A N-Grams implementation in C# in a Unity Game

BACHELORARBEIT 2

StudentIn Michael Webersdorfer, 0910601034
BetreuerIn DI Robert Praxmarer

Salzburg, 07. Mai 2012

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst, und keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe. Weiters versichere ich hiermit, dass ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

Datum Unterschrift

Kurzfassung

Vor- und Zuname:	Michael WEBERSDORFER
Institution:	FH Salzburg
Studiengang:	Bachelor MultiMediaTechnology
Titel der Bachelorarbeit:	A N-Grams implementation in C# in a Unity Game
Begutachter:	DI Robert Praxmarer

N-Gramme sind eine Technik, die verwendet wird, um zukünftige Elemente einer Sequenz vorherzusagen. Sie werden am meisten im Gebiet der Linguistik verwendet, um bessere Ergebnisse in Anwendungen wie Text-to-Speech-oder Auto-Korrektur zu liefern.
In Videospielen werden N-Gramme verwendet, um einer KI (Künstliche Intelligenz) zu ermöglichen Spielerverhalten vorherzusagen. N-Gramme sind in der Lage zukünftige Aktionen vorherzusagen, indem sie die vergangenen Aktionen eines Spielers oder einer Spielerin analysiert und speichert, wie oft er oder sie sich für bestimmte Kombinationen von Aktionen entschieden hat.
N-Gramme ermöglichen es der KI, sich an den Spieler oder die Spielerin anzupassen. Auf diese Weise kann die KI, je nach Einsatzgebiet, entweder eine herausfordernde Gegnerin oder eine unterstützende Verbündete sein. Der Unterhaltungswert des Spiels kann durch die Verwendung von N-Grammen gesteigert werden, weil sie den Wiederspielwert des Spiels erhöhen und ein stärker individualisiertes Erlebnis für den Spieler oder die Spielerin bieten können.
Es gibt zwei Werke von Millington und Laramée, die sich mit der Verwendung von N-Grammen in Videospielen befassen. Dennoch gibt es wenig Informationen darüber, wie gut N-Gramme funktionieren, wenn sie tatsächlich in einem Spiel zum Einsatz kommen. Diese Bachelorarbeit hat sich zum Ziel gesetzt solche Informationen zu erarbeiten.
Um dieses Ziel zu erreichen wird das Spiel *Blocky* geschaffen und als eine Umgebung für die N-Gramme Implementierung verwendet. Um Informationen darüber zu erlangen, wie gut diese Implementierung arbeitet, werden mehrere Tests durchgeführt und deren Ergebnisse in dieser Bachelorarbeit besprochen.

Schlagwörter: Künstliche Intelligenz, Videospiel, Vorhersage von SpielerInnenverhalten, N-Grams

Abstract

N-Grams are a technique used to predict future elements of sequences. They are mostly know for their usage in linguistics where they are used to provide better results in applications like text-to-speech or auto correction.
In video games N-Grams are used to create an AI (artificial intelligence) which can predict player behavior. N-Grams are able to predict future actions of a player by analyzing his past moves and storing how often a player chose which combination of moves.
N-Grams enable the AI to adapt itself to the player. By doing this the AI can either be a challenging opponent or a supportive ally depending on the design of the AI agent. In any way the entertainment value of the game can be increased by using N-Grams because they increase the replayability of the game and can provide a more individualized experience for the player.
There are two works which deal with using N-Grams in video games by Millington and Laramée. But there is little information about how well N-Grams work when actually implemented into a game. This thesis set it's goal to provide such information.
To reach this goal the game *Blocky* was created and used as an environment to implement the N-Grams. To provide information about how well the N-Grams implementation works, several tests were conducted and their results are discussed in the thesis.

Keywords: *artificial intelligence, N-Grams, video game, prediction, player prediction, sequence analysis, linguistics*

Contents

1 Introduction

1.1 Motivation

The literature I studied for my thesis about player prediction suggests N-Grams are very good at predicting future moves in beat-em-up and other fighting games (see Millington 2006, 591). But there is hardly any information about how often or in which specific games N-Grams were implemented. It's therefore hard to comprehend how well N-Grams really work and how relevant they are for AI (Artificial Intelligence) in games. N-Grams are a technique mostly used in the field of linguistics. They are well established in this field and are used for various applications. But the question remains of how useful and efficient they work in a real time simulation like a video game.
In fighting games the player will often stick to the same combination of moves. It might be because they have proofed to be good or there is a bonus for doing moves in that order. But even subtle things like how the moves are mapped to the gamepad can lead to the player favoring a certain combination of moves. So there's a high chance of a player following some sort of pattern.
An AI which doesn't predict and adapt to such a pattern or player behavior in general will always react with the same moves. This results in a boring and repetitive experience for the player because he or she won't be challenged. N-Grams enable the AI to predict repetitive actions of a player. By doing so they also adapt to the player and provide him or her with a more individualized AI opponent. Such an opponent will be more challenging for the player because the player is forced to come up with new combinations of moves to win.
N-Grams can also be used in a lot of other scenarios besides fighting games. Basically N-Grams can be useful in any scenario where a human player has to choose from a set of possible actions. However they are not capable of predicting general player behavior, like the players tendency to play aggressively or defensively.

1.2 Research Question

Millington and Laramée provide pseudo code and code snippets of a N-Gram implementation (Millington 2006, 583) and (Laramée 2002, 597). But they don't provide a complete implementation. They also don't present any information about how successful N-Grams were in a video game environment, although Millington provides some small tests (Millington 2006, 587). There have been remarks in the literature about memory concerns when using N-Grams, especially when the number of possible actions is large (see Millington 2006, 587). Also there is hardly any information about N-Grams being used in a commercial game.
It's therefore elusive how good N-Grams really perform in a state of the art video game. This thesis provides an implementation of a N-Grams predictor in C# in a video game based on the Unity engine. That implementaion together with the conducted tests will deliver answers to the following research questions:

1. How can N-Grams be implemented in a Unity game?
2. How successful are N-Grams at predicting future player actions?
3. Does the memory consumption and time complexity of the algorithm allow its usage in state of the art computer games?

1.3 Structure of the thesis

This thesis first provides some theoretical background on N-Grams in section 2. In the subsection 2.1 the usage of N-Grams research fields beyond video game AI will be discussed. This is relevant because N-Grams have a history in the field of linguistics and are rather exotic in the field of game AI. N-Grams in the setting of video games will be discussed afterwards in subsection 2.2.

Section 3 is about the practical implementation done in conjunction with this thesis. The subsection 3.1 first provides the knowledge about Unity which is needed to understand the rest of the implementation. Subsection 3.2 explains the game design and rules of the game *Blocky*. In the next subsection the implementation of the game *Blocky* will be presented. This game was created as an environment for the N-Grams implementation. It also serves as a test setting. The N-Grams implementation itself is explained in subsection 3.4. The code for both the game *Blocky* and the N-Grams implementation is displayed in the appendix. This was done because the code is several pages long and would have disturbed the flow of reading.
The test set-up and the expected outcome of those tests are described in section 4. This section also explains the reasons why those tests were done in that way.
Section 5 is about the results of the conducted tests. Subsection 5.1 provides a theoretical analysis and presents the results of some performance tests. The AI vs AI test results are revealed and discussed in subsection 5.2 followed by the Human vs AI test results in subsection 5.3.
The final conclusion of this thesis can be found in section 6.

2 N-Grams

2.1 N-Grams in other fields

N-Grams are widely used for linguistics and also have their roots in this field of research.
In linguistics the term N-Grams describes the result of splitting a text into fragments. The fragments get pooled into groups of the size N. Those groups are N-Grams. The fragments can be words, letters, phonemes or similar chunks. This means what a N-Gram contains is not consistent across different systems.
An example of a N-Gram with words as fragments is the following:

- serve as the industry 607
- serve as the info 42
- serve as the informal 102
- serve as the information 838

This example was taken from the Google N-Grams Corpus. The number behind the sentence describes how often this sentence occurs in the corpus (Franz and Brants 2006).
The term N-Gram model describes the technique of modeling sequences by using the statistical properties of N-Grams. With a N-Gram model it's possible to compute the probability of a sequence or the next next member of a sequence. Such statistical models are also called language models (see Jurafsky and Martin 2008, 93).
A N-Gram with the size 1 is called an unigram and often written as 1-Gram. Same for 2-Grams which are called bigrams and 3-Grams which are called trigrams and so forth. The term window size describes how many previous fragments get taken into consideration when predicting a future fragment. The window size is N-1 for a N-Gram.
N-Gram models are used a lot more in the field of linguistics than they are used for game AI. The research of N-Grams in the field of linguistics started to be popular in the 1970s. But the basic idea of N-Grams was proposed by Markov in 1913 (see Jurafsky and Martin 2008, 129). There have been a lot of successful applications of N-Grams in various scenarios since then and N-Gram models are a well established technique nowadays.

N-Grams are most commonly used for statistical natural language processing (NLP). The goal of NLP is to extract information which can be processed by an AI from natural language or to create

natural language output from such information. Example applications in this area are: part-of-speech tagging, natural language generation, word similarity, authorship identification, sentiment extraction and predictive text input systems for cell phones (see Jurafsky and Martin 2008, 94).
In speech recognition N-Grams are used to handle the noisy input from speech. A lot of sounds get distorted or sound too similar to other sounds to be discerned correctly. N-Grams help by calculating what results are most probable.
There has also been successful research about using of N-Grams for handwriting recognition (see Cavnar and Vayda 1992).
N-grams are also one of the core techniques used for machine translating. They can discern which results are probable and which aren't (Mariòo et al. 2006).
Spelling correction can also be achieved by using N-Grams. The sentence "He wants to fine out" is an example of a semantic error. That error would not be detected by natural language parsing because "fine" is valid English word. Analyzing this sentence with a N-Grams model could provide you with the information the probability that this sentence is valid is way smaller than the probability of the sentence "He wants to find out" (Kukich 1992).
Authorship attribution is another application of N-Grams. The goal of the application is to identify an author of an anonymous text, or text whose authorship is in doubt or to detect plagiarism. To achieve this goal first a byte-level N-Gram author profile of an authors writing is created. The profile is essentially a relatively small set of frequent N-Grams. Two important operations are to choose the optimal set of N-Grams to be included in the profile, and to calculate the similarity between two profiles (see Keselj et al. 2003, 256).

Google uses N-Grams for some of the applications mentioned earlier. Through their work they have been able to accumulate massive amounts of data from public web pages. They used this data to produce a training corpus of one trillion words and shared that data with the public in 2006 (Franz and Brants 2006). In 2010 Google also released the *Google Ngram Viewer* which allows the user to view the usage of a word or a phrase over time. This application uses a data corpus of 500 billion words from 5.2 million books which were digitalized by Google. This data was used for research in several papers and articles like "Quantitative Analysis of Culture Using Millions of Digitized Books" (Jon 2010).
Microsoft also made access to their *Web N-gram Corpus* public. This corpus can be updated as deemed necessary by the user community and "the corpus makes available various sections of a web document, specifically, the body, title, and anchor text, as separates models as text contents in these sections are found to possess significantly different statistical properties and therefore are treated as distinct languages from the language modeling point of view" (Wang et al. 2010).
In 2001 the World Wide Web Consortium (W3C) drafted the "Stochastic Language Models (N-Gram) Specification". "This document defines syntax for representing N-Gram (Markovian) stochastic grammars within the W3C Speech Interface Framework" (Brown, Kellner, and Raggett 2001). It introduced the <n-gram> tag and standards of how to describe N-Grams in XML.

