance of

having to ask the framework for a list of events again if all the events are found to be poor choices.

Decision trees required a much larger refactor. In particular, the decision tree can now evaluate additional data types beside numerical values and booleans. This includes strings, enums, custom objects, and events. To handle strings, we convert strings to a number by summing their character values. While not optimal this simply converts them to an enum value. For custom objects, developers must extend the FrameworkObject class and implement the getValue() function to provide a numerical value for them. For events, the string name of the event is used and converted to an integer value. The last change made to decision trees is to dynamically choose better values to split the data on. Previously a hard-coded value of 10 was used since most of the data ranged from 0 to 20. We now keep a rolling average as we iterate through the list and split the data based on that. These changes make the decision tree classifier more flexible and applicable to other games.

4.8 Saving and Loading

The last change we made to the framework was to add the ability to save and load the data history. This is not only extremely useful for testing purposes but also is a general benefit to have in many video games. To do this, all the objects in the framework are serializable which allows the data to be written to an XML file or read out.

Chapter 5

FRAMEWORK

In this section, we detail the work that must be done to integrate successfully with our framework. In particular, we discuss the various responsibilities that developers must take care of as well as the tasks they must complete. While our updated framework requires some of the same information and approach as the original MimicA, it has been divided into a framework with clear hooks and classes that must be defined. This is a partially a result of a major refactor we performed in order to cleanly split the original framework from its integration with Lord of Towers, the example game used as a proof of concept.

5.1 Using the Framework

The first step to take is to define the boundaries of the game's map. The developer must define the bottom left and top right of the map in world coordinates. This is defined through the FrameworkGameData class's static method setWorldDimensions(). FrameworkGameData is the object that holds static data that can be accessed by all companions including the data history.

Next, a developer must initialize the companion's "brain" which is contained in the FrameworkCompanionLogic object. This requires several parameters including a list of forbidden events, a default event, and a list of location features. The list of forbidden events can be empty initially but is useful for defining any events that the developer knows the companion should never perform. The location features, as mentioned earlier, are used by the KNN classifier to determine a location for events. As optional parameters it takes the classifier to use, n-gram sequence lengths, and a

percentage to perform feature selection. The classifier defaults to decision trees if not passed in and feature selection percentage defaults to 10%. If n-gram sequence lengths are defined, they are used in the decision making process, discussed in section 4.4.

Developers must also define a game state vector class that extends FrameworkGameStateVector so that it can be saved to the FrameworkGameData class. Ideally a game state vector is saved whenever the player performs an action but the developer can choose to omit certain actions if they do not want this to be recorded. However, more data collected will allow the classifiers to perform better. To get a decision from FrameworkCompanionLogic, developers simply need to call getDecision() which requires the current game state, past events and their locations. The past events and their locations are required because the framework cannot track the past events of companions and they are not expected to be saved into the data dictionary as they are companion actions.

Next, developers must define the set of events that can occur within the game. These should extend one of two interfaces, FrameworkEvent or FrameworkLocationEvent. FrameworkLocationEvent extends FrameworkEvent and simply adds a FrameworkRegion as a member variable. FrameworkRegion are the regions discussed in section 4.2 and hold world coordinates to specify a portion of the map. FrameworkLocationEvents should be used typically if the event is location specific. In addition, these events have a variable, trainOnLocation, which is a boolean flag that determines whether the event's location is important. This defaults to true but when set to false means that the framework will not return a FrameworkRegion when that event is chosen. This should typically be set to actions that are specific to one location or to actions whose location does not matter, such as checking one's inventory or healing, if the healing location is only ever at one spot on the map. Note that since the framework returns FrameworkRegions to the developer, they must create logic for determining where the event should take place within the range of coordinates.

For example, when the framework gives the action to build a wall in a particular region, the developer must handle the logic to determine the best place within that region to build a wall with regards to the surrounding environment. This completes the required work that the developer must do to ensure that companions operate successfully.

There are additional parameters that could improve companion behavior. This includes the ability to restrict events which we noted earlier. To do this, FrameworkCompanionLogic provides functions such as addForbiddenEvent() and addCurrentlyForbiddenEvent() to permanently and temporarily restrict an action respectfully. Additionally, the framework allows for the developer to manually call retrain at any point to train the classifier on the current data stored in FrameworkGameData. The last two methods in the FrameworkGameData allow for saving of the class's data as well as loading and are named saveDataHistory() and loadNewDataHistory(). They require the filename of the XML file to save data to or read from.

Figure 5.1 is a UML diagram of the framework and displays many of the basic functions. It also displays the Companion, Event, and GameState objects which need to extend parts of the framework.

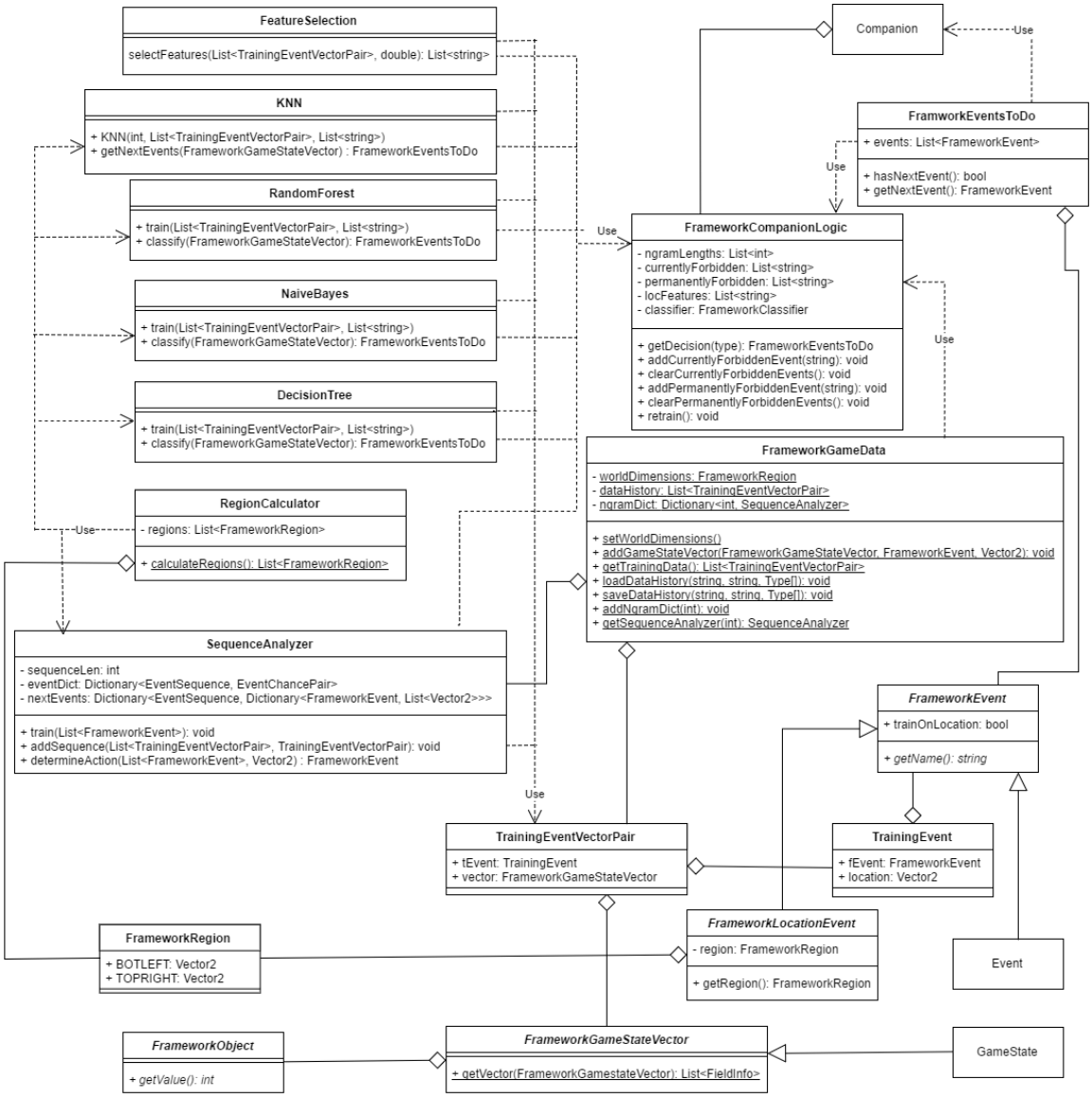


Figure 5.1: UML diagram of Framework

Chapter 6

CASE STUDIES

In this section we discuss two games that were implemented using our framework. The first is an upgraded version of the original Lord of Towers used in MimicA. The second game is a different game that involves map exploration and has different mechanics to provide another example of the flexibility of our framework as well as the application of mimicking behavior in companions.

6.1 Lord of Towers

Lord of Towers (LOT) is a rather unorthodox tower defense game that involves defending a town. The uniqueness comes from the fact that the player must control an avatar that moves through the 2D map to build defenses. This is not often seen in tower defense games though it does appear in some video games such as Dungeon Defenders [32]. In addition, the map, which is 24 by 12 tiles large, is entirely open with enemies able to traverse any tile on the map to reach the town. The player must build defenses to obstruct their path and kill them before they reach the base. These defenses include towers, walls, and trenches. While walls and trenches do not damage enemies, they serve as excellent buffers and cheap ways to block a path. Towers are the center of the defense as they are the primary tool for killing enemies. While most enemies will typically avoid attacking defenses, they resort to attacking buildings if all path to the town are found to be blocked.

While the player initially starts alone, as time passes, companion characters will join the player to help build and repair the defenses. Up to three companions join the fight with one spawning every three minutes. Companions will prompt the player

before performing most actions. However, when the second companion spawns the player dies allowing them to spectate until the enemy hordes inevitably overwhelm the defenses. The strategic part of this game is setting the groundwork of the base sufficiently enough to both fend off enemies as well as train the companions on what to do so that they can ultimately expand and maintain the defenses after the player is gone.

Figure 6.1: Gameplay from Lord of Towers

We made several large changes to the game in order to increase its complexity. This was done by adding new features to the game including expanding the actions players (and companions) can perform as well as adding new enemies to the game. These changes are also accompanied by several edits that are aimed at making the game easier to play or fixing behavior. Lastly, we performed a visual update to the game that involved replacing most of the art for animating sprites and adding sound effects. A complete list of changes can be seen below.

- Visual update to character sprites, animations, sounds effects
- Mine and Delete Tower Action
- New Enemies: Archer, Golem, Wizard, Vampire
- Tower Upgrades: Damage, Slow, Range, Area of Effect
- Upgrades to Tower upgrades
- Updated logic for determining location in events
- Feedback System with location indicator of event location
- Added UI for toggling companion behavior

6.1.1 Enemies

The first major change made to LOT was the addition of four new types of enemies. This increases the diversity of enemies while introducing new considerations to the game to increase its overall difficulty. The original three enemy types were kept and updated as well. They include zombies who simply try to run at the town, fire spirits which aims for buildings the player has created and self-destruct to do significant damage, and skeletons who try to attack the player and companions.

The new enemies bring some new features to the game. The archer is a ranged low-health unit that targets players, companions, and towers. If ignored, it can deal decent damage. The golem is a slow but high-health unit which functions similar to a zombie. He makes a great distraction for towers to focus on while other enemies slip by. The vampire is tricky unit that can teleport every so often. This allows him to potentially bypass parts of the defense while he tries to attack the town. Lastly, the wizard is a support type character who casts a healing spell that restores health to all enemies within its radius. While slow and weak, they can make large groups of enemies extremely dangerous.

With the introduction of new enemies and events (described in subsection 6.1.2), the balance in LOT was readjusted. This was primarily done by increasing the growth rate of enemies over time. For example, zombies gain more health, vampires teleport

farther, and wizards cast more potent spells. This change ensures that the game is still relatively easy early in the game while getting progressively harder over time. In addition, the number of zombies per wave was reduced to ensure that there was a constant mix of all enemy types at any point in the game.

6.1.2 Events in Lord of Towers

This section details the updates that were made to events in LOT. This includes updates to existing actions in the game as well as the addition of new events. A complete list is available in Table 6.1. In particular, much of the changes to older actions were aimed at choosing better locations for some actions. While regions were integrated into the framework, LOT still maintains six equally divided sectors. To convert a region given by the framework into a location, sectors are chosen based on which has the most tiles in the region.

The biggest change to existing events was the update to the logic for building towers, trenches, and walls. This was done in order to improve location picking and reduce the number of times a companion was unable to decide where to build something. In addition, logic was added to prevent companions from building a structure that would completely obstruct a path to the town as this would make enemies forcibly open a path by attacking buildings. It also prevents companions from ruining strategies that involve funneling enemies through a specific path. An example of this is a winding maze-like path that maximizes enemy exposure to towers.

The new actions added to the game shift the balance from focusing on the quantity of towers to cultivating existing towers in order to make them more powerful. Four of these new actions involves choosing specialties for a tower such that each gains a unique ability. These four types are range, slow, damage, and area-of-effect. The range tower gains significantly more range and damage but loses some fire rate. The

slow tower adds a 50% slow to an enemy for a couple seconds. The damage tower adds a percentage of its current damage to each shot. Lastly, the area-of-effect tower adds a small radius that hits surrounding enemies when it lands a shot. In addition, each of these towers can be further upgraded by fire rate and special ability. Upgrading fire rate was a mechanic taken from the original LOT. Upgrading the special ability of a tower adds some damage to each shot and increases the ability's potency (such as a larger slow or more range). While towers can be upgraded multiple times, the cost increases linearly with each upgrade. These upgrades are fundamental to building a strong base that can weather the onslaught of enemies.

We also added several other new actions to the game. One of these is the ability to delete towers. This was primarily introduced as a way to remove towers that were accidentally created or to remove a buildings created by the companion. This also provides a great example of an action that players can forbid companions from performing.