In biology N-Grams are used for example to analyze whole-genome protein sequences (Ganapathiraju et al. 2002) and for Systematic Analysis of Coding and Noncoding DNA Sequences (Mantegna et al. 1995).

2.2 N-Grams in video games

In contrast to the terminology in the linguistics field, in the game AI field the term N-Grams describes the whole technique of analyzing and predicting data. The term "N-Gram predictor" describes the part of the process which is responsible for calculating the most likely future action. N-Grams can be classified as an adaptive AI technique. By recognizing, predicting and countering player behavior they adapt to that player. Such adaptive AI techniques can provide the player

with an individualized gaming experience, because the AI will adapt to his or her actions. Since the actions of a human player differ from one player to the other, the resulting AI behavior will be uniquely tailored for that specific player. The N-Grams predictions could be used not only to beat but also to support a human player. Since the variety of play styles displayed by human players is large an AI, which is set up to fit them all, will probably not be as fitting as an adaptive AI in many situations. The replayability of a game is also improved by integrating a N-Grams AI. A nonadaptive AI would always do the same moves given the same input from the human player. In a game which focuses on realism such repetitive behavior can lead to the player losing the illusion of the game being real. Besides that use case a player could also beat an unadaptive enemy by trial and error if he or she just tried hard enough.
But such adaptive AI techniques aren't used very often in video games. The main reason for this is that developers and publishers shun the possibility of unpredictable AI behavior (see Millington 2006, 565-566). It's just saver if every AI behavior can be tested before shipping the game. This usually can't be done with adaptive AI because the number of possible behaviors is too large. However it doesn't have to be an either/or decision. Adaptive AI techniques like N-Grams can be integrated into the game, but be equipped with certain limits. That way the AI is still adaptive, but only to a certain degree. This would grant the AI a more stable behavior as desired by the developers and publishers.

Compared to the research and applications in the linguistics field the information about N-Grams in video games is sparse. There are two works which can be considered as main works of this field. The first is an article titled "Using N-Gram Statistical Models to Predict Player Behavior" by Laramée (Laramée 2002, 596-601). The second is a subchapter which Millington dedicated to N-Grams in the book "Artificial Intelligence for Games" (Millington 2006, 582-591). Besides discussing N-Grams in general they both provide ideas of how they could be implemented, but not a full implementation. The main difference in the two approaches is the way the data is saved and accessed. Laramée uses a N-dimensional array (see Laramée 2002, 598), Millington uses a hashtable (see Millington 2006, 585). Laramées approach will be more efficient if the number of actions and the order of the N-gram are both low. The benefit of Millingtons approach is that it's more flexible concerning the window size and number of actions and that it avoids wasting data on combinations which are never seen (see Millington 2006, 585).
As mentioned earlier N-Grams are very good at analyzing sequences in a given dataset and at predicting the most probable next member of that sequence. In video games the recorded actions of a player is this data. For example a character in a typical RPG (Role Playing Game) could have the abilities normal attack (N), strong attack (S) and heal self (H). If recorded the last 14 used abilities or actions could be:

$$NNS\ NNS\ NNH\ NNS\ NN$$

If a NPC (Non Player Character) would exhibit such a behavior a human player could easily recognize the pattern and adapt to it. N-Grams enable the NPC to do the same. The first step in achieving this is analyzing the players' actions with a N-Gram. If this is done with a trigram on the example data described earlier, the resulting data would be such as seen in table 1.

Trigram sequence	*Times the sequence was seen*
NNS	3
NNH	1

Table 1: Example analysis of recorded player actions

The example data with the human players actions ends with the two actions "NN". Using the data in table 1 the NPC could now predict that the probability of the human player doing the

strong attack ability next is 75% and that the chance of him or her using the heal self ability is only 25%. The NPC could then react according to this prediction by using a defensive ability which nullifies the strong attack. By going through this whole process the NPC adapted to that specific players' behavior. The resulting actions of the NPC depend on the players behavior and hence will change with it. Therefore the NPC will provide an individualized experience for every player. The number of reactions is limited by the number of possible combinations the player can exhibit and the number of reactions the NPC can choose from.
An use case for N-Grams as described above would be a NPC in a RPG or MMORPG (Massivly Multiplayer Online Role Playing Game). In such games the player is often given the task to kill a certain number of enemies of the same kind. This task is quite monotonous for the player because he or she usually found out certain combinations of abilities work best if applied in a certain order from former similar tasks. So the player ends up with repeating the same skills in the same order to kill the same kind of NPC. Now if the NPC would adapt to the player as described above, the player will probably change his or her choice of skills because they get countered. This creates more diverse and challenging fights for the player, which can be more entertaining for the player than monotonous fights.

N-Grams are a technique which require learning. The players' behavior needs to be learned by the N-Grams predictor for it to be able to predict the future actions of the player. Learning techniques can be classified into online and offline learning techniques (see Millington 2006, 564).
Online learning is performed during the game, for example while the player is fighting an enemy NPC. The AI would adapt to the player during the fight. Offline learning is done during the development of a game, often in the Quality Assurance phase of game development (see Ponsen and Spronck 2004, 7-8). Offline learning is done by processing data collected from games played by human players and trying to calculate strategies or parameters from them (Millington 2006, 564).
The benefit of online learning is that it enables the AI to be very adaptive to a certain player. The disadvantage of it is that it's hard to debug for the developer because the behavior is hard to replicate. It can also lead to unwanted results. For example, if the NPC learns that it's good to run against the wall. Another disadvantage of online learning is that the alogrithms need to be quite fast, because they need to be performed at runtime. This limits the range of techniques which can be applied in the game (Millington 2006, 564).
Offline learning is easier to debug because the behavior of a NPC doesn't change during runtime. The results from offline learning can be tested more easily for the same reason. Since offline learning doesn't happen during runtime, it also allows more time consuming techniques to be utilized. Offline learning, however, requires some sort of data corpus to learn. For games this data corpus would be recored games of human players. Collecting such data can represent quite some effort for a game developer. Offline learning is also only able to provide an AI which is adapted to the data provided by the developer. If a player would display an unknown behavior the AI wouldn't be able to react to it accordingly. Since offline learning doesn't adapt to a certain player during runtime, it won't be able to provide an individualized experience for the player and could lead to the repetitive behavior, as described in the example earlier (see Millington 2006, 564).
N-Grams can be used for both online and offline learning. In linguistics it's common to use offline learning for N-Grams because it's not necessary to adapt them to a specific human since the usage of a certain language doesn't vary much from human to human. In computer games it depends on the game and what the N-Grams are used for, if online or offline learning should be used. Laramée suggests to use offline training with a corpus of games played by different players. He provides a rule of thumb about how big that corpus should be in the form of the formula seen in formula 1.

$$C = k * M^{N-1} \tag{1}$$

In this formula N is the order of the N-Gram, k is a constant between 10 and 25 and M is the number of possible actions and C is the numbers of games which need to be played. Why the value of k should be in that range is not explained (see Laramée 2002, 599-600).

As mentioned earlier the collection of a corpus of games can provide difficulties for a game developer. This is the case because a corpus of one game can't be used for another game since the games are different. It's the the same as in the field of linguistics where a corpus of one language can't be used to learn N-Grams about another language because the languages are different. Besides that it requires some resources to acquire a corpus of games and AI often isn't a top priority in video games. But there are smoothing algorithms, which deal with the problem of small training data:

> "One of them (smoothing algorithms) ... involves computing the probability that the next sequence observed, if the training corpus were larger, would be one previously unseen. This value is then divided by the number of legal sequences that do not appear in the corpus, and the quotient is assigned to each of them. The probabilities of sequences that appear in the training corpus are then discounted by a related factor to bring the total probability mass back to 1." (Laramée 2002, 600)

When the sequences which should be analyzed are rather large and N-Grams are used for online learning it can take quite long until the N-Grams predictor is able to predict properly. This is the case because each sequence will need to be registered several times to get enough data for the predictor to be work accurately. This will take time and the performance of the predictor will be poor until then. Hierarchical N-Grams are a technique suggested by Millington to solve this problem (see Millington 2006, 588).
Depending on the game, it might be required to analyze a longer sequence of the players' actions. In this case a larger window size would give the N-Grams predictor the potential to reach a better performance. A smaller window size would potentially not be as good, but would reach its peak performance faster. Hierarchical N-Grams combine several N-Grams with different window sizes to work in parallel. This way a N-Gram predictor of a smaller order can make predictions until the larger order N-Gram predictor has collected enough data (see Millington 2006, 588).
When using hierarchical N-Grams any given sequence data is processed by all used N-Grams. This is best explained in an example:

ASD AAD AAD SAA ASD

A hierarchical 3-Gram, consisting of a 3-Gram, a 2-Gram and a 1-Gram, would register the last sequence of this sample data of actions as follows: The 3-Gram would register "ASD", the 2-Gram would register "SD" and the 1-Gram would register "D". This would be done for every sequence. When the hierarchical 3-Gram is required to make a prediction about a given set of actions, it will first try to use the 3-Grams to predict the most probable future action. If the 3-Gram doesn't have enough data to make a prediction, because the given set of actions wasn't seen often enough, the 2-Gram predictor will try to predict the most probable future action. If the 2-Grams fail as well the 1-Grams predictor will try to predict. If even the 1-Grams fails, it's not possible to make a prediction with the hierarchical 3-Grams. In this case the AI could make a random guess (see Millington 2006, 588).
It's up to the game developer to decide how often a sequence should be registered until a N-Gram of a certain order should make a prediction based on that data. For example it might be better to let the 2-Gram make a prediction instead of the 3-Gram, if the sequence "SD" was registered five times and the sequence "ASD" was only registered one time. There isn't one correct value about how often a sequence needs to be registered. It's possible to find this value by trial and error. According to Millington, 3-4 is a good value to start testing with. N-Grams outside of the field of video game AI often use larger values. But considering the consequences of decisions made by a game AI aren't as sever as those of AIs in other fields, it's no problem to use smaller values (see Millington 2006, 588).

3 N-Grams implementation in the game *Blocky*

3.1 Unity

The game *Blocky* was developed with the Unity engine (Unity 2012). Since the focus of this thesis is the N-Grams technique there would have been no benefits in developing the game without such an engine. This engine was chosen as development environment because it offers a lot of the features needed to implement such a game. The author's previous experience with this engine and the fact that the Salzburg University of Applied Sciences provided the author with the opportunity to develop with Unity were influencing factors as well.
Unity supports three scripting languages: JavaScript, C#, and a dialect of Python named Boo. All three are equally fast and can interoperate. All three can make use of .NET libraries which support databases, regular expressions, XML, networking and so on (Unity 2012). Most of the game functionality is usually programmed with such scripts. Unity C# scripts define a class which inherits from MonoBehaviour. This enables the class to define functions like update, Start and several OnMouse functions. The update function is called before rendering a frame.
Unity provides an integrated Editor which allows the developer to build the game world in a real time environment. With the editor the game world and the GameObjects can be created. A GameObject is an object which can have several components, like a Renderer, which is responsible for drawing the GameObject. The properties of every GameObject in Unity can be viewed and edited in the Inspector, which is a window in the editor.

3.2 Game rules

In the game *Blocky* the player has to choose one of five different blocks to beat his opponent. Each block has a different color, can beat two other blocks and can be beaten by two other blocks. The game is round based and the players pick one block per round. At the end of the round the game decides the winner based on the rules. The winner of the round will gain one score point. The goal is to reach a certain amount of score. The player, which reaches this score first, wins the game. It's possible to gain up to two points more than the score needed to win due to the bonus scores.
The game is controlled either by picking cubes with the mouse, or by pressing the keys a, s, d, f or g to pick a cube in accordance with the order by which they are displayed in the game.
The game supports two players and a player can be either a human or an AI. The AI can be set up to pick by random, predict the other players choice with N-Grams and to pick a counter or to repeat a pattern. The visualization is focused on the human vs AI scenario with the human playing as player 1.

The game has a few basic rules about which block beats which. The rules are similar to a five way Rock-paper-scissors game and are shown in figure 1. The arrows describe which other blocks a block can beat. For example red can beat blue and white, but gets beaten by green and black. Besides those basic rules there are is a bonus system which creates combos. If a player can beat his or her opponent with a combo he or she will get a bonus score. Every block has two combo blocks, which either give one or two extra score. The rules for the bonus system can be seen in figure 2. In that figure the arrows describe how much bonus score can be gained, if the block at the point of the arrow was the previously picked block. For example, if a player wins a round with a red block and picked a blue block in the last round, he or she will gain two bonus score.

The appearance of the game can be seen in figure 3. The appearance was kept very simple since the focus of this thesis is the N-Grams technique and not the visuals of the game. The first rows of colored squared are the blocks of player 1, the second row are the blocks of player 2. The smaller blocks on top of the block of player 1 are the visualization of the bonus blocks. They describe

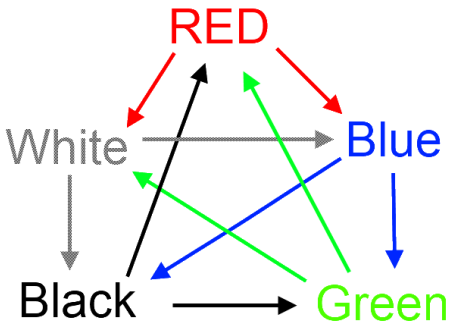

Figure 1: The basic rules of *Blocky*

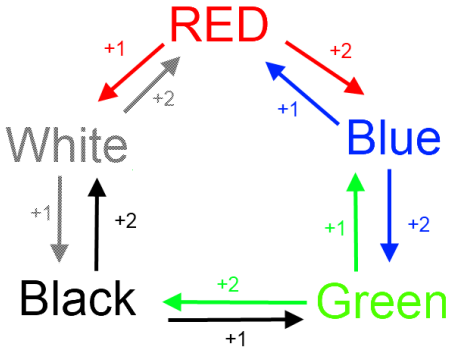

Figure 2: The bonus rules of the combo system

how much bonus score can be gained by winning the current round with that block if one of the smaller blocks was the previous block. On the right side are two small tables which display the current score as well as the information about the score needed to win. On the very left are the game rules which get magnified if the player hovers the mouse over them. The text at the very bottom displays the result of the round and the winner at the end of the game. There are three buttons at the bottom. The "Next Round" manually starts a new round if the player clicks on it. The "Auto Mode" button enables the "AutoMode" which means that the player doesn't manually have to start a new round. If the "AutoMode" is ON the new round will automatically start after the delay specified with the "Auto Mode Delay" button. The standard value for that delay is one second. The possible values are: 0, 0.5, 1 and 1.5 seconds.

As mentioned earlier N-Grams are described as being good at predicting moves in fighting games. Of course the production of such a game required too much resources and therefore it wasn't possible to produce an entire fighting game for this thesis. The mechanics of the game *Blocky* and a fighting game aren't that different however.
In a typical beat em up game there is a certain amount of moves which each have advantages and disadvantages. A punch, for example, might have a smaller range than a kick but could be faster. In *Blocky* there's also a certain amount of moves and the blocks also each have advantages and

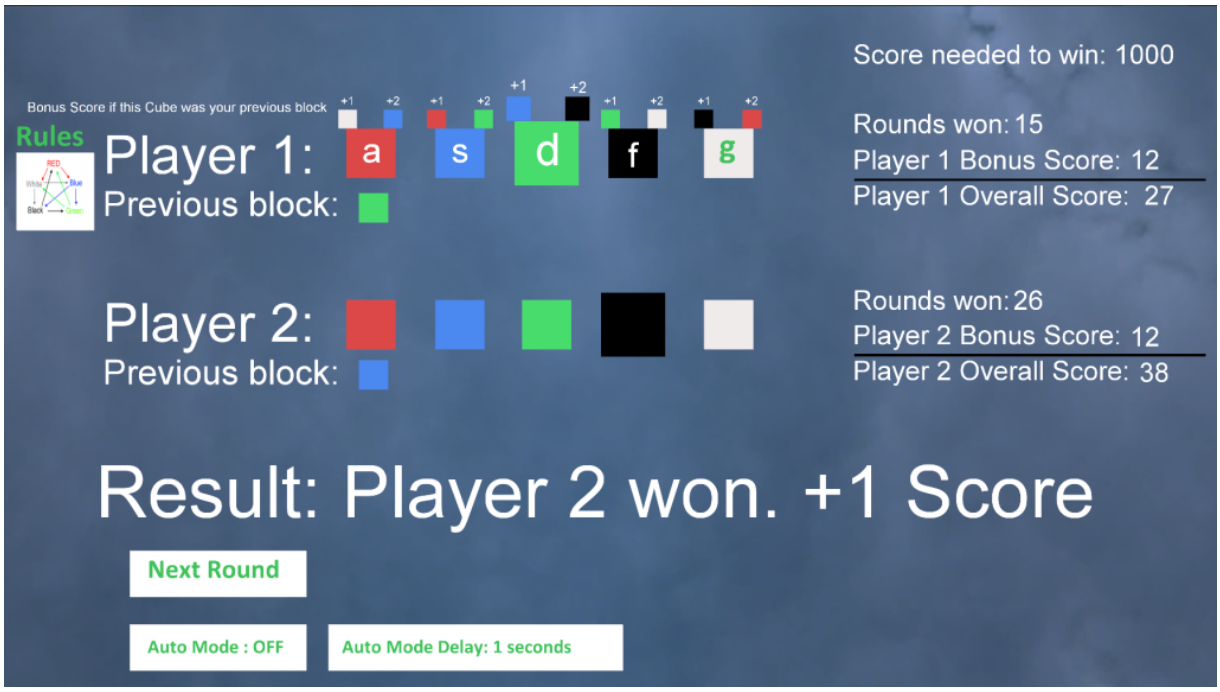

Figure 3: A screenshot of the game *Blocky*

disadvantages.
In many games there's a reason why a player will repeat his or her actions. Those reasons could be cooldowns (the time until the ability can be used again), resources needed to complete actions or simply bonus effects like an enemy taking bonus damage if it is hit by Attack2 after being hit by Attack1. In a beat em up game, for example, the character would do a special attack if the player managed to land two punches and three kicks. Such game mechanics strongly encourage the player to learn and repeat the best possible combination of abilities. *Blocky* tries to simulate such a mechanic with its bonus system as explained earlier and shown in figure 2. Without the bonus system it wouldn't have mattered for a human player what his or her previous choice was. That might have led to the player just randomly picking blocks which is usually not the way players play a game.
The choice of which action is chosen can also be influenced by how the actions are mapped to the keyboard or gamepad. For example, if the movement of the character is controlled with the a, s, d and f keys, the player is more likely to use an ability that is mapped to r than one that is mapped to p. That is the case because r is nearer to the movement keys than p. To simulate such effects the blocks are ordered in a certain order on the screen and have keyboard keys assigned to them in accordance of their appearance on the screen.
The game *Blocky* was developed solely for the purpose of testing the N-Grams technique implemented for this thesis.

3.3 Implementation of the game *Blocky*

The game *Blocky* was developed with Unity 3.5. The whole project can be found on GitHub under following URL: `https://github.com/mwebi/Blocky`.

The blocks are simple cubes which are a standard object of Unity. Their functionality gets defined in the Cube.cs script. This script reacts to mouse and keyboard controls for the cube and allows the cube to be selected by the player.

A block can be of a certain CubeType. CubeType is an Enum defined in the Cube.cs script as well. Possible CubeTypes are red, blue, green, black, white and none. The script allows the developer to set the CubeType for a particular cube in the Unity Inspector through the public field cubeType.

A lot of the gameplay and game logic is done in the GameLogic.cs script as seen in section C of the appendix. This script has several properties, which are mostly references to other GameObjects, which get set in the Start function or in the Unity Inspector. This script is attached to the GameLogic GameObject, which doesn't have any visual representation.
In the update loop the script checks first if the round or the game has already ended. If that is not the case the script will let the AI pick its cube provided it's the turn of the AI to pick one. After that the script checks if both players picked a cube. If that is the case the winner is determined, the scores are updated and the game checks if a player reached the score required to win the game. The function determineWinner determines the winner by using the winLooseMatrix array. This two dimensional integer array makes it possible to look up the result for any given input. The input are the two CubeTypes, which the players picked, casted to integers. The output is an integer with the value 0, 1 or 2 which represents the results draw, player 1 won and player 2 won. This approach of determining the winner seemed preferable to a huge switch statement.

As mentioned earlier the game has two players which can be either human or an AI. The players are represented with a GameObjects, called player 1 and player 2, which each have the script PlayerScript.cs attached to them. That file is displayed in section B of the appendix. The GameObject are the parent of several other GameObjects, which consist mainly of the GameObjects, which represent the GUI text. The PlayerScript itself has several properties for the same purpose as the gamelogic script. One of those properties is ngramPredictor of the type NgramPredictor, which is as the name suggests the N-Gram predictor.
The Start function only initializes the ngramPredictor if the player is a N-Grams AI. The function pickCubeByAI handles by which method the AI will pick its cube depending on how the AI is set up to act. The random AI will simply decide on a cube to pick with the Random.Range function provided by Unity. The pattern AI will continue to pick cubes in the order red, blue, green, black and finally white. The N-Grams AI will use the ngramPredictor to predict the most likely pick of the other player. That estimation is then passed on to the pickCounterCube function which returns one of the two cubes which will beat the given cube. If the ngramPredictor returns CubeType none it means that there couldn't be any estimation made and so the N-Grams AI will randomly choose a CubeType to pick.
The PlayerScript.cs also takes care of storing and sorting the last actions of the player. Those actions are stored in the previousCubePicks array of the type CubeType. If a player picks, for example, the red block, this action is stored by setting the last member of the Array to CubeType red. This data is used for registering the players action with the N-Grams predictor. It is only used if the enemy is actually using a N-Grams predictor.

3.4 Details of the N-Grams implementation

The N-Grams implementation done in *Blocky* is based on the pseudo code provided by Millington (see Millington 2006, 583). It can handle any window size greater than 1. It uses online learning, which means that the N-Gram predictor adapts to the current player. No data is stored permanently, the N-Gram predictor starts every game without any data about the current or former opponent.
The N-Grams implementation is done in the NGram.cs file and can be seen in the appendix in listing 1. This file defines the KeyDataRecord struct and the NGramPredictor class.
The two most important functions in the NGramPredictor class are registerSequence and getMostLikely. The main data object is the hashtable data. It stores a KeyDataRecord for every sequence

of actions which gets registered.
The KeyDataRecord struct stores the total times a sequence was registered. It also counts how often which action was taken by the player after that sequence. This is stored in the counts hashtable, which takes the CubeTypes as key and has ints as value. Figure 4 shows the contents of the data hashtable during the game.