The last action we added was the "Mine" event. This event can be performed in the bottom left corner of the map on the stone quarry. While it can only be occupied by one person at a time, it can generate income when resources are scarce. When a player or companion mines, they generate ten gold a second. This creates an action that is useful and is a good example of an action that is generally always helpful.

6.1.3 Feedback System

To match the updates to the feedback system in the framework, the UI revolving prompting player regarding player actions was also upgraded. In the original LOT, if the player is alive, companions will ask the player if they can perform an action which the player can then confirm or deny with no indication of where it will occur. This was given a massive rework in order to give more information to the player. Firstly,

Table 6.1: Events in Lord of Towers

Event List in Lord of Towers		
AOE Upgrade	Heal	Repair Wall
Build Tower	Mine	Slow Upgrade
Build Trench	Move	Upgrade Tower Ability
Build Wall	Range Upgrade	Upgrade Tower Fire Rate
Damage Upgrade	Repair Tower	Wait
Delete Tower	Repair Trench	

these requests are now placed in a queue to prevent requests from stacking on top of each other and timing out when there are multiple companions. Players can respond to these requests by answering: 'Yes', 'Not Now', or 'Never'. Yes accepts the action while answering not now temporarily bans the action at that location, and never permanently bans the action. Most importantly, requests are now accompanied with a blue tile that appears on the map to specify where the companion wants to perform that action. Figure 6.2 provides an example of this. Thus instead of simply asking to build a tower, the companion will now ask to build a tower at a specific tile. LOT keeps a dictionary of events and a list of the most recently banned locations. These lists have a maximum capacity of five and discard the oldest location when full. Thus player are able to execute a higher degree of control over these companions, especially in terms of location.

In addition to these changes, players can also toggle companions behaviors. They can choose to accept all companion actions which makes companions stop prompting for each action and allows them to perform any action they want. The other option is to force companions to stop performing any new actions for ten seconds. This is particularly useful if the player has a specific goal in mind and wants to save resources. This makes the game more playable and prevents the player from simply managing

Figure 6.2: Feedback System in Lord of Towers

companion behavior.

6.2 Lord of Caves

In addition to LOT, we created another game to prove the framework's flexibility and ability to apply to other games. To do this we built a sister game to LOT, called Lord of Caves (LOC), which focuses on map exploration. In LOC, the player again controls an avatar who starts alone and must explore a procedurally created 2D map covered in fog of war. The goal is to find and destroy zombie spawners spread through the map. These spawners constantly create zombies who roam the world trying to feed on the player.

LOC contains some of the same actions as LOT in that towers and walls can be built. However, they can not be upgraded, are more expensive, and weaker compared to towers in LOT because this game is less about building a base and more about

moving around the map and killing zombies. These buildings remove fog of war and can act as watch posts to alert the player of a zombie's position. The combat system in the game is different as well. Players able to manually attack enemies around them by pressing the spacebar. In addition, structures now cost gold and wood. These are rewarded when zombies are killed but can also be collected from resources around the map. These resources are trees and mines which populate the map and are procedurally placed by the generator. In addition to providing resources, trees and mines also create natural barriers to block off zombies. However, zombies can still remove these obstacles though at a much slower pace.

Companion houses also spawn around the map. These houses spawn a companion who joins the player on their journey when the player interacts with their house. These companions can perform much the same actions as the player but their locations are centered around the player. For example, their attack action will attempt to attack the nearest zombie to the player. They can also explore the map for the player and attack spawners. It should be noted that the game ends only when the player and all companions have been killed so companions can also complete the mission for the player if they fall in battle.

While they may look similar to LOT since most of the art is reused, it has fundamentally different tactics and is much more player-centric. Instead of a tower defense game, LOC is an adventure that changes with every playthrough because of the procedural generation algorithms used to create the maps.

6.2.1 Integrating the Framework

We demonstrate the ease of integration by describing the process taken to connect Lord of Caves with our framework. The major steps of this include defining both events that occur in the game as well as creating the game state. For a more step-

Figure 6.3: Gameplay from Lord of Caves

by-step approach to using our framework, refer to Appendix B.

Development for this game began with the ability to create a map, which is procedurally created using a variety of techniques including quad trees and cellular automata. Next we made sure to pass the size of the map to the framework through the setDimensions method. After this, we instantiated the player and defined the set of actions that can occur within the game. These are the events that must be defined for the framework so we extend FrameworkEvent and FrameworkLocationEvent. Note that important to this step is the choosing the location of the event given some bounds. Particularly, these events that were defined took a FrameworkRegion and found some position within that space to perform the action. For example, for the BuildTower event in LOC, we searched for a choke point to build the tower.

Another important step is defining the game state vector. This must extend FrameworkGameStateVector and be passed to FrameworkGameData whenever the

player performs some action. After doing this, we defined a set of variables for the game. In total we created about sixty-six different features which include varying things such as player health, number of nearby trees, and number of resources.

The last step of this process was creating the companions. This process involved instantiating the FrameworkCcompanionLogic in the companion and then calling getDecision() with the correct parameters. In addition, work was done to process the returned list of events and enable the ability to ban an event. Lastly, we use set some optional parameters, specifically using the random forest classifier and n-grams with sequence length of two.

To note, we left several important steps in game development absent from our discussion because they were not directly involved in the integration process. This includes steps such as finding art (which was directly taken from Lord of Towers), game balancing, and UI creation. These steps were primarily performed prior to our integration with the framework.

Chapter 7

EXPERIMENTAL DESIGN

In this section, we discuss the evaluation methods we used in order to test our system. This consisted of two parts. First, we created automated simulations in order to analyze the effectiveness of feature selection and the random forest classifier. Second we performed a user study with a group of students in order to get feedback and opinions on the companions' usefulness.

7.1 Automated Simulation Setup

In order to test the effectiveness of the feature selection and random forest classifier, we ran automated tests using Lord of Towers (LOT). This consisted of creating seven different testing configurations each of which would be run ten times on each classifier. A complete list of configurations can be seen in Table 7.1. By using multiple configurations, we ensure more variability and represent different strategies used in LOT. Each configuration and classifier pair was also run a total of three times with feature selection set to differing levels. The first was set to have no feature selection, the second with 10% of features removed, and lastly with 30% of features removed. Thus each configuration had 30 iterations per classifier. Tests consisted of having a scripted player performing some set of actions before starting the enemy waves and having companions spawn. Each test spawned up to three companions and the town started with the default 20 lives.

Each test was run at 4x speed which is possible in Unity by setting the time scale. While increasing this further would complete testing much quicker, we noticed that running the tests at 32x resulted in noticeably shorter game times. Thus we chose 4x

in the interest of time, though running at normal speed would be optimal.

The first two configurations involved having the scripted player choose decisions randomly. This consisted of picking fifteen actions randomly before starting the enemy waves. Then the player continues building randomly until either dying or the first companion spawns which occurs every minute. Location is also chosen randomly. It should be noted that the available actions include every single action in LOT including deleting towers and mining. Some actions, such as upgrading a tower's ability or creating a slow tower which require a tower to exist prior also build a tower if no available towers were found on the map. The first of these random configurations picked all actions with equal probability and could choose any tile on the map except the left-most column and couple right-most column. This led to greater variability in the configurations created and provides a baseline to compare other configurations against. The second random configuration was constrained. Specifically, basic actions, such as building a tower and wall, were weighted to occur more frequently. In addition, we only forced locations to be chosen within a tighter range to ensure a denser setup.

The next four configurations included setup of buildings and upgrades before starting the waves and removing the player. These configurations made use of every action in the game except mining, waiting, and deleting a building to give companions training in most actions. Companions then spawned every five seconds meaning each would have access to the same data history.

The first configuration called Single Wall can be seen in Figure 7.1. It consists of a rather powerful setup that focuses much of its actions in the back of the map. The single wall in the middle has a small gap in the middle and allows enemies to come from three different sides. This effectively splits them.

The next configuration can be seen in Figure 7.2 is called Spread Out because a set

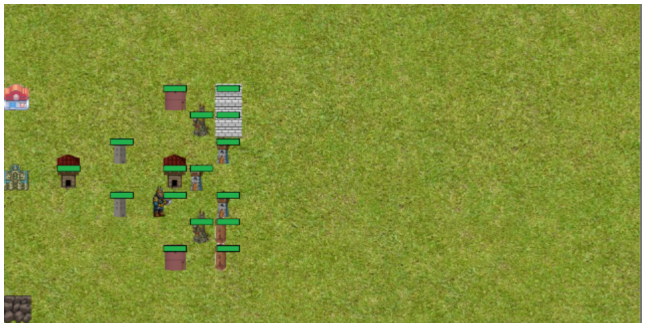


Figure 7.1: Configuration for Single Wall

of buildings consisting of a wall, trench and two towers is built in each sector. These towers were also upgraded to a specific path. This type of configuration distributes actions across the map evenly dividing the regions.

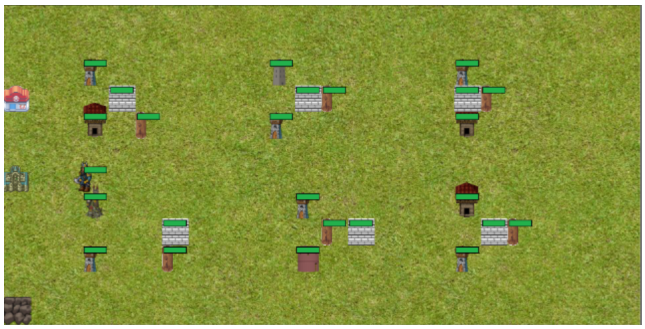


Figure 7.2: Configuration for Spread Out

The last two of these configurations created the exact same formation of towers. However, the second configuration moved all buildings exactly nine tiles forward, which equates to being in the middle two sectors of the map. These two configurations can be seen in Figure 7.3 and Figure 7.4 respectively. In addition to building two walls a couple tiles apart, there is another placement of towers behind them. The main goal here was to see if moving the base forward would have a positive or negative effect on location picking.

The last configuration consisted of building a winding line of towers and walls starting from the left. The fully finished configuration can be seen in Figure 7.5

Figure 7.3: Configuration for Split Wall

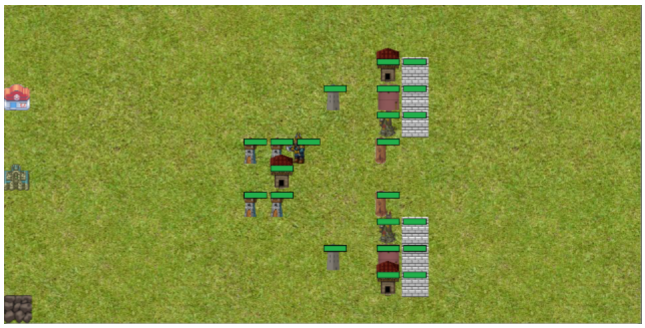


Figure 7.4: Configuration for Split Wall Forward

but it almost never reached this point because enemy waves began spawning as the second row began construction. This gave them time to damage parts of the walls and potentially kill the player before it was complete. This strategy is typical of many tower defense games with open maps because it maximizes the amount of space an enemy must travel before reaching the goal. This configuration only includes building towers, walls, and sometimes trenches and did not use any upgrades. Also, companions spawned every two minutes with the player dying after six minutes. This means that each companion would have different amounts of information in their data history. This configuration is therefore able to best mirror a typical game in LOT.

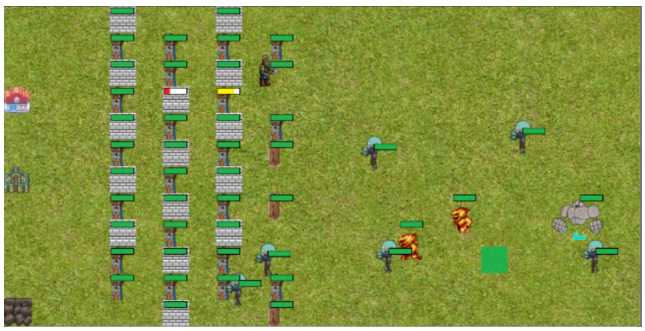


Figure 7.5: Completed Configuration for Winding Wall

7.1.1 Testing Feature Selection and Random Forest

In order to test feature selection during our automated simulations, we needed to record how long the companion took to make decisions. To do this, we utilized C#'s Stopwatch class. The stopwatch was used to keep track of how long it would take to get a list of events back from the framework's getDecision() method. We also kept track of initialization time as feature selection requires some overhead. Thus, we started the stopwatch when the companion is first instantiated and stop it after the first call to getDecision() is returned. This allows us to compare the trade off with using features selection to determine if this improvement was able to cut decision making times. Note that LOT has 247 different variables in its game state vector and as such removing 10% culls 24 features and 30% removes 74.

The length of each game was also collected in each iteration of the testing. This can be used to compare the ability of classifiers to make decision and determine if our new classifier, Random Forest, was able to pick better decisions and create stronger bases that would result in a longer game. In addition, we can compare game lengths between the different levels of feature selection for each classifier and configuration to see if the search space was refined using this method.

Table 7.1: Description of Testing Configurations

Description of Automated Simulation Configurations	
Random	Equal chance for all actions and any tile except the right and left most columns. Player dies once a companion spawns
Constrained Random	Random actions but higher chance to perform useful actions. Tighter bound for location. Player dies once a companion spawns
Single Wall	One single wall with a small gap in the middle and upgraded towers in a checker pattern to allow enemies to pass through. Player dies once waves begin
Spread Out	Two upgraded towers, wall, and trench in each sector. Player dies once waves begin
Split Wall	Two small walls with trenches on the side. Upgraded towers are behind these walls and a semi-circle of towers is farther back. Player dies once waves begin
Split Wall Forward	Same as Split Wall configuration but 9 tiles to the right. Player dies once waves begin
Winding Wall	Maze-like configuration made of non-upgraded towers, walls, and trenches. Enemies spawn as the second row is started and player dies once second companion spawns

7.2 User Study

For our user study, we had participants play Lord of Towers and then take a survey. The complete survey can be seen in Appendix A. Participants were gathered from a computer engineering class, Interactive Entertainment Engineering which studies video games and game development. It was filled primarily with computer science students.

We had our participants play the game for twenty minutes to ensure they would get at least two plays of the game. Each participant was assigned a letter designating which classifier they were to play for the duration. After time passed, the participants then completed the survey through Google Forms. Lastly, participants played the game one more time on a different classifier. It should be noted that the participants were split in half and also played another variant of Lord of Towers. This means that roughly half the participants were familiar with Lord of Towers prior to starting our study.

Chapter 8

EVALUATION RESULTS

This section details the results of the evaluation performed including both the automated simulations and the user study. For the automated results, we are able to see how the random forest classifier compares to the other classifiers as well as if our feature selection was successful in reducing decision making times while improving the quality of the features set. For the user study, we analyze the responses given by the participants to determine if our companions were useful.

8.1 Testing Random Forest

In order to test the effectiveness of the random forest classifier, we ran it multiple times over seven different configuration with ten iterations at differing levels of feature selection. We look at the results of our tests in this section to determine if our additional classifier was able to outperform the other classifiers by looking at each configuration. The results are displayed in a chart that depicts the average time survived in seconds over the thirty iterations per classifier.

We first look at the random configurations in Figure 8.1 and Figure 8.2. In the random configuration, results varied heavily as sometimes games would last only about a minute since the initial setup was poorly structured and ineffective. In other cases, the random actions created a decent base that was able to stand for a longer period. In this test, KNN was able to score the best on average with decision trees and random forest performing about the same. Naive Bayes performed significantly worse than the others on average. This trend is similar in the constrained random test with KNN performing the best again. However, random forest outperformed

decision trees by about twenty seconds and Naive Bayes performed worst again. As a baseline, it shows that KNN is typically the strongest performer with decision trees performing well, random forest performing near the top, and Naive Bayes scoring the worst.

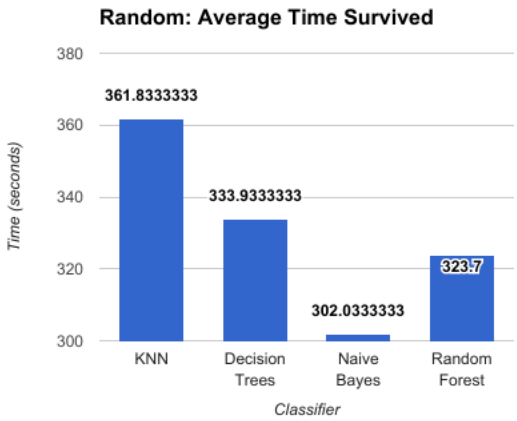


Figure 8.1: Random Configuration Times

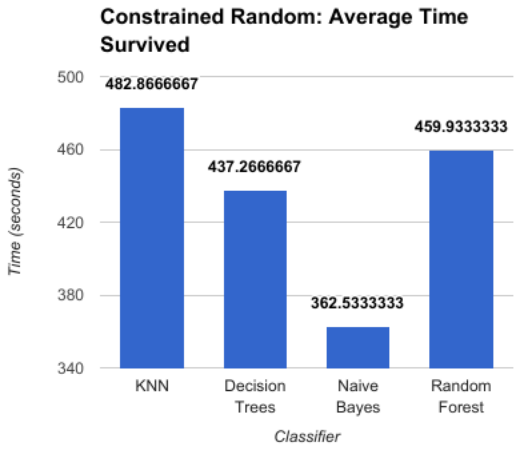


Figure 8.2: Constrained Random Configuration Times

The next group of configurations consist of those in which the player created a configuration of towers before starting the waves. These tests ran for much longer given that they made much more intelligent use of building choice and placement

and also used most actions available in LOT. These configurations did not follow a particular trend, compared to the random tests earlier.

In the single wall configuration, in Figure 8.3, we see KNN once again perform the best but only slightly better than random forest. The other two classifiers performed rather poorly, lasting over a minute less on average. In the case of Naive Bayes, it seems to focus primarily on building walls and trenches which becomes less effective as the game progresses.

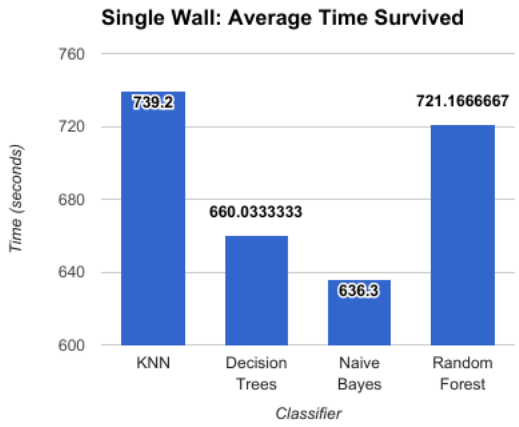


Figure 8.3: Single Wall Configuration Times

Random forest performed extremely well in the spread out configuration, shown in Figure 8.4. On average it lasted about a minute longer. The other classifiers performed within a minute of each other with decision trees performing the next best. It was interesting to see KNN perform poorly given the previous tests. After some review, this may be because most of the actions performed were focused near the right side of the map and many of the companions were killed quickly by skeletons.

In the split wall configuration, Figure 8.5, random forest again performed very well. In addition, Naive Bayes did surprisingly well. KNN and decision trees performed worst, averaging about a minute less. This is because Naive Bayes used the strategy of upgrading tower ability or fire rate which was surprising effective in this

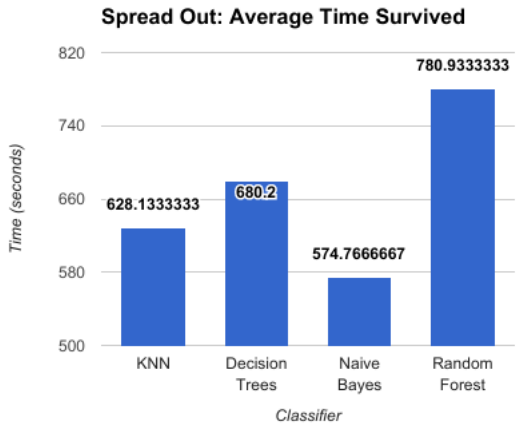


Figure 8.4: Spread Out Configuration Times

configuration.

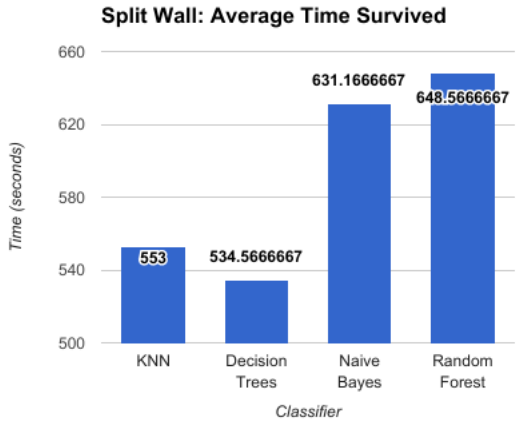


Figure 8.5: Split Wall Configuration Times

The split wall forward configuration, Figure 8.6, resulted differently with Naive Bayes performing very well, similar to the split wall configuration. On the other hand, random forest performed much worse and scored similar to KNN which it narrowly outperformed. This seems to be because, in this configuration, the companions seemed to focus building walls and towers instead of building tower and upgrading them as they did in the split wall configuration.

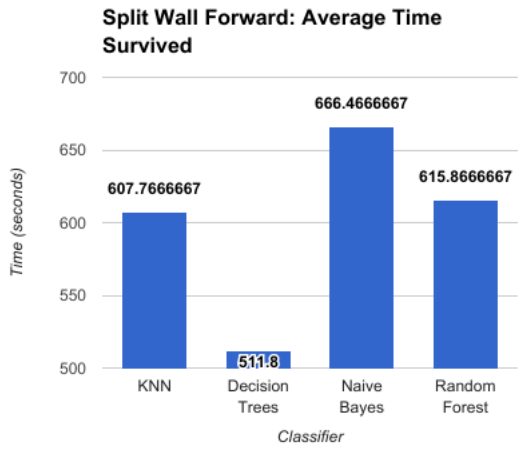


Figure 8.6: Split Wall Forward Configuration Times

The last test, winding wall configuration seen in Figure 8.7, was most similar to an actual game since player still built towers while enemies waves came and companions spawned at different points during this period. In this test, KNN performed the best with random forest performing only slightly worse. It was also interesting to see here that these times were, on average, about the same length of the random configuration meaning that the companions did not survive long.

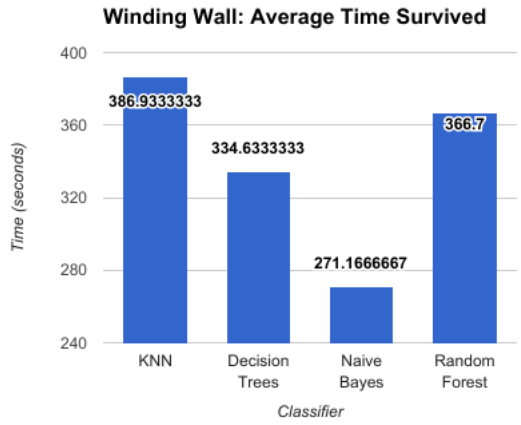


Figure 8.7: Winding Wall Configuration Times

These results highlight that different classifiers excel at different tests and have different strengths. KNN performed very well in our tests and scored the best on the

most configurations though on the other configurations it scored third. Naive Bayes performed extremely poorly on most tests but did extremely well on the split wall configurations. Decision trees was typically in the middle and never performed the best. Random forest performed the best in several of our tests and second in all others except one. Thus, the random forest classifier represents a significant improvement because it is consistently a strong classifier. While it may not be the best classifier in each case, it typically scores in the top and its performance varies much less than the other classifiers.

8.2 Testing Feature Selection

We can analyze the effects of feature selection in two parts. First, we can see if feature selection was able to improve the speed with which decision times were made. We hypothesize that increasing the percentage removed causes initial setup and training to take longer but will also reduce the time to make decisions. In the graphs below, we look at the effects of feature selection, in terms of decision time, on each classifier in terms of its initial training time and the run-time afterward to make decisions. It is important to note here that the companion does not retrain during any of the tests. Second, we can look at the effects of different levels of feature selection to see if companions made better decisions as the percentage of features removed increased. Success is measured again by overall survival time.

8.2.1 Feature Selection effect on Decision Making Time

For the KNN classifier, we can see in Figure 8.8 that training times were heavily impacted by the features selection with as much as a 400% increase. In addition, the decision making performed during run-time was not affected and even worsened. Thus, feature selection had a negative effect on KNN in respect to time to make

decisions.

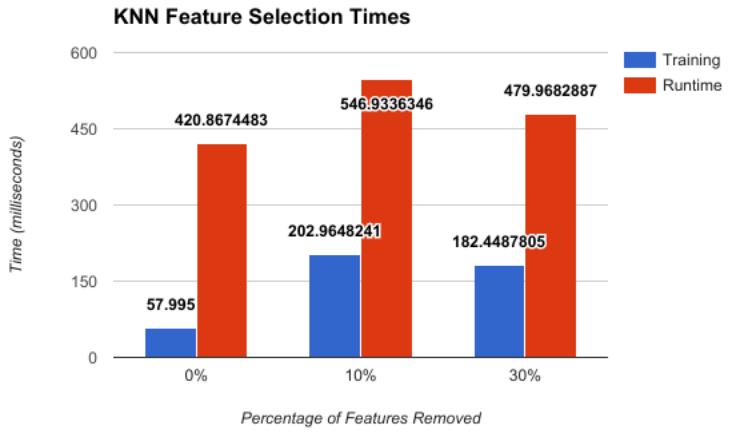


Figure 8.8: K-Nearest-Neighbor Times to make decisions

Similar to KNN, feature selection had a negative impact on the training time of decision trees as displayed in Figure 8.9. However, the effect was to a much smaller degree with times only increasing by about forty seconds. In terms of run-time, the effect was very small as the time it takes to make a decision in these cases is very short. It should be noted however, that decision trees made many more calls to getDecision(). While the other classifiers typically had about between 200-500 calls depending on configuration, decision trees typically had about 1000-2000 calls and some even over 10,000. While this may be acceptable given the short amount of time it takes to get an event from the tree, it does highlight the weakness that only one action is returned every time. This massive increase in calls is partially a result of the game state not changing in the short amount of time between calls to the framework resulting in the same event being chosen.

Naive Bayes exhibited times that we had initially expected to see when implementing feature selection as seen in Figure 8.10. That is, while training times were increased, the overall run-time decreased as the number of features we used decreased thanks to feature selection. However, it should be noted that the increase in time

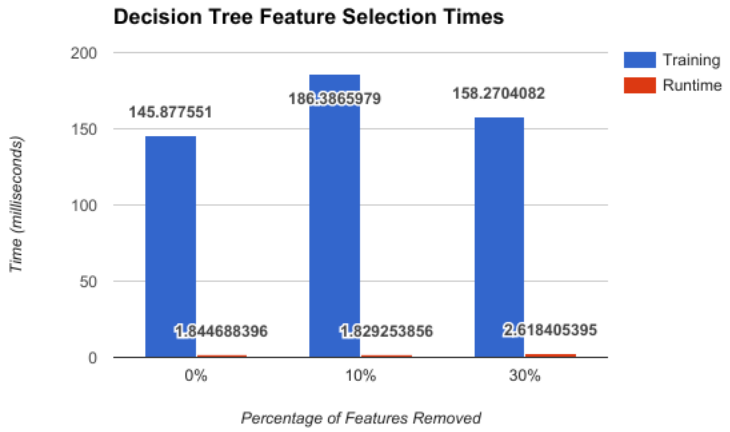


Figure 8.9: Decision Tree Times to make decisions

was much larger than the decrease in run-time decision making. This trade-off may be more acceptable in cases where the companion can have more time to train while in game, decision making time needs to be much faster.