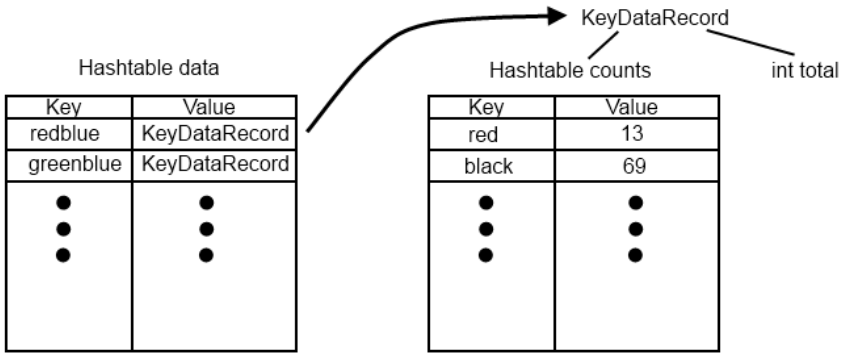

Figure 4: The contents of the data hashtable during the game

The registerSequence function has one parameter called "actions", which is a CubeType array. This array contains the last N, where N is the window size + 1, actions of the player which get analyzed. This array is split up into the current action and a smaller array with the previous actions. The array with the previous actions serves as key for the data hashtable. The hashtable stores a KeyDataRecord for each key. This KeyDataRecord in turn has a hashtable counts. The current action of the player is used as the key for the counts hashtable. For registering the sequence the value of this key and the total counter of the KeyDataRecord are simply incremented.
A closer look on the registerSequence function reveals that it was necessary to use a string as the key of the data hashtable instead of the array itself. This string is created by the names of the actions. For example the string for the actions blue and red would be "bluered". It was necessary to use such strings as keys instead of the arrays themselves, because the hashtable would register arrays with the same content as different keys. So two arrays which both contain red and blue as actions in the same order would get registered as two different keys because they are seen as a different object by the hashtable. This could have been prevented by allocating an array for every possible combination of actions before registering anything. But this approach doesn't scale well in situations where most actions never occur. Besides that it's generally presumed to be bad practice to use strings for such use in video games, so this detail would better be reworked for an implementation in a commercial game.
The getMostLikely function has also one parameter called "actions", which is also an a CubeType array. This array however is exactly as large as the window size. It contains the actions of the player to which the most likely succeeding action should be found. This is done by first creating a string out of the actions array again. This string is used to access the data hashtable to retrieve the KeyDataRecord for this sequence. The counts hashtable of that KeyDataRecord is then looped through to find the action with the highest value. The value represents how often that action was performed by the player. The highest value is therefore the most likely action, which the player will perform. In the end that action is returned to the PlayerScript which called the getMostLikely function.

4 Test set-up and expectations

The performed tests can be classified into two categories, "AI vs AI" and "AI vs human". The AI vs AI category in turn is divided in several tests. The N-Grams based AI will be set up to play against a random AI, a pattern based AI and another N-Grams based AI. Those tests will be mainly performed to determine if the implementation works correctly. They also serve to assess the performance of the N-Grams algorithm and how long it takes to make a prediction. The N-Grams AI is set up to use trigrams in all cases. The AIs will play 100 games to a score of 250 for every scenario.
In the N-Grams vs random AI scenario the two AIs should roughly perform equally good. The random AI should choose actions by random and the N-Grams AI shouldn't be able to perform better than the random AI, because there is no pattern which the N-Grams AI could predict.
Same applies to the N-Grams vs N-Grams scenario. The AIs should start by making random decisions because they have no data for the opponents last moves. Those first random decisions of each AI would be the basis of the predictions the other AI would make. Since the data which the predictions are based on is random, the predicted action would be random as well. This cycle would continue until the end of the game. It should be random which one of the N-Grams AIs wins the game.
The pattern AI vs N-Grams AI should proof that the N-Grams implementation is actually able to successfully predict player behavior. The pattern AI presents the best case scenario for the N-Grams AI. It's a player which behavior follows a pattern for every action. In the worst case the N-Grams AI should be able to win every turn after the pattern was completed for the first time. In the best case the N-Grams AI would win every round, if it's able to win the rounds until the pattern is repeated by choosing random.

Provided that the N-Grams implementation succeeds in the previously described tests, the AI vs human tests should reveal if and why a N-Grams AI is considered a better opponent than a random AI by human players. It will also reveal how successful the N-Grams AI is against human players.
Several participants will be set up to play one game up to a score of 250 against a random AI and a N-Grams AI. The rules of the game and the controls and the interface will be explained to them. They will not know which kind of AI they are playing against. Half of the participants will play against the random AI first and the N-Grams AI second, the other half against the N-Grams AI first and the random AI second. The participants will also not be told that one of those AIs is trying to predict their behavior. The participants will be observed during the game and an observer will fill out the form seen in figure 5 for every participant.
The observer will ask the participants the two questions written at the bottom of the form. The first question is worded that way because the goal of the question is to find out which AI the participants prefer. A question like "Which AI predicted your actions better" would only yield information about what AI the participant thinks predicted better, but it wouldn't actually give information about the participants preference. Valid answers for the first questions are: random, N-Grams or draw. The second question tries to get further information about the participants preference. This information might reveal if the participants actually enjoy playing against a N-Grams AI.
The bonus system of *Blocky* should lead to people preferring certain combinations of blocks. This strategy will provide them with an advantage over the random AI since it doesn't focus on bonus score. But the participants might abandon this strategy if the N-Grams AI is able to detect and counter it.
If the participants switch to random choices, the N-Grams AI should still be able to find some patterns in their behavior. Literature suggests that humans tend to prefer certain sequences to others when tasked to produce random sequences (Griffiths and Tenenbaum 2001).
The key alignment and the order of the blocks might also influence the participants. This might also lead to patterns in their behavior which the N-Grams AI can predict.

Figure 5: Test form filled out for every participant by the observer

If a human participant is bored by the game and / or decides to play on automode with 0 seconds delay, that might lead to the participant just pressing keys on random. But for the reasons stated earlier this behavior should be countered by the N-Grams AI.

5 Discussion of the N-Grams implementation and the test results

The data analyzed in this section can be found in the appendix in section D.

5.1 Performance

As discussed earlier there are several hashtables used in the implementation. As long as those hashtables aren't full, retrieving and assigning hashes are constant time processes (Millington 2006, 585). With only accessing the hashtables the time complexity of the function would be $\mathcal{O}(1)$. But it's necessary to split the actions sequence into a key and a value for the hashtable. This

is done by looping through the actions array and copying the members. This leave us with a time complexity of $\mathcal{O}(n)$ for the registerActions function, where n is the window size used for the N-Grams predictor.
In the getMostLikely function we have to loop through all possible actions to find the best one. Besides that procedure there's nothing done which is relevant for the time complexity. The getMostLikely function is therefore $\mathcal{O}(m)$ in time where m is the number of possible actions.
It would be possible to swap the time complexities of those two functions by sorting the counts hashtable by value. But since in most use cases the registerActions function is called more often than the registerActions, this isn't necessary (Millington 2006, 585).
The memory complexity of the N-Grams predictor is $\mathcal{O}(m^n)$, where n is the nValue and m is the number of actions. The nValue is the window size of the N-Grams predictor plus one (Millington 2006, 585).

Some tests were performed to measure the performance of the N-Grams predictor in practice. The average time was measured by taking the time of how long the registerActions and the getMostLikely take to finish from the first to the last line. In all tests the N-Grams AI played against the pattern AI. That way it was possible to get results for the getMostLikely function with higher windows sizes. Using a random AI would have yielded no results because the chance of meeting a sequence, which can be predicted, is very slim when using larger window sizes. The hardware specifications, on which those tests were preformed, are as follows. CPU: AMD Phenom II X6 1090T, Memory: DDR3-RAM KIT 8 GB 1333 MHz CL7, GPU AMD Radeon HD6950. The results can be seen in table 5.
With window size of 2 the average time to register a sequence of actions and to predict one was both roughly 0.02 milliseconds. A frame rate of 60 FPS (Frames Per Second) is considered to be good for a video game. With that frame rate a video game has 16.67 milliseconds to update and draw the game per frame. Considering this, the 0.02 milliseconds the N-Grams predictor needs for its operations are very good. Therefore it can be concluded that the implementation as discussed earlier could be used in a state of the art video game.
The average times increased to around 0.1 milliseconds with a window size of 20. This is still suitable for state of the art video games, but it limits the use cases. For example, it wouldn't be possible to equip a larger number of NPCs with such a N-Gram predictor, but it would still be possible for a small number of special NPCs.
With a window size of 200 the average times went up to roughly 1.1 milliseconds. Since the N-Grams predictor would only be a small part of a whole game this value might be too large to be used in a state of the art video game. On the other hand it's questionable in which scenario an AI would need to predict an action based on the previous 200 actions.
The built in profiler tool of Unity was used to confirm the results of the test scenario with window size 2. The screenshots of the profiler can been seen in figure 7 and 6 in the appendix. The profiler suggests the functions take more time to complete on the first glance. It suggests a time of 0.07 milliseconds for both functions. But if studied in detail it's evident that 0.05 respectively 0.06 milliseconds are taken up by the parsing of the actions array to a string format. That procedure also produces some memory allocations. This means that the N-Grams predictor on its own performs well, but that the parsing should not be done if this implementation were to be used in a state of the art game. Both, the wasted time and the allocations, are considerably large. The memory allocations are undesirable in a video game, because they are done during the runtime of the game. This would trigger the garbage collection at some point, which in turn will cause a small stutter in the game.

5.2 AI vs AI

The N-Grams AI vs pattern AI tests generated results as expected. The data can be viewed in table 2 in the appendix. The N-Grams AI won every game and detected the pattern of its

opponent every time. As expected the N-Grams AI never lost more than 6 rounds in one game. In some games it won without loosing a single round. Those results proof that the N-Grams implementation, provided in this thesis, is in fact able to predict player behavior.
The N-Grams vs N-Grams scenario also turned out as expected. The two N-Grams AIs performed roughly equally good. As displayed in table 4 the N-Grams AI 1 won 44 games where as N-Grams AI 2 won 56 games. This result might seem a bit unbalanced but should be viewed in combination with the scores. The average scores of the two AIs are very similar. The difference would decrease even further if more games were played.
Finally the N-Grams vs random AI tests didn't turn out as expected. As shown in table 3 the random AI won with 71 against 29 games. The rounds of the N-Grams and the random AI are nearly identical. The reason why the random AI won more games is that it was able to achieve roughly 20 more bonus score on average per game than the N-Grams AI. The N-Grams prediction worked nevertheless, it just didn't have any pattern it could predict. Apparently some part of the logic of the AI, implemented in the playerScript.cs file, causes this result. It could be the part in the pickCounterCube function, where the N-Grams AI randoms, which of the two countering cubes to pick.