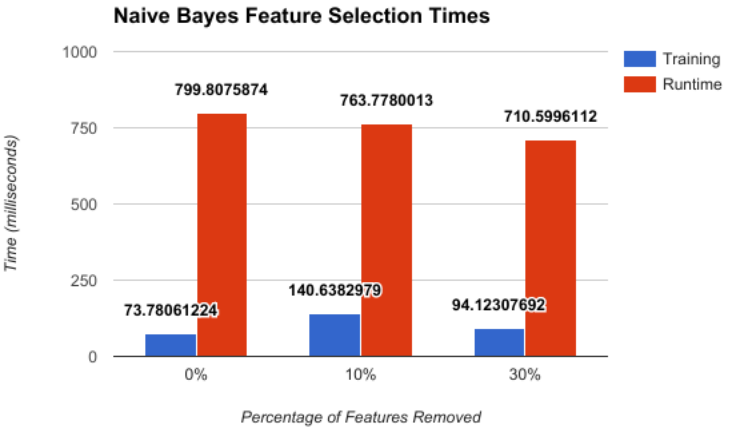


Figure 8.10: Naive Bayes Times to make decisions

Random forest in Figure 8.11 also displayed results that supported the idea that feature selection would reduce overall run-time. In this case, we saw much greater decreases in decision making time while initial training increased by a much smaller margin. This may be the result of having to train the decision trees on less informa-

tion.

Overall, it seems that feature selection's effect on decision making time depends on the classifier being used. From our tests, we see that feature selection adversely effects on KNN, has little effect on decision trees, but is able to make improvements for both Naive Bayes and random forest.

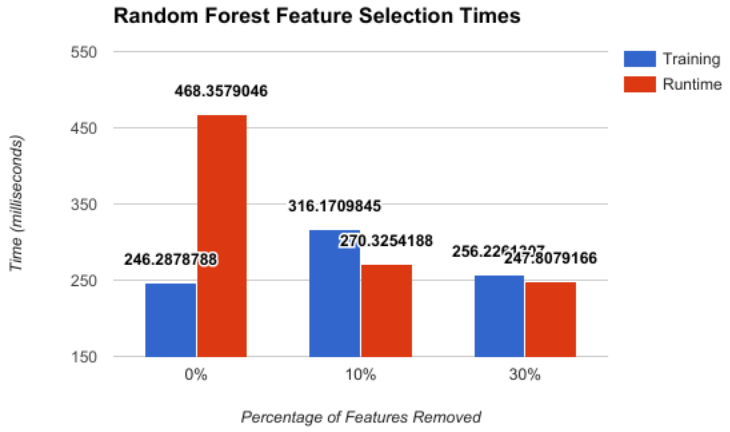


Figure 8.11: Random Forest Times to make decisions

8.2.2 Feature Selection effect on Performance

Next we compare the effectiveness of feature selection in being able to cull low information features to refine the data set. To do this, we measure performance based on the total time survived each in a game. We analyze them again using the seven configurations discussed earlier and with feature selection percentages of 0%, 10% and 30%.

For KNN, a common trend does not emerge as we can see in Figure 8.12. Using feature selection to remove 10% of features seems to either improve behavior slightly or cause it to worsen by a wider margin. On the other hand, removing 30% of features causes a larger change to survival time and for the most part has a negative impact.

The exception to this is the trend seen in the winding wall configuration where we see feature selection improve the classifier's performance. However, overall, it seems that feature selection adversely affects the performance of KNN.

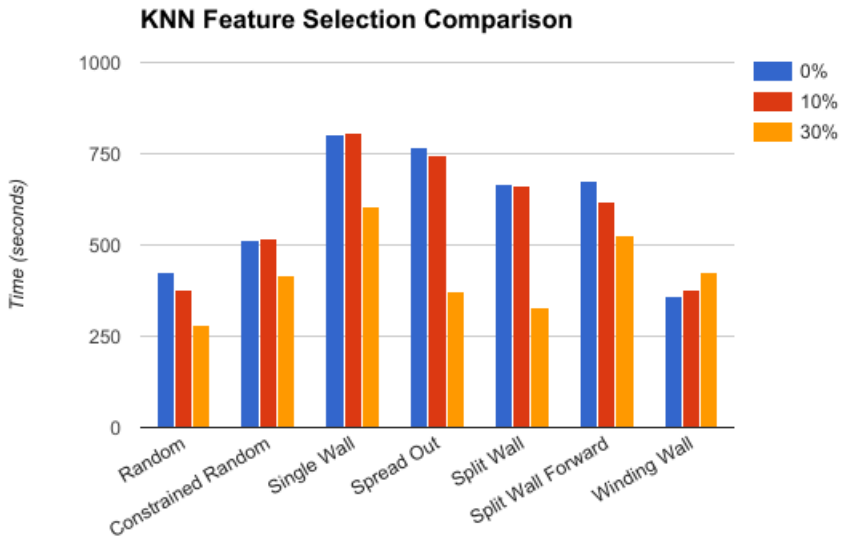


Figure 8.12: KNN feature selection comparison

Decision trees also display no common trend across the different configurations as seen in Figure 8.13. In some cases, including the random, constrained random single wall, and split wall configurations, feature selection seems to improve performance. On the other hand, feature selection results in worse performance in the others. The effects of feature selection on decision trees is therefore very mixed and may indicate that many of the variables present in our game state vector are useful and thus, should not be culled.

Feature selection has a mostly negative impact on the performance of the Naive Bayes classifier as well as seen in Figure 8.14. In a majority of the cases, feature selection reduces the performance of Naive Bayes which can be seen in the random, constrained random, spread out, and split wall forward configurations. In both the split wall and winding wall configurations, there is little to no change in performance.

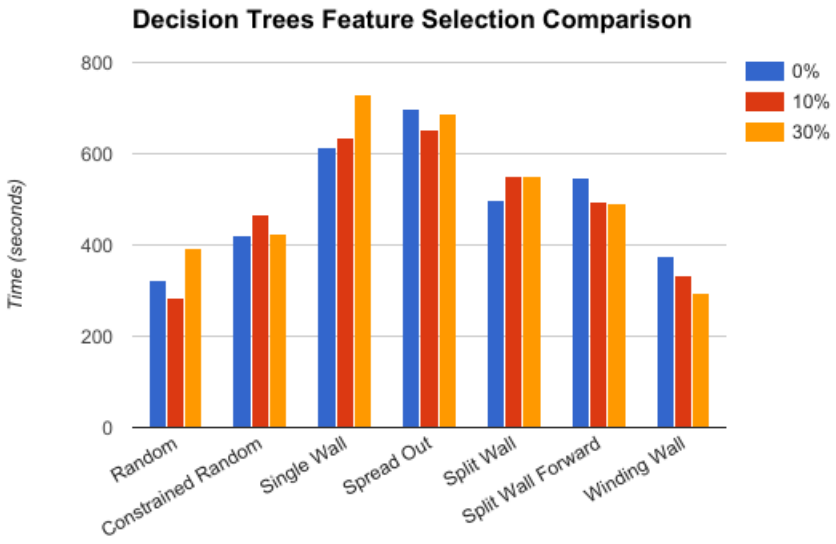


Figure 8.13: Decision Trees feature selection comparison

The exception to this pattern is the single wall configuration where feature selection results in a decent improvement in performance. Thus, feature selection does not seem to have a positive effect on the classification ability of Naive Bayes in our framework.

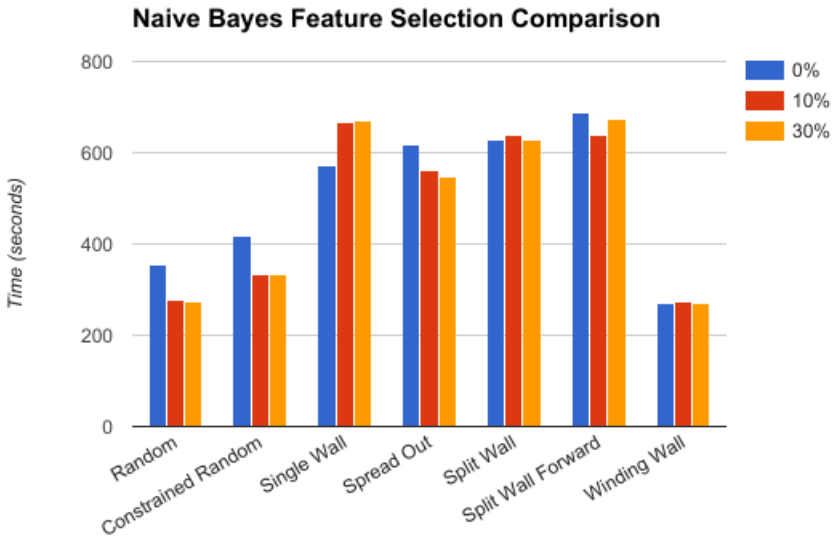


Figure 8.14: Naive Bayes feature selection comparison

Random Forest is an exception to the previous cases where feature selection proved

to primarily reduce performance. The results for random forest can be seen in Figure 8.15. With the exception of the random configuration, feature selection at 10% seems to offer an improvement in performance or have no effect and keep constant performance in the other configurations. Feature selection at 30% seems to vary a bit more with it either offering a slight improvement or reduction in overall time survived. It seems that the ideal percentage of features to remove with this classifier in LOT is 10%.

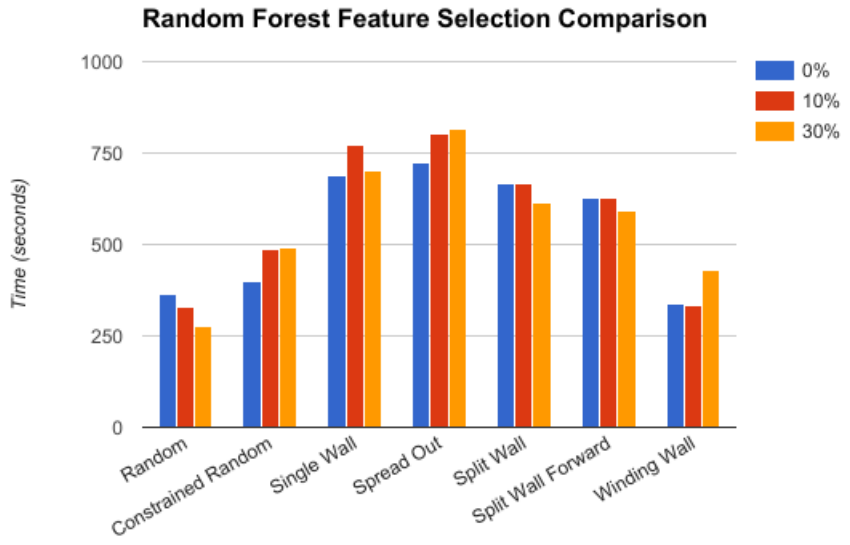


Figure 8.15: Random Forest feature selection comparison

For a majority of cases, feature selection seems to reduce the performance of the classifier. This may be the result of several reasons but on a whole this may show that many of the game state features were valuable in classification. However, there are a number of cases in which feature selection was shown to have a positive effect. This means the developer should test different combinations of classifiers with varying levels of feature selection to find the pair that best fits the companion and the level. Thus, it is useful to be able to specify the feature selection levels in our framework or toggle it off entirely. In conclusion, it seems that feature selection had

little beneficial effect on classifiers when used with Lord of Towers with the exception of random forest which both decreased decision making time and improved overall performance. However, it is important to note that feature selection is heavily reliant on the feature set. This is defined by the game developer and as such its effects would vary depending on the game. In the case of Lord of Towers, most of the features are useful and needed for classification.

8.3 User Study

In our user study, we asked participants to give their opinions in several areas primarily focusing on the usefulness of our companions and the framework. We also asked for their opinions on Lord of Towers. For each of our analysis points, we split our results into groups based on when they played the game. This is because half of our participants played another variant of Lord of Towers prior to participating in our study.

The first part of our survey focused on whether our participants enjoyed playing Lord of Towers. This was meant to see if they had a satisfactory overall experience and

if our updates to the game could evoke a positive response. For the most part, participants enjoyed playing Lord of Towers which can be seen in Figure 8.16 where a majority of participants agreed. To note, the first group's responses had a higher variation but ten of sixteen enjoyed the experience while in the second group no one disliked the game. When asked further, eleven participants responded that they found the game simple and fun. On the other hand, five noted the game felt too linear in terms of strategy and another three felt the game was too difficult.

When asked about the strengths of the game, many stated that they enjoyed the variety of enemies and towers as well as the mix mechanics present in the game. More

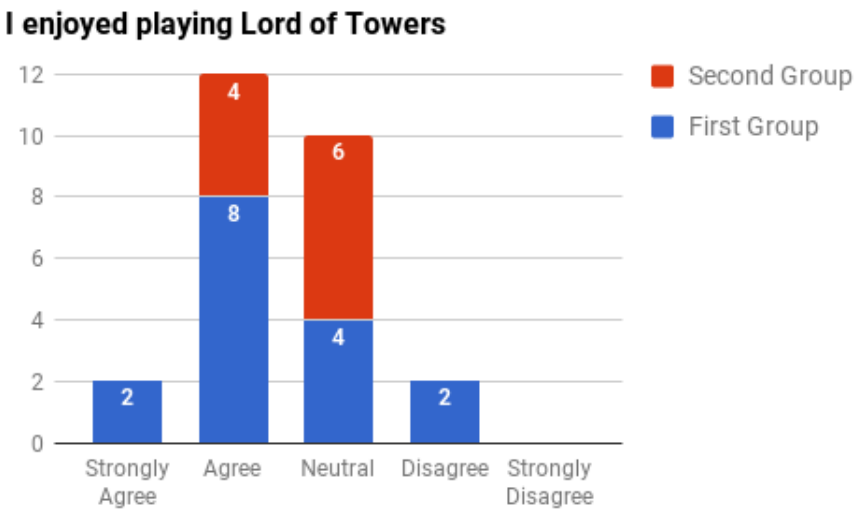


Figure 8.16: Survey Response for Enjoyment of Lord of Towers

importantly, four directly mentioned that they felt the companions were useful with one specifically mentioning they felt the companion was mimicking them. This is important because at this point in the survey, we have not mentioned companions at all. In regards to the responses about the game's weaknesses, a majority talked about the gameplay. In particular, many felt the game was too difficult, linear in strategy, or that the game was too fast. In regards to the game speed, this would occur after the player died once and was meant to reduce the amount of time players would have to watch their companions (which could be very long if trained well). However, this would not toggle off in subsequent games. While this can be difficult for new players, it should be partially alleviated as they play multiple times for the duration of 20 minutes. A table of sample responses and graph of the total count of each type of response can be seen in Figure 8.17 and Table 8.1. Despite this, most players still had a positive experience playing Lord of Towers.

Our next questions asked about perceptions of the companion characters. These questions were mainly focused on determining if the players felt the companions were learning from them and being useful. For the most part, these responses were

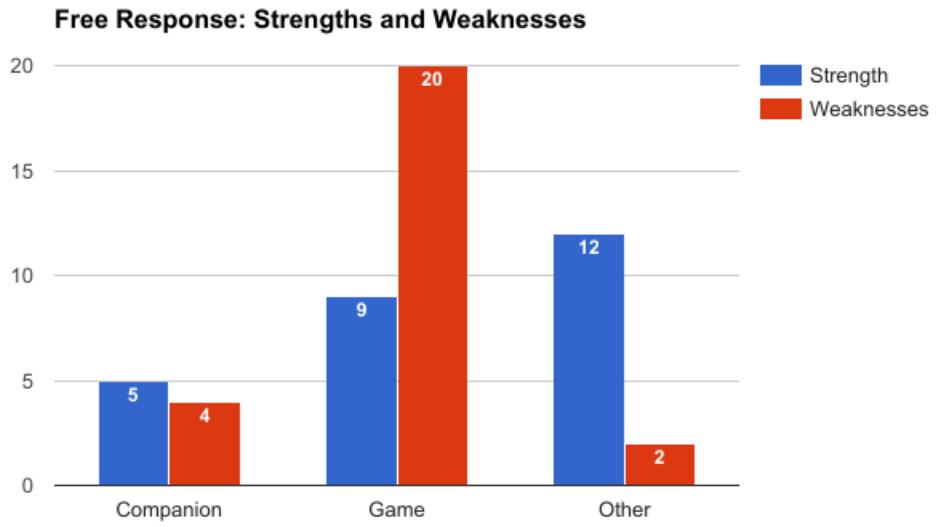


Figure 8.17: Free Responses to Strengths and Weaknesses of Lord of Towers

Table 8.1: Sample Responses for Strengths and Weaknesses of Lord of Towers

Type	Code Category	Sample response
Strength	Companion	The AI could sometimes mimic the player very well
	Gameplay	it was fun to play and try to come up with a strategy, nice variation in the defense types
	Other	The animations were great and once I got used to the controls, playing became fluid and fun
Weakness	Companion	Player death in 6 minutes. Companions were pretty useless.
	Gameplay	very fast paced and frantic (hard at the start with limited resources)
	Other	Having to walk to where you want to build

positive. We present two graphs for each group in Figure 8.18 and Figure 8.19. In both groups, we see similar trends though those in the second group found the companions more useful on the whole but were more neutral about companions spending money inappropriately. In total, seven participants noted that they felt that companions were very helpful and emulated their actions. Some of these participants also mentioned location, which they felt was accurate most of the time. Surprisingly, we found a high number of neutral responses. However, upon further inspection, we found that six participants noted that they either were unable to get to the point where companions joined (three minutes into the game) or did not observe them enough to have opinions. As a result, these participants primarily answered neutrally for these questions. This is partially a result of the previously described problem regarding game speed but can also be attributed to players focusing more on playing the game instead of watching their companions. Despite this, these responses highlight that the companions were found to be helpful to the player. Another important result here is that no participants mentioned that companions performed actions in an unexpected order, in particular that companions would build multiple towers without repairing them. This means that our implementation of n-grams was successful in having companions follow popular sequences of actions. We have included a table of sample responses and graph of responses by coded response in Table 8.2 and Figure 8.20.

Our next section asked if companions worked well together. While this is not a particular feature of our game, we wanted to gather their perceptions of our game with multiple companions. Particularly we wanted to see if they thought the companions just all worked on the same action. Their responses are recorded in Figure 8.21. Similar to the previous section, some participants were unable to make it to this point in the game which explains the high number of neutral responses. However, of those players that did, several noted that they worked together on tasks such as completing buildings. We can see that these participants felt that they did in fact

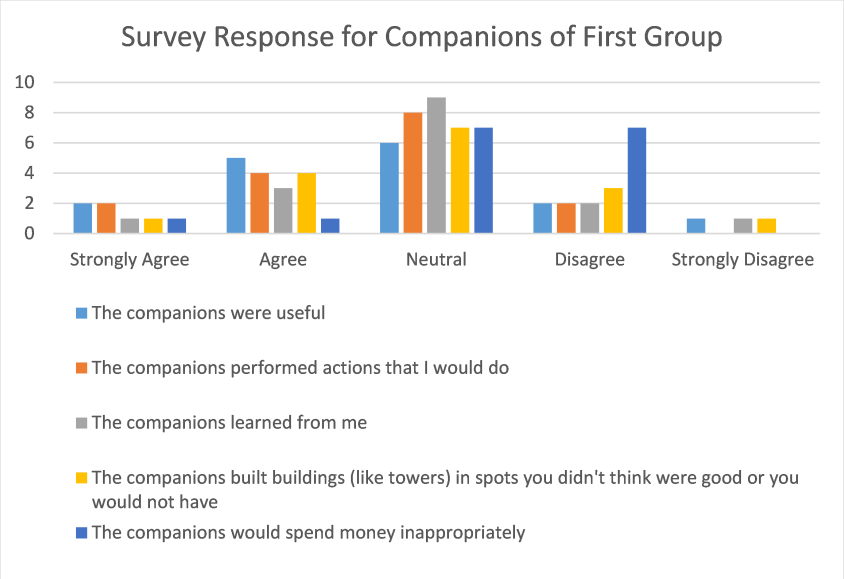


Figure 8.18: Survey Responses about Companions in Lord of Towers

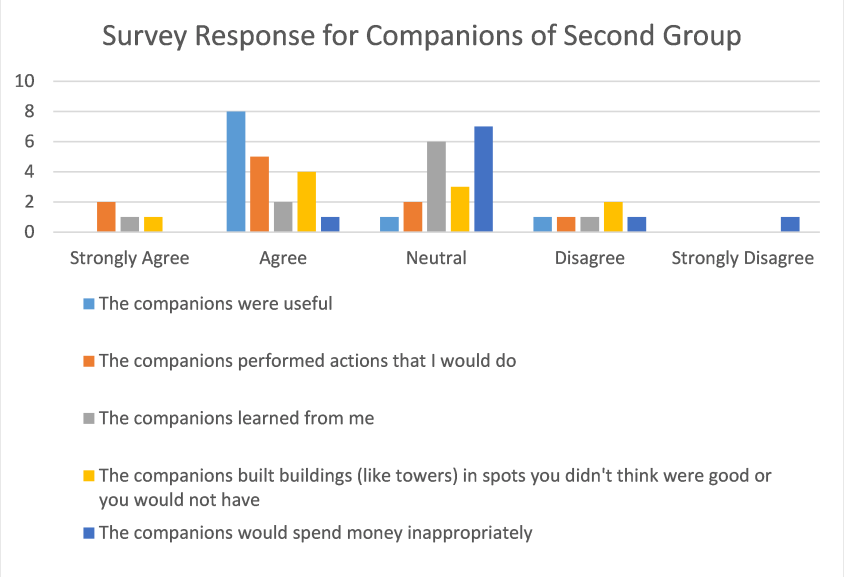


Figure 8.19: Survey Responses about Companions in Lord of Towers

cooperate well. One notable comment stated that one companion mined resources while the others built defenses at the beginning of the game which would accurately mimic what several human players would do when they were low on resources.

In regards to the feedback system, our questions were aimed at determining the

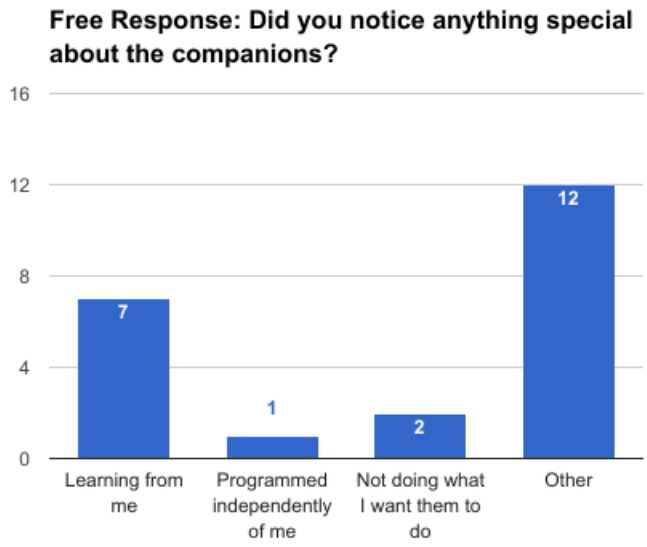


Figure 8.20: Free Responses to Companion Comments

Table 8.2: Sample Responses for Comments about Companions

Code Category	Sample response
Learning from me	They nearly emulated my behavior, but they would not always plug holes in the defenses.
Programmed independently of me	no I didn't even really notice them do anything in particular
Not doing what I want them to do	they need to prioritize repairing. If they're going to upgrade towers, have them upgrade damage.
Other	I didn't live long enough to get a good feeling for them

usefulness of our new feedback component. In each question, a majority of the responses were neutral. We believe that the reason for this is that they felt the amount of feedback was good enough as they did not feel the need to provide more information or less. In addition, five participants agreed that they liked the amount of feedback they could give as well as finding the "Not Now" option helpful. The "Never" option was primarily neutral because participants probably did not use the "Delete Building"

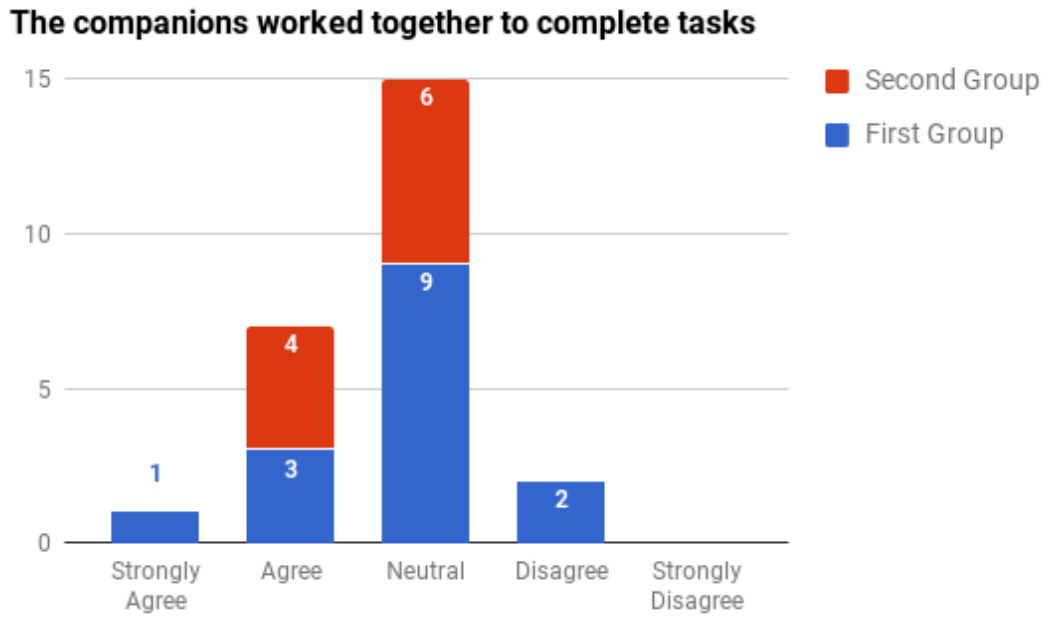


Figure 8.21: Survey Response for Companions working Together

option and permanently banning common actions such as building a tower would be a poor decision. In addition, ten participants used the "Accept all Actions" button. This may reflect that they felt the companions were useful and that they did not feel the need to provide more feedback in these cases. Only one student used the "Wait a Bit" option which was most likely to save resources for an upgrade.

The last section of our survey asked about possible applications of these companions as well as if they felt this tool could be useful to a game developer. For applications of these companions, most of the students mentioned strategy games and RPGs while four stated that they apply to any type of game. Interestingly, two directly mentioned tower defense games. The second part of this section asked if they thought our framework would be useful to a developer. Of twenty-six, nineteen stated they felt they could be useful to a developer. When asked to elaborate on this response, several noted that these companions add interesting mechanics in terms of teaching an AI and most felt that learning companions were applicable to games and

therefore could be useful to a game developer.