5.3 Human vs AI

The data analyzed in this subsection can be found in the appendix in section E. 16 people were willing to participate in the "Human vs AI" test. All of them were students of the University of Applied Sciences Salzburg. Some of the participants were at first a bit overwhelmed with the rules of *Blocky*. It usually took them some time to figure out which block beats which and the rules about the bonus system. Some where trying hard to analyze the opponent AI because they assumed that their opponent will go for some kind of strategy. That often led to those players being especially frustrated with the random AI since it didn't display any pattern which could have been detected. A few participants also became bored of the game at some point. This can be attributed to the rather not exciting visualization of the game and the size of the score which needed to be reached to win the game. Some participants, like participant #15, seemed to haven't given the rules, especially the bonus system, much thought. That particular participant claimed the chance of winning in the game is 50%, which is not the case because of the bonus system mechanics.
Millington claims that a N-Grams AI can provide an unbeatable opponent in combat games (see Millington 2006, 583). The tests conducted with human players underpin this statement. Out of the 16 participants no one was able to beat the N-Grams AI. The random AI was beaten by 12 participants. 8 of the participants played against the random AI first and the N-Grams AI second. The results show that there is no difference in which order the participants encountered the AIs. The average time the players took to finish their match against the random AI is 8 minutes and 4 seconds. The average time of the matches against the N-Grams AI was 12 minutes and 1 second. The longer average time against the N-Grams AI can be attributed mainly to participant #2, who played over 53 minutes against the N-Grams AI.
The N-Grams AI not only won every game, but it also achieved an average score of 250,63, which is higher than the average score of 206,31, achieved by the random AI. The N-Grams AI was also able to reach a higher bonus score than the random AI. When playing against the N-Grams AI the participants average scores were noticeable lower than when they played against the random AI.
Without the bonus system of *Blocky* the random AI would probably have won around half of the games. In the 4 games which were won by the random AI the participants had a low bonus score amount. Since the random AI doesn't utilize bonus combos, it lost against players who figured the bonus system out and used the bonus combinations.
Best example for this is participant #3. In that case the random AI actually won more rounds than the participant, but because the participant utilized the bonus system he won nevertheless. In the debriefing that participant stated that he tried to use the same tactic against the N-Grams

AI but failed.
As written earlier participant #2 played over 53 minutes against the N-Grams AI. According to him, he tried his very best to predict his opponent and to outmaneuver him. His scores are below average which suggests that this strategy did not work. This indicates that even if a human player puts a lot of effort into it, it's hard to beat the N-Grams AI.
Participant #13 achieved the lowest scores of all participants against the N-Grams AI. This particular participant played against the random AI first and was frustrated because he couldn't figure out the system by which the AI chose its blocks. He became frustrated by this experience and ended the game against the random AI by pressing random keys. That way he lowered the chances of winning against the random AI because he didn't use the bonus system and in the end lost with 226 against 250 overall score. In his second game he continued to press buttons without any strategy. He repeated them in the same order. The N-Grams AI recognized this pattern and countered it very successfully.
The closest to beating the N-Grams AI was participant #18 with an overall score of 225 points. He played quite slowly and tried to play by random. He picked the blocks by pushing keys on the keyboard, but used only his right hand for the first half of the game. That lead to certain patterns the N-Grams AI was able to detect and counter. After reaching around 150 overall score however he switched to pressing keys with two hands. This changed the combinations of blocks he chose significantly. The N-Grams AI didn't performed well until the counts for the new patterns were higher than the counts for the old patterns.
Considering this data one could conclude that the N-Grams AI in the game *Blocky* is in fact unbeatable. However this is not the case. Due to the bonus system a human player can beat the N-Grams AI. Theoretically he or she could play half of the rounds in a way which would let the N-Grams AI win every round without gaining any bonus score. He or she could then switch his pattern so that the N-Grams AI would predict his behavior in the wrong way because it has been fed with data from earlier. That would lead to the N-Grams loosing for approximately the amount of round which have been played. If that second pattern applied by the human player grants him bonus points he or she should be able to win the game because the N-Grams AI didn't obtain those bonus points. However, if the goal of the developer was to create an unbeatable AI, this strategy could be countered. The AI could simply be enabled to detect long streaks of round losses and react by changing its decisions.
To the question "Which AI did you feel was better?", 9 participants answered that the N-Grams AI was better, only 3 answered that they felt the random AI was better and the other 4 thought the AIs were equal.
In the follow-up question the participants were asked why they considered that AI better. They delivered a variety of answers.
Out of the 4 participants who answered that they consider the AIs equally good, 2 reasoned that both didn't follow any recognizable pattern. Participant #9 argued that both AIs were able to recognize his strategy and countered it. Participant #14 reasoned that the N-Grams AI was able to recognize and counter his patterns, but that he enjoyed playing against the random AI more. Participant #15 reckoned that the random AI was more predicable than the N-Grams AI and that the chance of winning in the game is 50% anyway.
Participants #4, #12 and #15 both deemed the random AI better. Participants #4 reasoned that the random AI followed a pattern, which was effective against his pattern. Participants #12 doesn't like to loose and therefore preferred that random AI because he was able to win against it.
Out of the 10 participants which opted the N-Grams as the better AI, the participants #3, #5, #6, #7, #8, #10, #11 and #18 all thought that the N-Grams AI recognized and countered their strategy. They valued this feature of the AI. Participant #5 stated that he tried his best to not play by any system, which could be detected but felt like the AI knew what he would pick nevertheless. Participant #10 and #13 valued the simple fact that the AI won as factor why they consider it to be better.
Considering the results as described above, it can be concluded that the majority of the participants appreciate a N-Grams AI, because it was able to predict and counter their strategies. Some

players, however, prefer to win and not be challenged that hard. So an unbeatable AI might not be the most fun opponent to play against. In a commercial game, which usually has the goal to entertain the player, it might be best to balance the AI to be challenging but not overwhelmingly strong.

6 Conclusion

N-Grams have been used a lot in the field of linguistics, but aren't employed often in video games. Game developers often prefer techniques like decision trees and finite state machines, when developing an AI for a video game. Such techniques give the AI a variety of behaviors, without the danger of the player experiencing any unforeseen AI behaviors, because the AI doesn't adapt to the current player. But adaptive AI techniques, like N-Grams, can provide additional depth to the AI, challenge the player harder and give him or her an unique experience.
The answers to the research questions from subsection 1.2 can be summarized as follows.

- How can N-Grams be implemented in a Unity game?

This question was answered by providing the code of the N-Grams predictor implementation in the appendix in listing 1 and by discussing that code in subsection 3.4. Furthermore some of the core parts of the game code are listed in the appendix in sections B and C. The whole Unity project is available on GitHub: `https://github.com/mwebi/Blocky`. Regarding this information it can be said that N-Grams predictor was successfully implemented and then integrated into the Unity game *Blocky*.
One improvement, which could be made to the implementation, would be the usage of the bonus system by the AI. This could be done by checking which of the two cubes, that counter the predicted cube, grant a bonus score. Another improvement would be to not use strings as keys for the data hashtable.

- How successful are N-Grams at predicting future player actions?

The tests conducted in both subsection 5.2 and 5.3 proofed that N-Grams are very successful at predicting future player actions in a video game. The N-Grams vs pattern AI tests proofed that the N-Grams implementation provided actually works in the game. The N-Grams AI was able to recognize, predict and counter the repeating pattern provided by its opponent. The tests where human players played against the N-Grams AI yielded the clear result that the N-Grams AI is also able to recognize, predict and counter human behavior. The N-Grams AI won against all 16 participants.
This raises the question if such an AI is really desirable. Some test participants were frustrated because they lost, some even preferred playing a random AI solely because they could win against it. On the other hand many participants appreciated that the N-Grams AI outmaneuvered them and that this behavior gave the AI a certain edge.
It might be better to enable the AI to predict the behavior but not use that knowledge to counter the human player without leaving him or her a chance to win. For example, in the game *Blocky* it would be possible to reward the player for coming up with a strategy or for guessing what the opponent will pick. In such cases the AI could just let the human player win. The AI could also be adjusted to only beat the human player when she or he is ahead with the score.

- Does the memory consumption and time complexity of the algorithm allow its usage in state of the art computer games?

Subsection 5.1 presented the results of the performance tests. The tests were conducted in a state of the art environment, a game in the Unity engine, on a modern end user hardware. The results of the performance tests and the field tests with human players proofed that N-Grams can be used in a state of the art computer game.

List of Figures

Listings

List of Tables

A NGram.cs

Listing 1: NGram.cs

```
0  using UnityEngine;
1  using System.Collections;
2
3  public struct KeyDataRecord{
4      //Holds the counts for each successor action counts
5      public Hashtable counts;
6
7      //Holds the total number of times the window has been seen
8      public int total;// = 0;
9  }
10
11 public class NGramPredictor{
12
13     // Holds the frequency data
14     Hashtable data;
15
16     // nValue is the size of the window + 1
17     public int nValue = 3;
18
19     public string playerThisNgramPredictorBelongsTo = "";
20     public bool doDebugLogs;
21
22     public void Start(){
23         data = new Hashtable();
24     }
25
26     private string returnStringName(CubeType[] actionsArray){
27         string name = "";
28         for (int i = 0; i < actionsArray.Length; i++)
29             name += actionsArray[i].ToString();
30         return name;
31     }
32
33     // Registers a set of actions with predictor, updating
34     // its data. We assume actions has exactly nValue
35     // elements in it.
36     public void registerSequence(CubeType[] actions)
37     {
38         if(actions.Length != nValue){
39             Debug.Log("actions.Length != nValue");
40             Debug.Break();
41         }
42
43         // Split the sequence into a key and value for the Hashtable
44         CubeType[] previousActions = new CubeType[nValue-1];
45         for (int i = 0; i < nValue-1; i++)
46         {
47             previousActions[i] = actions[i];
48         }
49
```

```
50          CubeType currentAction = actions[nValue-1];
51
52          string keyStringOfArray = returnStringName(previousActions);
53          if(doDebugLogs)
54              Debug.Log(playerThisNgramPredictorBelongsTo +
55              "registered␣sequence:␣" + keyStringOfArray +
56              "␣+␣" + currentAction);
57
58          if (!data.ContainsKey(keyStringOfArray))
59          {
60              data[keyStringOfArray] = new KeyDataRecord();
61          }
62
63          // Get the correct data structure
64          KeyDataRecord keyData = (KeyDataRecord)data[keyStringOfArray];
65
66          if (keyData.counts == null)
67              keyData.counts = new Hashtable();
68
69          // Make sure we have a record for the follow on value
70          if(!keyData.counts.ContainsKey(currentAction))
71              keyData.counts[currentAction] = 0;
72
73          // Add to the total, and to the count for the value
74          keyData.counts[currentAction] = (int)keyData.counts[currentAction] + 1;
75          keyData.total += 1;
76
77          data[keyStringOfArray] = keyData;
78      }
79
80      // Gets the next action most likely from the given one.
81      // We assume actions has nValue - 1 elements in it (i.e.
82      // the size of the window).
83      public CubeType getMostLikely(CubeType[] actions)
84      {
85          if(actions.Length != nValue-1){
86              Debug.Log("actions.Length␣!=␣nValue");
87              Debug.Break();
88          }
89
90          string keyStringOfArray = returnStringName(actions);
91
92          //if we don't have data for the actions, we cant predict
93          if (data[keyStringOfArray] == null){
94              if(doDebugLogs)
95                  Debug.Log(playerThisNgramPredictorBelongsTo +
96                  ":␣Prediction␣fail:␣no␣data␣for␣this␣sequence:␣" +
97                  keyStringOfArray);
98              return CubeType.none;
99          }
100         if(doDebugLogs)
101             Debug.Log(playerThisNgramPredictorBelongsTo +
102             "␣predicting␣sequence:␣" + keyStringOfArray);
103
```

```
104         // Get the key data
105         KeyDataRecord keyData = (KeyDataRecord)data[keyStringOfArray];
106 
107         // Find the highest probability
108         int highestValue = 0;
109         CubeType bestAction = CubeType.none;
110 
111         // Get the list of actions in the store
112         ICollection possibleActions = keyData.counts.Keys;
113 
114         // Go through each
115         foreach(CubeType action in possibleActions){
116             // Check for the highest value
117             if ((int)keyData.counts[action] > highestValue)
118             {
119                 // Store the action
120                 highestValue = (int)keyData.counts[action];
121                 bestAction = action;
122             }
123         }
124 
125         return bestAction;
126     }
127 
128 }
```