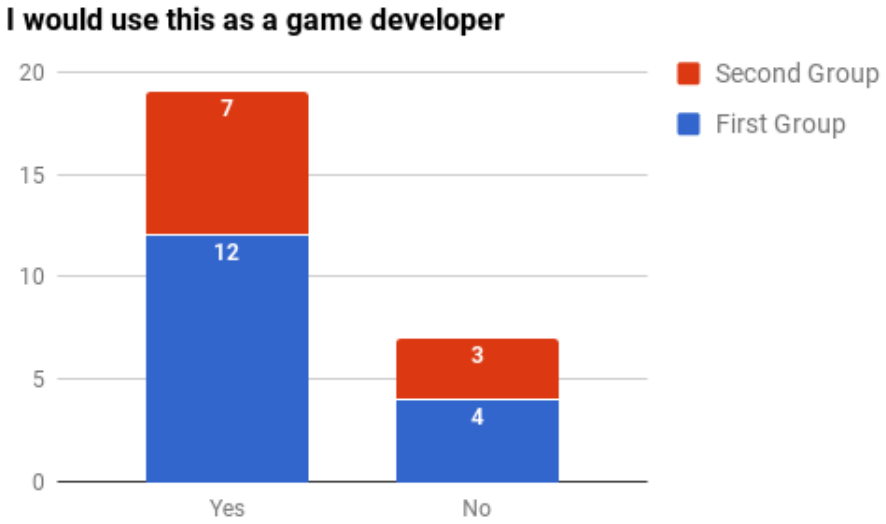


Figure 8.22: Survey Response for Using the Framework as a Developer

Overall, the responses provided by our participants were mostly positive in regards to the companions in Lord of Towers as many found them to be useful and helpful. Several also clearly stated that the companions were learning from them, even before being prompted. In addition, a majority enjoyed playing the game with many noting the diversity of elements in the game. This reflects positively on our updates to Lord of Towers. We also received negative responses as well with many noting that the game was difficult and that the strategy felt rather linear in that they only felt the need to build a maze of towers. Our feedback system did not garner a lot of attention from the player in part because of their focus on gameplay but also means that they did not find anything lacking in these regards. No participants also noted behavior of companions pertaining to awkward sequencing demonstrating the success of our n-gram implementation. Lastly, a significant majority felt that these mimicking companions not only apply to a variety of games but also that these tools would be useful to a game developer.

Chapter 9

CONCLUSION

We present enhancements to the original MimicA framework designed to improve the companion's decision making qualities as well as the framework's ease of integration. To do this, we introduce a random forest classifier, feature selection, n-gram analysis, location determination, enhanced feedback system, as well as a large refactor of the original code. We used both automated testing and a user study to examine the effects of our changes and found that they were, on the whole, an improvement.

9.1 Challenges and Limitations

In this section, we discuss various challenges we encountered during development as well as some of the limitations of choosing to use mimicking behavior for companions.

9.1.1 Cooperation among Companions

Ideally, companions cooperate to perform different tasks that need to be completed. In Lord of Towers, this can be seen in a variety of different forms from working together to build a tower to having one create buildings while another mines more resources. Initially, we had a problem in the game where multiple companions created at the same time would try to perform the same action all the time meaning they would follow each other around. This is not always useful, particularly in the case of one character actions such as mining in Lord of Towers. We considered different approaches to this such as keeping a list of current actions performed by the companions. However, we ultimately decided that this task should be left to the game developer to implement for several reasons. Firstly, the framework has no way of

knowing what actions the companions are performing. It returns a queue of events which could be used to keep track of what event was last removed but this action is not necessarily the one chosen by the game. Additionally, implementation of such a system would require defining an additional parameter in events to determine which events could be performed by multiple companions. Thus, this task is better left to developers to create, for example, as a list of events (and their locations) which can be checked before performing an action. Despite our problem here, we note that players felt companions worked well together in our user study.

9.1.2 Training overhead

Another problem was overhead in training companions. Since we added several new components to the framework, we would inevitably be increasing the computation required during instantiation. We used several different factors to try to combat this. One was to use static global data for information that could be shared between companions to avoid having to redo work. This can be seen in the approaches to n-grams and dynamic region system. We also used a filter based feature selection technique over more computationally expensive ones such as principal component analysis and wrapper methods. This is primarily a result of our focus on a single session experience where the game does not necessarily need a large number of samples to train which can be hard to gather. Since we start with no initial information, we cannot perform preprocessing on the data and as a result need to rely on quick, efficient methods that may be more inaccurate but still generate good results. We note that none of the participants in our study complained about any in game pauses when companions entered the game meaning we were able to keep this overhead to a minimum.

9.1.3 Limitations of Mimicking Behavior

Lastly, we discuss the limitations of mimicking behavior primarily because this is not always desired and replicating a player's exact actions may not be useful in some games. Companions are expected to perform actions that work toward the goal of the game which can sometimes require them to take a strategy differing from the player. An example of this is supporting the melee character with ranged supportive healing spells instead of using melee combat as well. Often in games, players would rather their companions perform actions that are in line with their strategy but not always the same exact action. In our framework, one way to do this would be to log general classes of actions instead of specific actions. For example, in a RPG game, log an attack event instead of a more specific sword slash event. Then when the framework returns an attack event, the companion could determine what type of attack best fits the situation. However, this requires more work on the developer's side in terms of deciding on actions to take.

Additionally, mimicking companions do not try actions the player has yet to perform. While this ensures the companion maintains the same strategy as the player, it also prevents the player from potentially exploring different approaches and being able to expand and build on their strategies.

9.2 Summary of Contribution

In this paper, we have introduced numerous improvements to the MimicA framework aimed at increasing the usefulness of the companions as well as ensuring the framework is flexible and easy to use. In particular, the new region system is able to divide the map for location determination dynamically and avoids static sectors that must be defined beforehand. Our refactor of the original MimicA code has also made it much

easier to use, as evidenced by a clear layout of interfaces and objects to interface with and a distinct divide between the MimicA framework and Lord of Towers game.

To demonstrate the flexibility of the framework, we use two different games: Lord of Towers and Lord of Caves. While we updated Lord of Towers both visually and with new mechanics, we also introduced a new game. Though they are similar in terms of style, their differences in overall goals and gameplay display the different applications of mimicking companions.

Our automated tests showed the strengths of the random forest classifier we introduced as well as the applications of feature selection. The random forest classifier performed very well in of our seven configurations with it typically scoring near the top of the classifiers. Its performance varied much less than the other classifiers demonstrating that it is consistently strong performer. In addition, feature selection was found to be useful in some cases, especially when used with the random forest classifier where it not only reduced decision making time but also increased the overall performance of the decisions made. However, it was found to be less effective with the other classifiers and requires testing to determine what level of feature selection best applies.

In our user study, we also had positive results. The user study displayed that our implementation of n-grams were successful in ensuring a proper order of actions. In addition, the user study also demonstrated the usefulness of our feedback system as participants did not feel they needed to provide more feedback. Overall, the user study found that we were been able to maintain, if not improve, the usefulness of companions while also removing some of the issues present in the original framework and creating an easy to use framework for companion AI.

Chapter 10

FUTURE WORK

This section examines future work that could increase companions' use cases and usefulness to game developers.

One improvement would be to remove the framework from Unity and place it in a library, perhaps in C# or other languages. This would make it usable to a wider number of game developers while still allowing Unity developers to use it. In addition, this allows it to make use of other benefits such as being able to perform some of the work on another thread which is currently impossible in Unity.

Introducing planning to companions is another area of future work. This requires having companions create a goal and then construct a sequence of events to accomplish that goal. A challenge to this is updating the sequence as it is being performed to react to the game state. Using a plan could increase the companions' usefulness by giving them improved action determination.

Another improvement that can be made to the location determination in the game is to improve integration for 3D games. Currently, regions contain 2D coordinates which can still apply to 3D games if the game if the XZ plane is used. However, this would not work for a level with multiple floors.

In addition, it would be interesting to try expanding the framework to apply to enemy characters as seen in Forza 5 [33] and discussed in section 3.6. This not only expands the possible use cases of the framework but also could make for an interesting opponent which has dynamic behavior and difficulty as it learns from the player.

Another topic for future work would be to investigate different training of random forests in terms of dynamically deciding how many decision trees to train and what

percentage of the data to use. This would allow the system to adapt to different data sets and be able to determine a number and size for the classifier relative to the data available. This would also reduce the workload on the developer. To be noted however, this number should not be increased too much so as to avoid a high training overhead.

Lastly, it would be very useful to add some method of logging gameplay data to a file so it can be later reviewed. Specifically, logging data about the actions that were performed in the game would allow us to draw conclusions about what resulted in positive experiences. It would have been very interesting to investigate companions' actions in our user study to determine why our participants answered the way they did and perhaps find common trends.

BIBLIOGRAPHY

[1] Forza 5 vs. forza 4. https://forums.forzamotorsport.net/turn10_postst1510_Forza-5-vs--Forza-4.aspx. Accessed: May 9th, 2017.

[2] Forza ai (drivatars) stop putting it in game. https://forums.forzamotorsport.net/turn10_postst25458_Forza-AI-STOP-Putting-It-In-Game.aspx. Accessed: May 9th, 2017.

[3] ally AI in gears of war. http://www.gamespot.com/forums/xbox-association-1000003/ally-ai-in-gears-of-war-25632517/. Accessed: April 26th, 2017.

[4] T. Angevine. Mimica: A General Framework for Self-Learning Companion AI Behavior. Master's thesis, California Polytechnic State University, San Luis Obispo, CA, USA, 2016.

[5] T. Angevine and F. Khosmood. Mimica: A framework for self-learning companion ai behavior. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[6] S. Bakkes, P. Spronck, and J. V. D. Herik. Rapid adaptation of video game ai, 2008.

[7] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[8] M. Buro and T. M. Furtak. Rts games and real-time ai research. In *In Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS*, pages 51–58, 2004.

[9] M. Farokhmanesh. How Naughty Dog created a partner, not a burden, with Ellie in The Last of Us. `https://www.polygon.com/2014/3/22/5531146/the-last-of-us-ellie-was-designed-to`. Accessed: April 26th, 2017.

[10] M. W. Floyd and B. Esfandiari. A case-based reasoning framework for developing agents using learning by observation. In *ICTAI*, pages 531–538. IEEE Computer Society, 2011.

[11] J. A. Glasser and L.-K. Soh. AI in Computer Games: From the Players Goal to AIs Role. Technical report, University of Nebraska, Nebraska Lincoln, USA, 2004.

[12] T. Jansen. Player adaptive cooperative artificial intelligence for rts games, 2007.

[13] D. Jurafsky and J. H. Martin. Language modeling with ngrams. In *Speech and Language Processing (2Nd Edition)*, chapter 4. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2016.

[14] L. Ladha and T. Deepa. Feature Selection Methods and Algorithms. *International Journal on Computer Science and Engineering (IJCSE)*, Vol 3. Issue 5:1787, 2011.

[15] J. E. Laird. Using a computer game to develop advanced ai. *Computer*, 34(7):70–75, July 2001.

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

[17] M. Mateas. Artificial intelligence today. In M. J. Wooldridge and M. Veloso, editors, *Artificial intelligence today*, chapter An Oz-centric Review of

Interactive Drama and Believable Agents, pages 297–328. Springer-Verlag, Berlin, Heidelberg, 1999.

[18] 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.

[19] M. Mehta, S. Ontañón, T. Amundsen, and A. Ram. Authoring behaviors for games using learning from demonstration. In *in Proceedings of the Workshop on Case-Based Reasoning for Computer Games, 8th International Conference on Case-Based Reasoning (ICCBR 2009*, pages 107–116. AAAI Press, 2009.

[20] C. L. Moriarty and A. J. Gonzalez. Learning human behavior from observation for gaming applications. In *FLAIRS Conference*, 2009.

[21] H. Muñoz-Avila, C. Bauckhage, M. Bida, C. B. Congdon, and G. Kendall. Learning and Game AI. In S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 33–43. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[22] S. Ontañón, K. Bonnette, P. Mahindrakar, M. A. Gmez-martn, K. Long, J. Radhakrishnan, R. Shah, and A. Ram. Learning from human demonstrations for real-time case-based planning, 2011.

[23] A. Ram, S. Ontañón, and M. Mehta. Articial intelligence for adaptive computer games. In *Proceedings of the Florida Articial Intelligence Research Society Conference*, pages 22–29, 2007.

[24] A. Reed. Gamings worst companion AI.

http://www.gamesradar.com/gamings-worst-ai-companions/. Accessed: April 26th, 2017.

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

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

[27] EA Sports and EA Tiburon. Madden NFL. Various, 1988.

[28] Ensemble Studios. Age of Empires 2. Microsoft Windows, Mac OS, Playstation 2, 1999.

[29] Epic Games. Gears of War. [Xbox One, Xbox 360, Microsoft Windows], 2006.

[30] Naughty Dog. Last of Us. [PlayStation 3, PlayStation 4], 2013.

[31] Sega and 2K Sports. NBA 2K series. Various, 1999.

[32] Trendy Entertainment. Dungeon Defenders. Microsoft Windows, Xbox Live Arcade, PlayStation Network, iOS, Mac OS X, Linux, Android, 2010.

[33] Turn 10 Studios. Forza Motorsport 5. Xbox One, 2013.

[34] Unity Technologies. Unity game engine. https://unity3d.com/, 2017.

[35] M. Reynolds. Forza Motorsport 5: How the Xbox One racer's Drivatar system works. http://www.digitalspy.com/gaming/xbox-one/news/a530468/forza-motorsport-5-how-the-xbox-one-racers-drivatar-system-works/. Accessed: May 10th, 2017.