B PlayerScript.cs

Listing 2: PlayerScript.cs

```
0  using UnityEngine;
1  using System.Collections;
2  
3  public class PlayerScript : MonoBehaviour {
4  
5      public CubeType pickedCube;
6      public CubeType previousCube = CubeType.none;
7      public bool hasPickedACube = false;
8      public bool isAI = false;
9      public bool isAINgram = false;
10     public bool isAIPattern = false;
11     public bool isAIRandom = false;
12     public bool doDebugLogs = false;
13 
14     public int NgramWindowSize = 2;
15 
16     gamelogic gameLogicScript;
17 
18     public int roundsWon = 0;
19     public int OverallScore = 0;
20     public int BonusScore = 0;
21 
22     public GameObject RoundsWonTextNumber;
23     public GameObject ScoreTextNumber;
```

```
24      public GameObject BonusScoreTextNumber;
25
26      GameObject CurrentPrevCube;
27      public GameObject PrevCubeRed;
28      public GameObject PrevCubeBlue;
29      public GameObject PrevCubeGreen;
30      public GameObject PrevCubeBlack;
31      public GameObject PrevCubeWhite;
32
33      public CubeType[] previousCubePicks; //[0] is oldest, [ARRAYSIZE-1] newest
34
35      public NGramPredictor ngramPredictor;
36      private PlayerScript otherPlayerScript;
37      // Use this for initialization
38      void Start () {
39          previousCube = CubeType.none;
40          gameLogicScript = GameObject.Find("GameLogic")
41          .GetComponent<gamelogic>();
42
43
44          if(isAINgram){
45              ngramPredictor = new NGramPredictor();
46              ngramPredictor.Start();
47              ngramPredictor.nValue = NgramWindowSize + 1;
48              ngramPredictor.doDebugLogs = doDebugLogs;
49              ngramPredictor.playerThisNgramPredictorBelongsTo
50              = this.gameObject.ToString();
51          }
52
53          otherPlayerScript = gameLogicScript.returnOtherPlayer(this);
54          //track previous actions if other player uses ngrams
55          if(otherPlayerScript.isAINgram){
56              previousCubePicks = new CubeType[
57              otherPlayerScript.NgramWindowSize + 1];
58
59              for (int i = 0; i < previousCubePicks.Length; i++)
60                  previousCubePicks[i] = CubeType.none;
61          }
62      }
63
64      public void updateScoreText(){
65          RoundsWonTextNumber.GetComponent<TextMesh>().text
66          = roundsWon.ToString();
67          ScoreTextNumber.GetComponent<TextMesh>().text
68          = OverallScore.ToString();
69          BonusScoreTextNumber.GetComponent<TextMesh>().text
70          = BonusScore.ToString();
71      }
72
73      public void Reset(){
74          updatePreviousCubePicks();
75          hasPickedACube = false;
76
77          previousCube = pickedCube;
```

```
78          instantiatePrevCube();
79          pickedCube = CubeType.none;
80      }
81
82      void instantiatePrevCube(){
83          Destroy(CurrentPrevCube);
84          switch(pickedCube) {
85              case CubeType.red:
86                  CurrentPrevCube = Instantiate(PrevCubeRed) as GameObject;
87                  break;
88
89              case CubeType.blue:
90                  CurrentPrevCube = Instantiate(PrevCubeBlue) as GameObject;
91                  break;
92
93              case CubeType.green:
94                  CurrentPrevCube = Instantiate(PrevCubeGreen) as GameObject;
95                  break;
96
97              case CubeType.black:
98                  CurrentPrevCube = Instantiate(PrevCubeBlack) as GameObject;
99                  break;
100
101             case CubeType.white:
102                 CurrentPrevCube = Instantiate(PrevCubeWhite) as GameObject;
103                 break;
104
105             default:
106                 Debug.Log("default called, something went wrong");
107                 Debug.Break();
108                 break;
109         }
110     }
111     public void pickCubeByAI () {
112         if(isAINgram)
113             pickCubeByNGramAI();
114         else if(isAIPattern)
115             pickCubeByPattern();
116         else if(isAIRandom)
117             pickCubeByRandom();
118     }
119
120     private void pickCubeByRandom () {
121
122         pickedCube = (CubeType)Random.Range(0, 5);
123         if(doDebugLogs)
124             Debug.Log(transform.gameObject.ToString() +
125             " picked Cube By Random: " + pickedCube);
126
127         //updatePreviousCubePicks(pickedCube);
128         hasPickedACube = true;
129     }
130
131     private void pickCubeByPattern () {
```

```
132
133         switch(previousCube) {
134             case CubeType.red:
135                 pickedCube = CubeType.blue;
136                 break;
137             case CubeType.blue:
138                 pickedCube = CubeType.green;
139                 break;
140             case CubeType.green:
141                 pickedCube = CubeType.black;
142                 break;
143             case CubeType.black:
144                 pickedCube = CubeType.white;
145                 break;
146             case CubeType.white:
147                 pickedCube = CubeType.red;
148                 break;
149             case CubeType.none:
150                 pickedCube = CubeType.red;
151                 break;
152             default:
153                 Debug.Log("default called, something went wrong");
154                 Debug.Break();
155                 pickedCube = CubeType.red;
156                 break;
157         }
158         Debug.Log(transform.gameObject.ToString()
159         +  " picked Cube By Pattern: " + pickedCube);
160
161         //updatePreviousCubePicks(pickedCube);
162         hasPickedACube = true;
163     }
164
165     CubeType[] getOtherPlayersLastActions()
166     {
167         //lastactions are the previous actions without the current one
168         CubeType[] lastActions = new CubeType[ngramPredictor.nValue - 1];
169
170         for (int i = 0; i < ngramPredictor.nValue - 1; i++)
171             lastActions[i] = otherPlayerScript.previousCubePicks[i+1];
172
173         return lastActions;
174     }
175
176     private void pickCubeByNGramAI () {
177         CubeType predictedOtherPlayerPick = ngramPredictor
178         .getMostLikely(getOtherPlayersLastActions());
179         if(doDebugLogs)
180             Debug.Log(transform.gameObject.ToString() + " predicted: "
181             + predictedOtherPlayerPick);
182
183         //if no prediction was made, make a random guess
184         if(predictedOtherPlayerPick == CubeType.none){
185             pickedCube = (CubeType)Random.Range(0, 5);
```

```
186             if(doDebugLogs)
187                 Debug.Log(transform.gameObject.ToString()
188                 + " picked Cube by Ngram, fallback to random: " + pickedCube);
189         }
190         else{
191             pickedCube = pickCounterCube(predictedOtherPlayerPick);
192             if(doDebugLogs)
193                 Debug.Log(transform.gameObject.ToString()
194                 + " picked Cube by Ngram, counter picked: " + pickedCube);
195         }
196 
197         //updatePreviousCubePicks(pickedCube);
198         hasPickedACube = true;
199     }
200     public CubeType pickCounterCube (CubeType pickToCounter){
201         switch(pickToCounter) {
202             case CubeType.red:
203                 if(Random.Range(0, 1) == 0)
204                     return CubeType.black;
205                 else
206                     return CubeType.green;
207             case CubeType.blue:
208                 if(Random.Range(0, 1) == 0)
209                     return CubeType.red;
210                 else
211                     return CubeType.white;
212             case CubeType.green:
213                 if(Random.Range(0, 1) == 0)
214                     return CubeType.blue;
215                 else
216                     return CubeType.black;
217             case CubeType.black:
218                 if(Random.Range(0, 1) == 0)
219                     return CubeType.blue;
220                 else
221                     return CubeType.white;
222             case CubeType.white:
223                 if(Random.Range(0, 1) == 0)
224                     return CubeType.green;
225                 else
226                     return CubeType.red;
227             default:
228                 Debug.Log("default called, something went wrong");
229                 Debug.Break();
230                 return CubeType.none;
231         }
232     }
233     public void pickedACube(CubeType pickedType)
234     {
235         pickedCube = pickedType;
236         //updatePreviousCubePicks(pickedCube);
237         hasPickedACube = true;
238     }
239 
```

```
240     public void updatePreviousCubePicks()
241     {
242         //dont do anything if other player doesnt use ngrams
243         if(!otherPlayerScript.isAINgram)
244             return;
245 
246         //resort array
247         for (int i = 0; i < previousCubePicks.Length - 1; i++)
248             previousCubePicks[i] = previousCubePicks[i + 1];
249 
250         //previousCubePicks[0] = previousCubePicks[1];
251         //previousCubePicks[1] = previousCubePicks[2];
252 
253         //save current pick
254         previousCubePicks[previousCubePicks.Length - 1] = pickedCube;
255     }
256 }
```

C GameLogic.cs

Listing 3: GameLogic.cs

```
0  using UnityEngine;
1  using System.Collections;
2  
3  public enum RoundResult
4  {
5      player1Won,
6      player2Won,
7      draw
8  }
9  
10 public class gamelogic : MonoBehaviour {
11 
12     GameObject player1;
13     PlayerScript player1Script;
14     public GameObject[] player1Blocks = new GameObject[5];
15 
16     GameObject player2;
17     PlayerScript player2Script;
18     public GameObject[] player2Blocks = new GameObject[5];
19 
20     public GameObject activePlayer;
21 
22     GameObject resultText;
23     TextMesh comboText;
24 
25     int[,] winLooseMatrix = new int[5, 5] {
26             // Red, Blue, Green, Black, White
27         /*Red*/{ 0, 2, 1, 1, 2 },
28        /*Blue*/{ 1, 0, 2, 2, 1 },
29       /*Green*/{ 2, 1, 0, 1, 2 },
30       /*Black*/{ 2, 1, 2, 0, 1 },
31       /*White*/{ 1, 2, 1, 2, 0 }
```

```
32      };
33
34      public bool roundActive = true;
35      public bool gameEnded = false;
36      public RoundResult currentRoundResult;
37
38      public bool AutoMode = true;
39      public float AutoModeDelay = 0.5f;
40
41      public int MaxScore = 10;
42
43      // Use this for initialization
44      void Start () {
45
46          player1 = GameObject.Find("player1");
47          player1Script = player1.GetComponent<PlayerScript>();
48
49          player2 = GameObject.Find("player2");
50          player2Script = player2.GetComponent<PlayerScript>();
51
52          activePlayer = player1;
53
54          comboText = GameObject.Find("ComboText").GetComponent<TextMesh>();
55          resultText = GameObject.Find("ResultText");
56
57          foreach (GameObject cube in player1Blocks) {
58              cube.GetComponent<Cube>().playerThisCubeBelongsTo = player1;
59          }
60          foreach (GameObject cube in player2Blocks) {
61              cube.GetComponent<Cube>().playerThisCubeBelongsTo = player2;
62          }
63
64          GameObject.Find("ScoreNeeded").GetComponent<TextMesh>().text =
65          "Score␣needed␣to␣win:␣" + MaxScore;
66
67          if(AutoMode){
68              GameObject.Find("ResetButton").SetActiveRecursively(false);
69          }
70          gameEnded = false;
71      }
72
73
74      void Update () {
75          if(!roundActive || gameEnded)
76              return;
77
78          //let AI pick
79          if(activePlayer.GetComponent<PlayerScript>().isAI){
80              if(activePlayer == player1){
81                  player1Script.pickCubeByAI();
82
83                  player1Blocks[(int)activePlayer.GetComponent<PlayerScript>()
84                  .pickedCube].GetComponent<Cube>().gotPicked();
85                  activePlayer = player2;
```

```
86              }else if(activePlayer == player2){
87                  player2Script.pickCubeByAI();
88
89                  player2Blocks[(int)activePlayer.GetComponent<PlayerScript>()
90                  .pickedCube].GetComponent<Cube>().gotPicked();
91                  activePlayer = player1;
92              }
93          }
94
95          //check if everyone has picked, if so end the round,
96          //determine winner and reset
97          if(player1Script.hasPickedACube == true
98          && player2Script.hasPickedACube == true){
99              //determineWinner1();
100             determineWinner();
101             updatePlayerScoreTexts();
102             roundActive= false;
103             if(AutoMode){
104                 if(AutoModeDelay > 0.001)
105                     StartCoroutine( ResetAfterDelay(AutoModeDelay) );
106                 else
107                     Reset();
108             }
109         }
110
111     }
112     IEnumerator ResetAfterDelay(float delay){
113         yield return new WaitForSeconds(delay);
114         Reset();
115     }
116
117     void checkEndOfGame(){
118         if(player1Script.OverallScore >= MaxScore){
119             resultText.GetComponent<TextMesh>().text =
120             "Result:␣Player␣1␣won␣the␣game";
121             gameEnded = true;
122         }
123         else if(player2Script.OverallScore >= MaxScore){
124             gameEnded = true;
125             resultText.GetComponent<TextMesh>().text =
126             "Result:␣Player␣2␣won␣the␣game";
127         }
128
129     }
130
131     private void registerSequencesOfPlayers(){
132         //register the last actions of the other player
133         if(player1Script.isAINgram)
134             player1Script.ngramPredictor.registerSequence(player2Script
135             .previousCubePicks);
136         if(player2Script.isAINgram)
137             player2Script.ngramPredictor.registerSequence(player1Script
138             .previousCubePicks);
139     }
```

```
140     public PlayerScript returnOtherPlayer(PlayerScript callingPlayer){
141
142         if(callingPlayer == player1Script)
143             return player2Script;
144         if(callingPlayer == player2Script)
145             return player1Script;
146
147         Debug.Log("Couldn't return other player script");
148         Debug.Break();
149         return player1Script;
150     }
151
152     public void cubePicked(CubeType pickedType){
153
154         if(activePlayer == player1 && !activePlayer
155         .GetComponent<PlayerScript>().isAI){
156             Debug.Log("human player 1 picked: " + pickedType);
157             activePlayer = player2;
158             player1Script.pickedACube(pickedType);
159
160         }else if(activePlayer == player2 && !activePlayer
161         .GetComponent<PlayerScript>().isAI){
162             Debug.Log("human player 2 picked: " + pickedType);
163             activePlayer = player1;
164             player2Script.pickedACube(pickedType);
165         }
166     }
167
168     void determineBonusScore(PlayerScript givenPlayer){
169         if(player1Script.previousCube == CubeType.none
170         || player2Script.previousCube == CubeType.none){
171             //Debug.Log("no previous cube");
172             return;
173         }
174         //no bonus if round was draw
175         if(currentRoundResult == RoundResult.draw){
176             return;
177         }
178
179         switch(givenPlayer.pickedCube) {
180             case CubeType.red:
181                 switch(givenPlayer.previousCube) {
182                     case CubeType.blue:
183                         giveBonusScore(2, givenPlayer);
184                         break;
185                     case CubeType.white:
186                         giveBonusScore(1, givenPlayer);
187                         break;
188                 }
189                 break;
190
191             case CubeType.blue:
192                 switch(givenPlayer.previousCube) {
193                     case CubeType.green:
```

```
194                         giveBonusScore(2, givenPlayer);
195                         break;
196                     case CubeType.red:
197                         giveBonusScore(1, givenPlayer);
198                         break;
199                 }
200                 break;
201 
202             case CubeType.green:
203                 switch(givenPlayer.previousCube) {
204                     case CubeType.black:
205                         giveBonusScore(2, givenPlayer);
206                         break;
207                     case CubeType.blue:
208                         giveBonusScore(1, givenPlayer);
209                         break;
210                 }
211                 break;
212 
213             case CubeType.black:
214                 switch(givenPlayer.previousCube) {
215                     case CubeType.white:
216                         giveBonusScore(2, givenPlayer);
217                         break;
218                     case CubeType.green:
219                         giveBonusScore(1, givenPlayer);
220                         break;
221                 }
222                 break;
223 
224             case CubeType.white:
225                 switch(givenPlayer.previousCube) {
226                     case CubeType.red:
227                         giveBonusScore(2, givenPlayer);
228                         break;
229                     case CubeType.black:
230                         giveBonusScore(1, givenPlayer);
231                         break;
232                 }
233                 break;
234 
235             default:
236                 Debug.Log("default called, something went wrong");
237                 Debug.Break();
238                 break;
239         }
240     }
241 
242     void giveBonusScore(int bonusScoreToGive, PlayerScript toThisPlayer){
243         toThisPlayer.OverallScore +=bonusScoreToGive;
244         toThisPlayer.BonusScore +=bonusScoreToGive;
245 
246         if(bonusScoreToGive==1){
247             comboText.text = "Combo! +1 extra Score";
```

```
248         }else if(bonusScoreToGive==2){
249             comboText.text = "MEGA Combo! +2 extra Score";
250         }
251     }
252 
253     void determineWinner(){
254         if(player1Script.pickedCube == CubeType.none
255         || player2Script.pickedCube == CubeType.none){
256             Debug.Log("player picked no cube");
257             Debug.Break();
258         }
259 
260         //adapt players choise to be entered to matrix
261         int player1Choice = (int)player1Script.pickedCube;
262         int player2Choice = (int)player2Script.pickedCube;
263 
264         if(winLooseMatrix[player2Choice ,player1Choice] == 0)
265             nooneWonRound();
266         else if(winLooseMatrix[player2Choice ,player1Choice] == 1)
267             player1WonRound();
268         else if(winLooseMatrix[player2Choice ,player1Choice] == 2)
269             player2WonRound();
270     }
271 
272     void updatePlayerScoreTexts(){
273         player1Script.updateScoreText();
274         player2Script.updateScoreText();
275     }
276 
277     void player1WonRound(){
278         //Debug.Log("Player 1 won");
279         currentRoundResult = RoundResult.player1Won;
280         player1Script.roundsWon++;
281         player1Script.OverallScore++;
282         determineBonusScore(player1Script);
283 
284         resultText.GetComponent<TextMesh>().text
285         = "Result: Player 1 won. +1 Score";
286     }
287 
288     void player2WonRound(){
289         //Debug.Log("Player 2 won");
290         currentRoundResult = RoundResult.player2Won;
291         player2Script.roundsWon++;
292         player2Script.OverallScore++;
293         determineBonusScore(player2Script);
294 
295         resultText.GetComponent<TextMesh>().text
296         = "Result: Player 2 won. +1 Score";
297     }
298 
299     void nooneWonRound(){
300         //Debug.Log("Round is draw");
301         currentRoundResult = RoundResult.draw;
```

```
302             resultText.GetComponent<TextMesh>().text = "Result: Round is draw";
303         }
304 
305         public void Reset(){
306             if(!roundActive){
307                 resultText.GetComponent<TextMesh>().text = "Result:";
308 
309                 comboText.text = "";
310 
311                 //first update cubepicks, then register them
312                 player1Script.Reset();
313                 player2Script.Reset();
314                 registerSequencesOfPlayers();
315 
316                 foreach (GameObject cube in player1Blocks) {
317                     cube.GetComponent<Cube>().Reset();
318                 }
319                 foreach (GameObject cube in player2Blocks) {
320                     cube.GetComponent<Cube>().Reset();
321                 }
322                 if(player1Script.doDebugLogs || player2Script.doDebugLogs)
323                     Debug.Log("---------------NEW ROUND---------------");
324                 roundActive = true;
325                 checkEndOfGame();
326             }
327         }
328 
329 }
```

D AI vs AI test results

Table 2: N-Grams vs pattern AI

N-Grams vs pattern AI				
Games played:	100			
Round score to be reached:	250			
N-Grams games won	100			
N-Grams sum of Rounds won	15655		N-Grams average Rounds won	156,55
N-Grams sum of Bonus Score	9380		N-Grams average Bonus Score	93,8
N-Grams sum of Overall Score	25035		N-Grams average Overall Score	250,35
Pattern AI games won	0			
Pattern AI sum of Rounds won	292		Pattern AI average Rounds won	2,92
Pattern AI sum of Bonus Score	253		Pattern AI average Bonus Score	2,53
Pattern AI sum of Overall Score	545		Pattern AI average Overall Score	5,45

Table 3: N-Grams vs N-Grams AI

N-Grams vs N-Grams AI				
Games played:	100			
Round score to be reached:	250			
N-Grams 1 games won	44			
N-Grams 1 sum of Rounds won	17116		N-Grams 1 average Rounds won	171,16
N-Grams 1 sum of Bonus Score	6170		N-Grams 1 average Bonus Score	61,7
N-Grams 1 sum of Overall Score	23286		N-Grams 1 average Overall Score	232,86
N-Grams 2 games won	56			
N-Grams 2 sum of Rounds won	17420		N-Grams 2 average Rounds won	174,2
N-Grams 2 sum of Bonus Score	6159		N-Grams 2 average Bonus Score	61,59
N-Grams 2 sum of Overall Score	23579		N-Grams 2 average Overall Score	235,79

Table 4: N-Grams vs random AI

N-Grams vs random AI				
Games played:	100			
Round score to be reached:	250			
N-Grams games won	29			
N-Grams sum of Rounds won	15459		N-Grams average Rounds won	154,59
N-Grams sum of Bonus Score	7227		N-Grams average Bonus Score	72,27
N-Grams sum of Overall Score	22686		N-Grams average Overall Score	226,86
Random AI games won	71			
Random AI sum of Rounds won	15353		Random AI average Rounds won	153,53
Random AI sum of Bonus Score	9096		Random AI average Bonus Score	90,96
Random AI sum of Overall Score	24449		Random AI average Overall Score	244,49

Table 5: N-Grams Performance

N-Grams vs pattern AI performance test		
Window Size 2		
Games played:	100	
Round score to be reached:	250	
Average time to register a sequence	0.02109122	ms
Average time to make a prediction	0.02344708	ms
Window Size 20		
Games played:	1	
Round score to be reached:	10000	
Average time to register a sequence	0.0967881	ms
Average time to make a prediction	0.1035979	ms
Window Size 200		
Games played:	1	
Round score to be reached:	10000	
Average time to register a sequence	1.077298	ms
Average time to make a prediction	1.128417	ms

E Human vs AI test results

Table 6: Results of participants 1 to 6

Participant #	1	2	3	4	6
Game vs Random first	yes	no	no	no	yes
Game vs Random AI					
Time taken until Score was reached:	10:10:00	10:57:00	02:59:00	04:01:00	05:30:00
AutoMode used:	Yes, 0,5 sec	Yes, 0,5 sec	Yes, 0 sec	Yes, 0 sec	no
Player 1 rounds won:	105	121	101	63	106
Player 1 bonus score:	145	78	149	51	145
Player 1 overall score:	250	199	250	114	251
Player 2 rounds won:	115	153	117	132	94
Player 2 bonus score:	65	97	77	118	49
Player 2 overall score:	180	250	194	250	143
Game vs N-Grams AI					
Time until Score was reached:	08:45:00	53:20:00	11:53:00	10:21:00	15:40:00
AutoMode used:	Yes, 0,5 sec	Yes, 0,5 sec	Yes, 0,5 sec	Yes, 0,5 sec	no
Player 1 rounds won:	66	72	62	136	128
Player 1 bonus score:	46	60	64	69	70
Player 1 overall score:	112	132	126	205	198
Player 2 rounds won:	159	141	135	165	160
Player 2 bonus score:	91	110	116	87	91
Player 2 overall score:	250	251	251	252	251
Which AI did was better?	Draw	Draw	N-Grams	Random	N-Grams