[36] G. Robertson and I. Watson. A review of real-time strategy game ai. *AI Magazine*, 35(4):75–104, 2014.

[37] D. I. Thomas and L. Vlacic. Teammate: computer game environment for collaborative and social interaction. In *Advanced Robotics and its Social Impacts, 2005. IEEE Workshop on*, pages 60–65. IEEE, 2005.

[38] J. Tremblay. Improving Behaviour and Decision Making for Companions in Modern Digital Games. In *Proceeding of the 2013 Doctoral Consoritum at the Artificial Intelligence and Interactive Digital Entertainment*, AIIDE, 2013.

[39] J. Tremblay and C. Verbrugge. Adaptive Companions in FPS Games. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, FDG 2013, pages 229–236, 2013.

[40] H. Warpefelt. *The Non-Player Character Exploring the believability of NPC presentation and behavior*. PhD thesis, Department of Computer and Systems Sciences, Stockholm University, 2016.

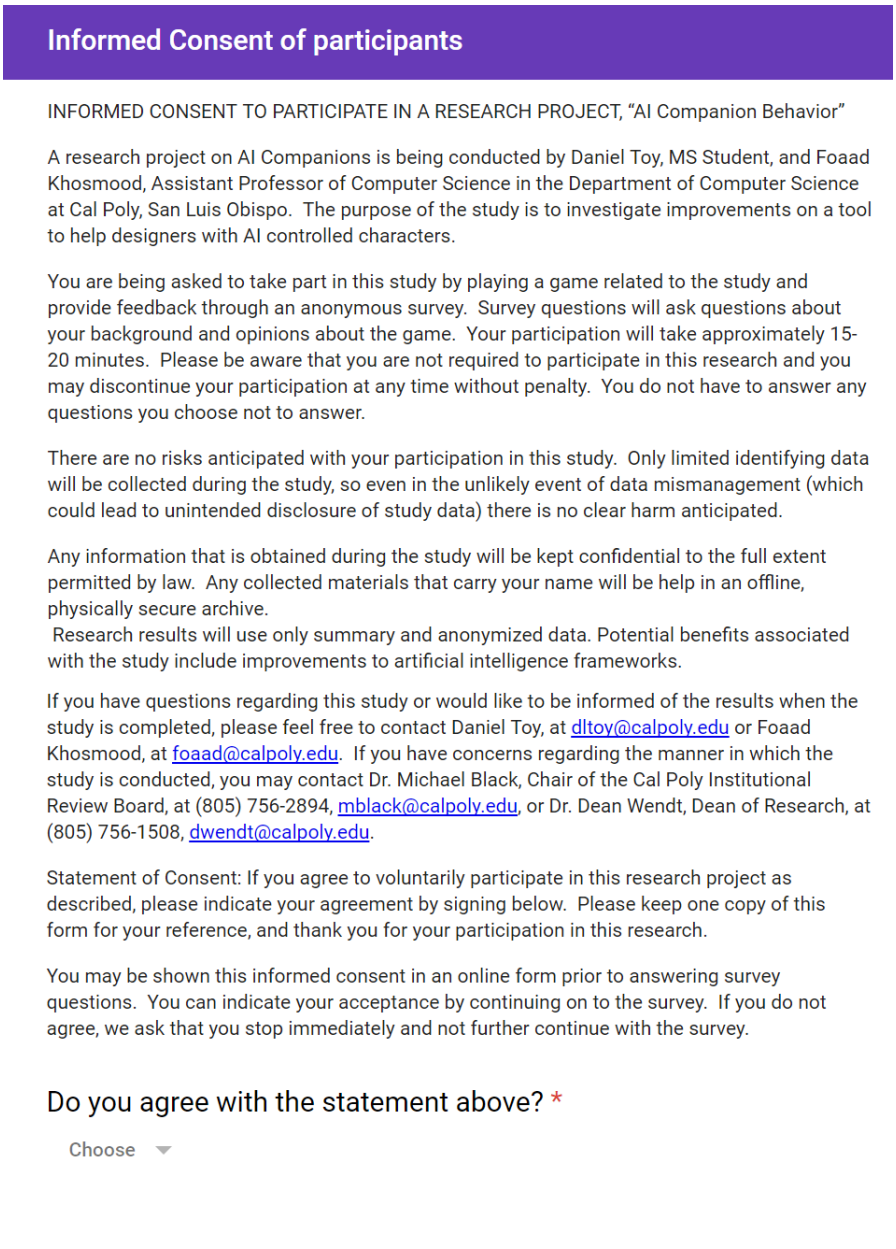
[41] S. Yildirim and S. B. Stene. A survey on the need and use of ai in game agents. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 124–131, San Diego, CA, USA, 2008. Society for Computer Simulation International.

[42] Z.-H. Zhou. Ensemble learning. *Encyclopedia of biometrics*, pages 411–416, 2015.

APPENDICES

Appendix A

USER STUDY SURVEY



Informed Consent of participants

INFORMED CONSENT TO PARTICIPATE IN A RESEARCH PROJECT, "AI Companion Behavior"

A research project on AI Companions is being conducted by Daniel Toy, MS Student, and Foaad Khosmood, Assistant Professor of Computer Science in the Department of Computer Science at Cal Poly, San Luis Obispo. The purpose of the study is to investigate improvements on a tool to help designers with AI controlled characters.

You are being asked to take part in this study by playing a game related to the study and provide feedback through an anonymous survey. Survey questions will ask questions about your background and opinions about the game. Your participation will take approximately 15-20 minutes. Please be aware that you are not required to participate in this research and you may discontinue your participation at any time without penalty. You do not have to answer any questions you choose not to answer.

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 (which could lead to unintended disclosure of study data) there is no clear harm anticipated.

Any information that is obtained during the study will be kept confidential to the full extent permitted by law. Any collected materials that carry your name will be help in an offline, physically secure archive.
Research results will use only summary and anonymized data. Potential benefits associated with the study include improvements to artificial intelligence frameworks.

If you have questions regarding this study or would like to be informed of the results when the study is completed, please feel free to contact Daniel Toy, at dltoy@calpoly.edu or Foaad Khosmood, at foaad@calpoly.edu. If you have concerns regarding the manner in which the study is conducted, you may contact Dr. Michael Black, Chair of the Cal Poly Institutional Review Board, at (805) 756-2894, mblack@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 signing below. Please keep one copy of this form for your reference, and thank you for your participation in this research.

You may be shown this informed consent 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 statement above? *

Choose

Figure A.1: Page 1 from the Survey

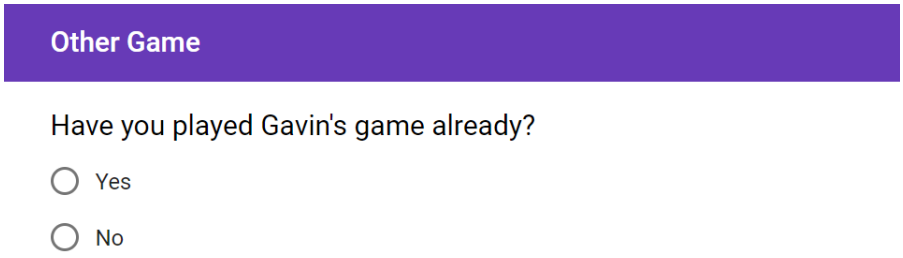


Figure A.2: Page 2 from the Survey

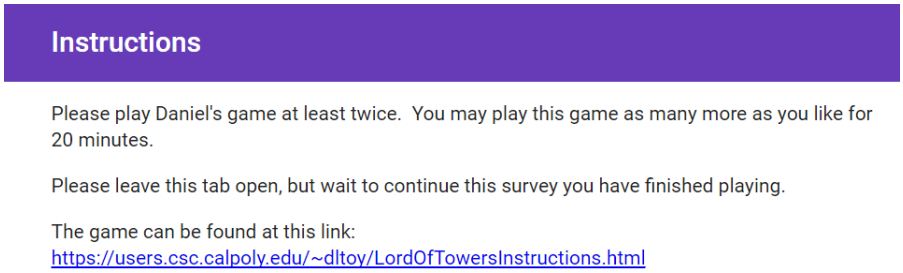


Figure A.3: Page 3 from the Survey

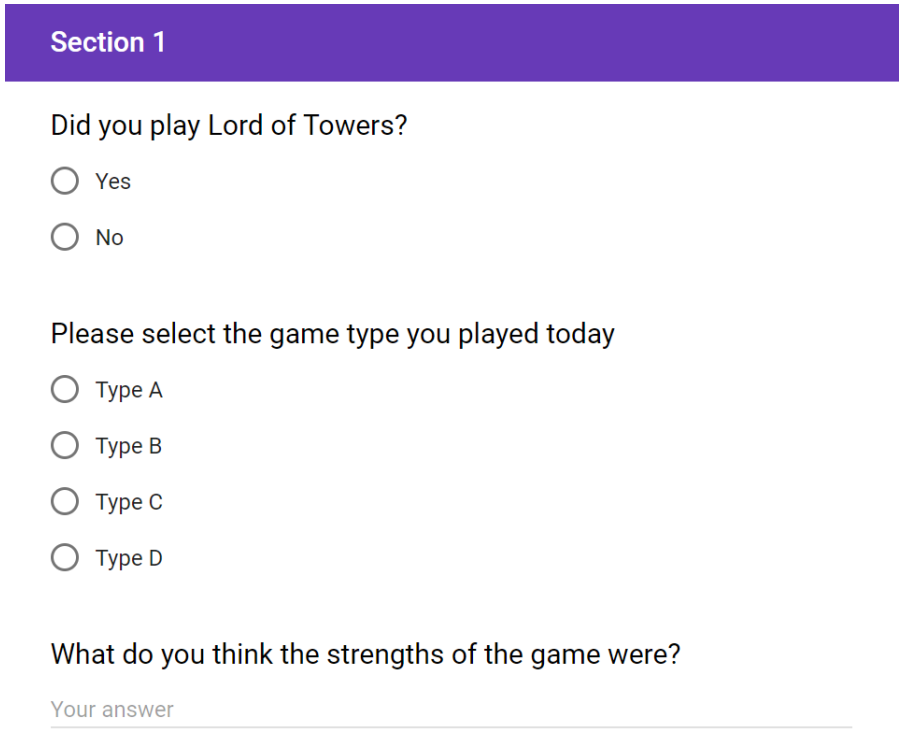


Figure A.4: Page 4 from the Survey

What do you think the weaknesses of the game were?

Your answer

How many different tower defense games have you played?

- None
- 1-2
- 3-5
- 6+

If you have played other video games that have a companion character, please list them here.

Your answer

Please rate your level of agreement with the following statments

	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
I enjoyed playing Lord of Towers					

Please elaborate on your answer

Your answer

Figure A.5: Page 4, part 2 from the Survey

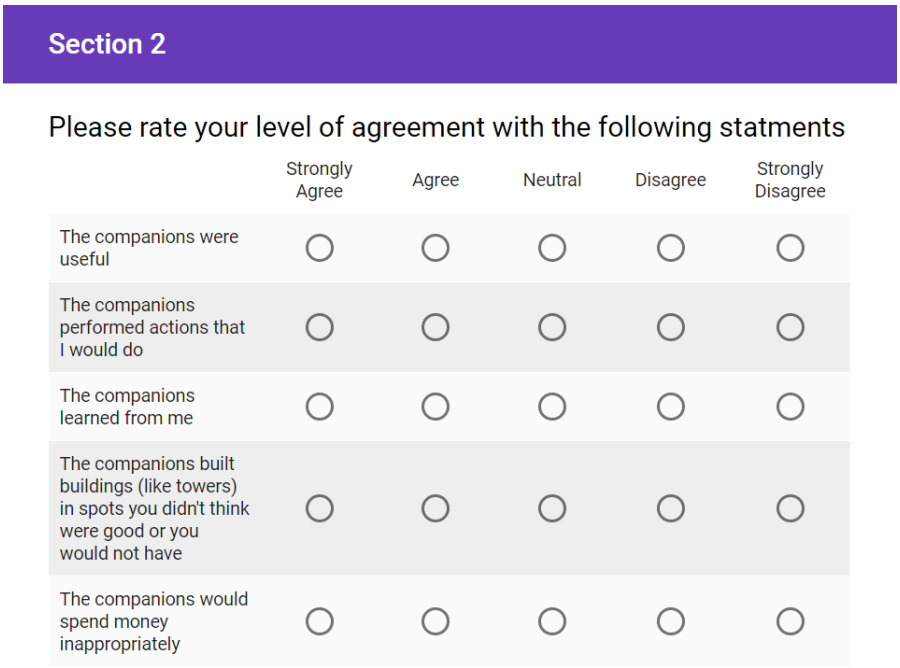


Figure A.6: Page 5 from the Survey

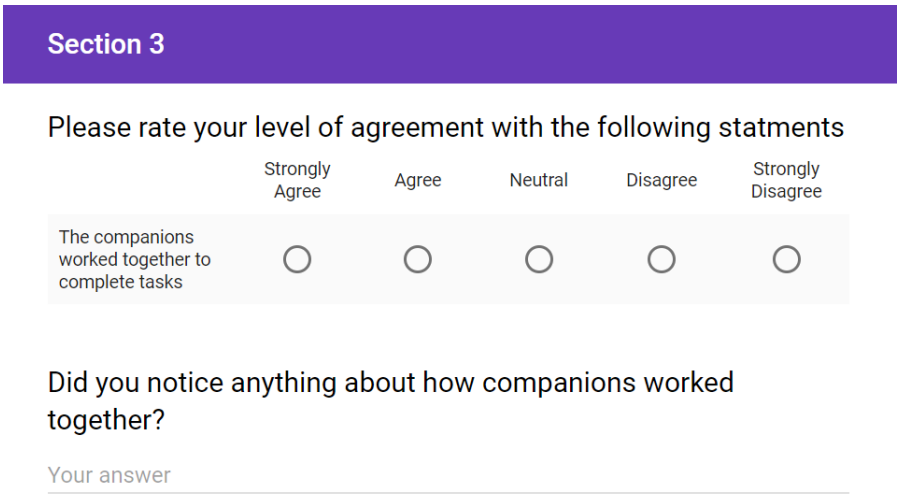


Figure A.7: Page 6 from the Survey

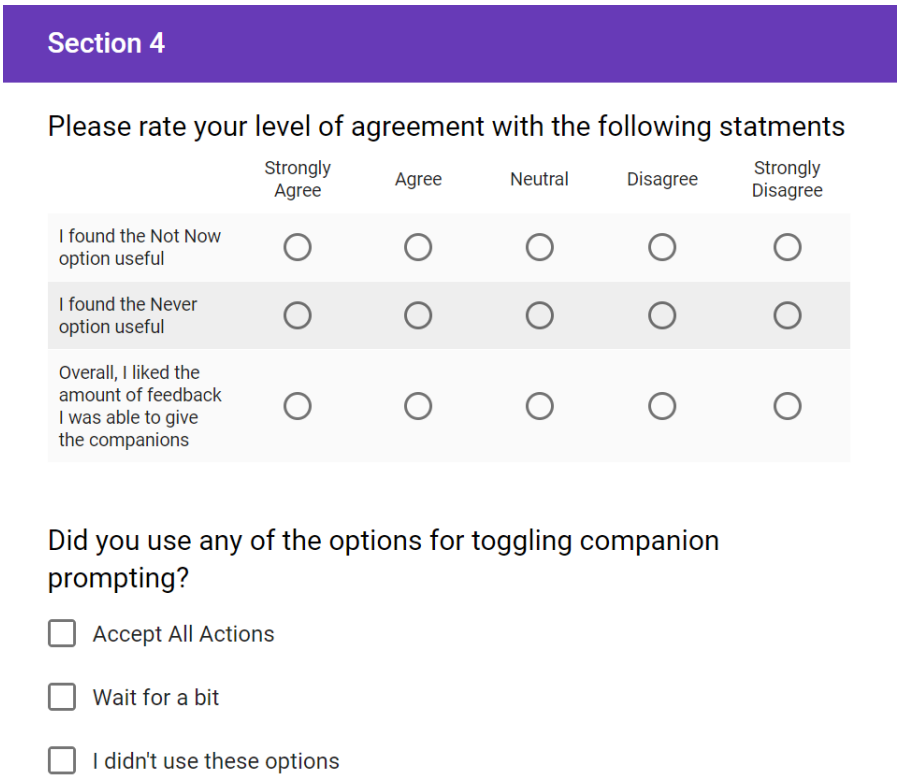


Figure A.8: Page 7 from the Survey

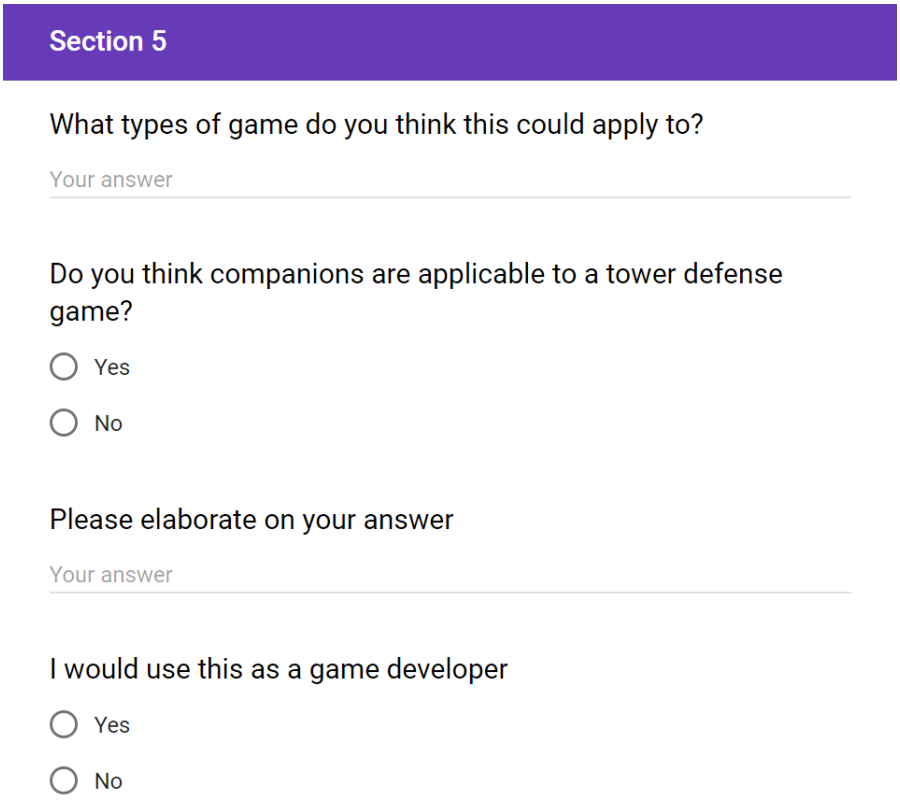


Figure A.9: Page 8 from the Survey

Appendix B

HOW TO USE THE FRAMEWORK

This guide assumes that the developer is building a game within the Unity game engine.

1. Define the coordinates of the world
   (a) This is done by calling the setWorldCoordinates(Vector2 botLeft, Vector2 topRight) method in FrameworkGameData.
   (b) If the world is defined in a tile system, it is very useful a conversion.
2. Create a game state class
   (a) This class must extend FrameworkGameStateVector.
   (b) Define all variables that are part of the game state as public variables (Reflection is used to gather only public variables so private variables can still be used for other tasks).
3. Define events that can take place in the game
   (a) These events must extend either FrameworkEvent or FrameworkLocationEvent. Use FrameworkLocationEvent when location is important to the event (e.g. building a tower)
   (b) If FrameworkLocationEvent is used, add logic to determine where to perform an action. Each FrameworkLocationEvent contains a FrameworkRegion to specify a range in world coordinates.
   (c) Make sure to define the getName() and equals() methods to get the string name of an event and to compare 2 events against each other.

4. Add game states to FrameworkGameData

    (a) Create a script for controlling the player.

    (b) Whenever the player performs an action, call FrameworkGameData's addGameStateVector(FrameworkGameStateVector vector, FrameworkEvent fEvent, Vector2 location) method. This requires the current game state, the event performed, and the world coordinates where it was performed.

5. Add the companion's "brain"

    (a) Create a script in the Companion prefab or Game Object to determine the companion's actions and add a FrameworkCcompanionLogic object as a class variable.

    (b) Instantiate the brain. This requires several parameters: List<string> possibleEvents, List<string>forbiddenEvents, FrameworkEvent defaultEvent, List<string>locFeatures. possibleEvents are all the events it can perform, forbiddenEvents are a list of events that should never be performed by default, defaultEvent is a event to perform if none can be done, and locFeatures are the names of the game state variables to use to determine locations of actions

    (c) Optional parameters are discussed in step 8.

6. Getting a decision from the framework

    (a) Call getDecision(FrameworkGameStateVector vector, List<FrameworkEvent >lastEvents, Vector2 lastEventLoc, bool doRetrain = false) which requires the current gamestate vector, last events performed by the companion, the location of the last event, and optionally whether to retrain the companion

    (b) This returns a FrameworkEventsToDo which contains a queue of FrameworkEvents that can be dequeued by calling getNextEvent().

7. Banning Actions

   (a) To temporarily ban an action for a particular companion call addCurrentlyForbiddenEvent(string eventName) in that companion's FrameworkCompanionGameLogic.

   (b) Similarily, to permanently ban an action call addPermanentlyForbiddenEvent(string eventName) in that companion's FrameworkCompanionGameLogic. Note this causes the companion to retrain.

   (c) Both of these lists can be cleared using clearCurrentlyForbiddenEvents() and clearPermanentlyForbiddenEvents().

8. Additional options for tuning companion

   (a) To choose a different classifier, set the optional parameter 'classifier' when instantiating the FrameworkCompanionGameLogic. The type is an enum called FrameworkClassifier.

   (b) To use n-grams, set the optional parameter 'ngramLengths' when instantiating FrameworkCompanionGameLogic. The type is a List of ints to specify the lengths of the sequences.

   (c) To specify a different percentage of features to cull, use the optional parameter 'fsPercentage' when instantiating the FrameworkCompanionGameLogic. It is a float and defaults 0.10f (10%).

9. Saving and Loading FrameworkGameData

   (a) To save, call FrameworkGameData's saveDataHistory(string keyFileName, Type[] fgsvTypes, string valueFileName, Type[] eventList). You must specify 2 file names for to save to. fgsvTypes is the type of the events and gamestate vector while eventList is the just the types of each event. (e.g. in c# typeof(GameStateVector), typeof(Event), typeof(AoeUpgrade)).

(b) Loading is similar, and is called by loadNewDataHistory(string keyFileName, Type[] fgsvTypes, string valueFileName, Type[] eventList, bool append = false). It adds the optional parameter append which can override the current dictionary or just append. Make sure the xml files exist before calling this method.

Appendix C

ART CREDITS FOR LORD OF TOWERS AND LORD OF CAVES

C.1 Sprites

- Archer: http://www.sprites-unlimited.com/game/?code=HMM2
- Bomb: https://www.spriters-resource.com/genesis_32x_scd/kidchameleon/sheet/51148/
- Golem: http://opengameart.org/content/golem-animations
- Knight (Companion): http://www.gameart2d.com/the-knight-free-sprites.html
- Mage: http://spritedatabase.net/file/14024
- Dwarf (Player): http://www.sprites-unlimited.com/game/?code=HMM2
- Skeleton: http://opengameart.org/content/skeleton-animations
- Default Tower: taken from Mimica's Lord of Towers
- AOE Tower: http://opengameart.org/content/brick-tower-large-sprite
- Damage Tower: http://ayene-chan.deviantart.com/art/RPG-Maker-VX-Tower-382044826
- Slow Tower: http://cityville.wikia.com/wiki/File:Wizard_Tower-SW.png
- Range Tower: http://opengameart.org/content/cannon-tower
- Vampire: http://opengameart.org/content/vampire-animations
- Zombie: http://opengameart.org/content/zombie-animations
- Ball: taken from Mimica's Lord of Towers
- Castle: taken from Mimica's Lord of Towers

- Cave Border: https://www.pinterest.com/pin/439312138627253428/
- Cave Wall: http://allacrost.org/staff/user/bigpapan0z/ss_browncave.png
- Blue Tile: http://www.iconsdb.com/caribbean-blue-icons/square-icon.html
- Gold Mine: http://ageofempiresonline.wikia.com/wiki/File:GoldMine.png
- Green Grass: https://www.pinterest.com/lcvick/atmospheric-textures/
- Health Center: https://www.reddit.com/r/pokemon/comments/1px3bb/the_advancement_of_the_pokemon_center/
- House: http://www.deviantart.com/morelikethis/297406876
- Log: taken from Mimica's Lord of Towers
- Stone Quarry: http://opengameart.org/content/2d-platform-ground-stone-tiles
- Wall: taken from Mimica's Lord of Towers
- Tree: Ms. Jensen Welton, Artist Extrordinaire

C.2 Sounds

- Ball bouncing: https://www.youtube.com/watch?v=qCm-hjMWnFg
- Basketball bounce: https://www.youtube.com/watch?v=aYvjZSYmkT8
- Bats: https://www.youtube.com/watch?v=nM0InF4UNqU
- Bow String: https://www.youtube.com/watch?v=KW8cSQ3nUj4
- Collapsing Bones: https://www.youtube.com/watch?v=qU_9lnQvLhk
- Death Scream: https://www.youtube.com/watch?v=AGce8M-MZxs

- Evaporating Water: https://www.youtube.com/watch?v=-IqKCSPH-SE&t=123s
- Evil Laugh: https://www.youtube.com/watch?v=ywtjxen2n1A
- Explosion: https://www.youtube.com/watch?v=nRwM7UEQ8Q0
- Fire Sound: https://www.youtube.com/watch?v=vOtzPWx7HXU
- Hammer hit: https://www.youtube.com/watch?v=xcDVBwaI-V0
- Hammer Sound: https://www.youtube.com/watch?v=y5Mw0O0BdwU
- Icicles: https://www.youtube.com/watch?v=Vi6Q86r2UPQ
- Man Scream: https://www.youtube.com/watch?v=H3vSRzkG82U
- Monster Growl: https://www.youtube.com/watch?v=Ii5MaHYlFzw
- Ouch Sound: https://www.youtube.com/watch?v=ZG32UnCzhqE&list=PL6i13PMXG_TezMEGVz6KD_UVINI0FN_sf&index=
- Sizzling Sound: https://www.youtube.com/watch?v=TVyth8uu-w4&t=2s
- Sword Slash: https://www.youtube.com/watch?v=X3liPsg21Cg
- Wand Sounds: https://www.youtube.com/watch?v=FmEiTpMCur8
- Yoga Ball: https://www.youtube.com/watch?v=xeifyIoD6RU
- Zombie Dying: https://www.youtube.com/watch?v=BjirvbYuq7c
- The Dragon Valley by Peter Crowley: https://www.youtube.com/watch?v=HSsO_9DbtOM
- The Kingdom Above the Sky by Peter Crowley: https://www.youtube.com/watch?v=dDwYzJBtv9w
- Peter Crowley's Youtube Channel: https://www.youtube.com/user/PeterCrowley83