Table 7: Results of participants 7 to 11

Participant #	7	8	9	10	11
Game vs Random first	yes	yes	no	yes	no
Game vs Random AI					
Time until Score was reached:	11:25:00	11:40:00	04:30:00	09:45:00	04:00:00
AutoMode used:	Yes, 0	Yes, 0,5 sec	Yes, 0	Yes, 0,5 sec	Yes, 0,5 sec
Player 1 rounds won:	133	95	151	144	160
Player 1 bonus score:	117	156	77	108	92
Player 1 overall score:	250	251	228	252	252
Player 2 rounds won:	134	93	155	123	141
Player 2 bonus score:	70	68	97	79	91
Player 2 overall score:	204	161	252	202	232
Game vs N-Grams AI					
Time until Score was reached:	02:38:00	13:30:00	06:58:00	06:09:00	14:10:00
AutoMode used:	Yes, 0	Yes, 0,5 sec	Yes, 0	Yes, 0,5 sec	Yes,1,5 & 0,5
Player 1 rounds won:	126	103	112	116	98
Player 1 bonus score:	85	81	55	84	61
Player 1 overall score:	211	184	167	200	159
Player 2 rounds won:	173	150	163	145	153
Player 2 bonus score:	77	100	89	105	97
Player 2 overall score:	250	250	252	250	250
Which AI did was better?	N-Grams	N-Grams	Draw	N-Grams	N-Grams

Table 8: Results of participants 12 to 18

Participant #	12	13	14	15	17	18
Game vs Random first	no	yes	yes	no	no	yes
Game vs Random AI						
Time until Score was reached:	08:46:00	10:57:00	13:30:00	12:15:00	01:20:00	07:20:00
AutoMode used:	Yes, 0,5 sec	Yes, 0	Yes, 0,5 sec	Yes, 0	Yes, 0	Yes, 0
Player 1 rounds won:	97	191	127	147	144	141
Player 1 bonus score:	155	35	124	104	106	110
Player 1 overall score:	252	226	251	251	250	251
Player 2 rounds won:	83	158	105	156	147	139
Player 2 bonus score:	43	92	51	92	89	78
Player 2 overall score:	126	250	156	248	236	217
Game vs N-Grams AI						
Time until Score was reached:	19:58:00	01:11:00	07:10:00	06:30:00	06:35:00	07:30:00
AutoMode used:	Yes,0 & 1,5	Yes, 0	Yes, 0,5 sec	Yes,0 & 0,5	Yes, 0	Yes,0 & 0,5
Player 1 rounds won:	68	21	95	100	91	124
Player 1 bonus score:	81	18	48	55	78	101
Player 1 overall score:	119	39	143	155	169	225
Player 2 rounds won:	172	161	146	180	154	165
Player 2 bonus score:	78	89	105	70	98	85
Player 2 overall score:	250	250	251	250	252	250
Which AI did was better?	Random	N-Grams	Draw	N-Grams	N-Grams	N-Grams

Table 9: Average results of all participants

Average of all Participants	
Game vs Random AI	
Average time until Score was reached:	08:04:03,75
Player 1 rounds won:	126,63
Player 1 bonus score:	109,5
Player 1 overall score:	236,13
Player 2 rounds won:	127,81
Player 2 bonus score:	78,5
Player 2 overall score:	206,31
Game vs N-Grams AI	
Average time until Score was reached:	12:01:07,50
Player 1 rounds won:	94,88
Player 1 bonus score:	66
Player 1 overall score:	159
Player 2 rounds won:	157,63
Player 2 bonus score:	93
Player 2 overall score:	250,63

Table 10: Miscellaneous results from the Human vs AI tests

Game vs Random first	
Yes	8
No	8
Participants	16
Wins vs Random	12
Loss vs Random	4
Wins vs N-Grams	0
Loss vs N-Grams	16
Which AI was better?	
Draw	4
Random	2
N-Grams	10

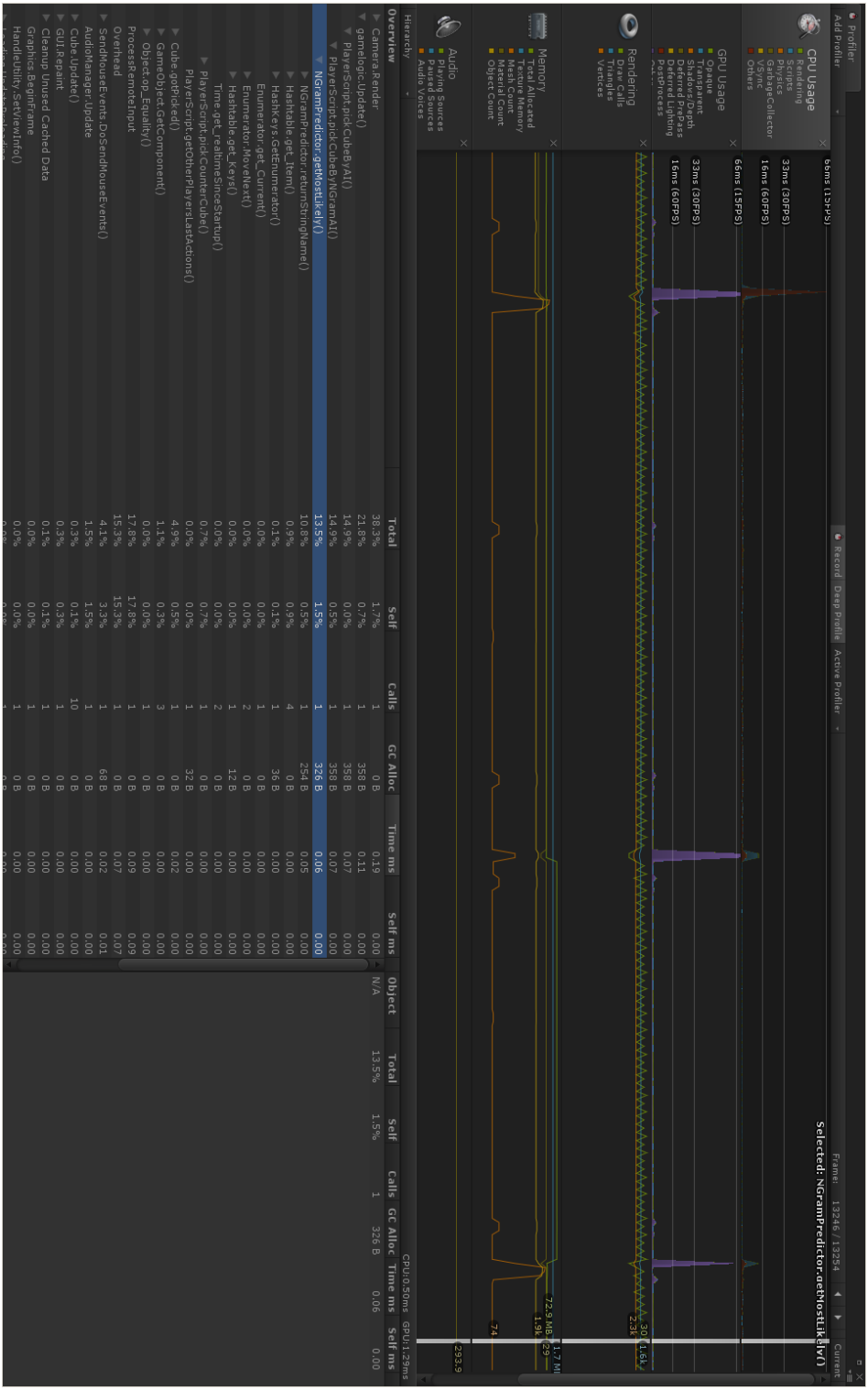

Figure 6: Unity Profiler of a frame where the getMostLikely function was called

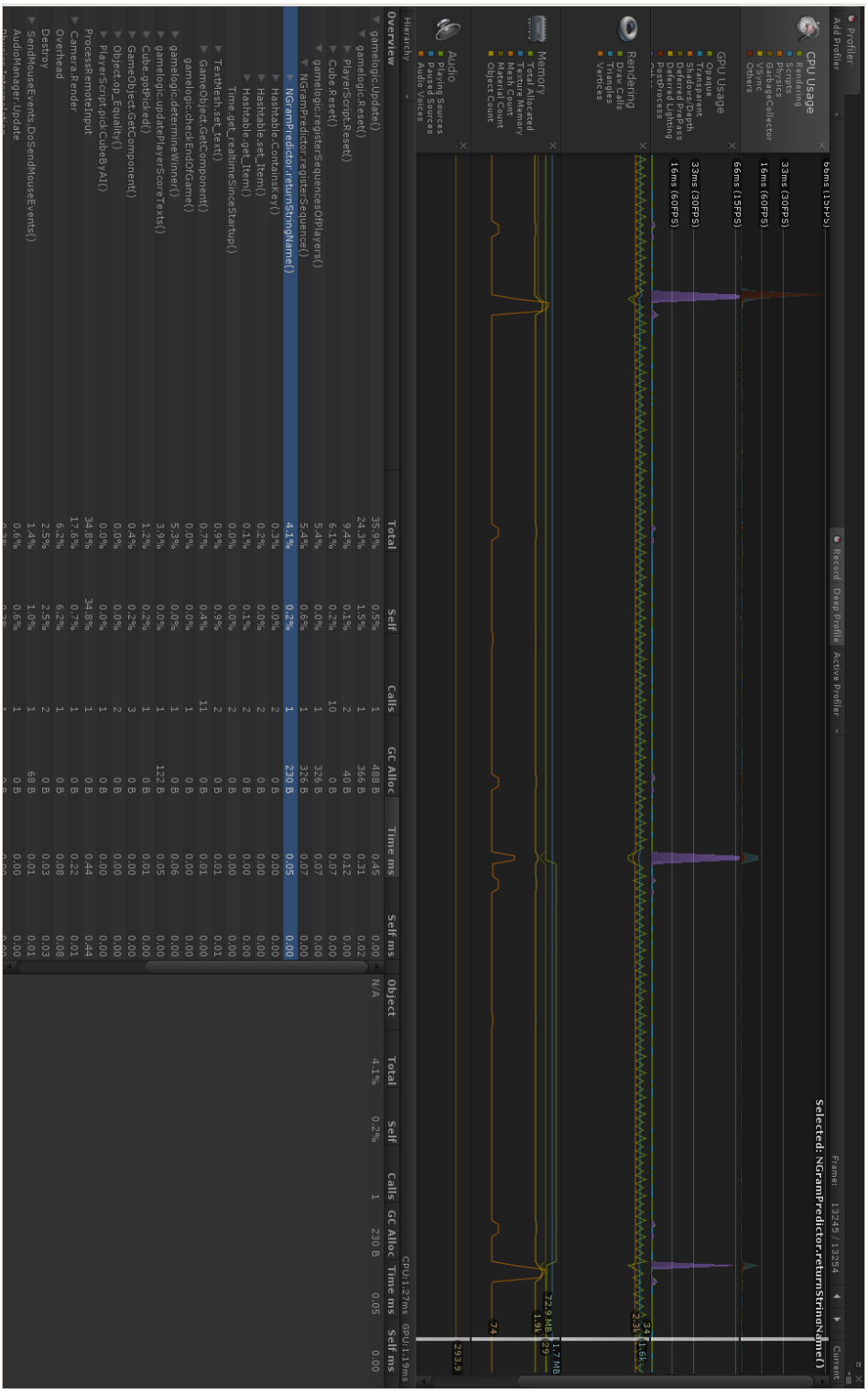

Figure 7: Unity Profiler of a frame where the registerActions function was called

References

Brown, Michael K., Andreas Kellner, and Dave Raggett. 2001. *Stochastic language models (N-Gram) specification.* W3C Working Draft. `http://www.w3.org/TR/ngram-spec/`.

Cavnar, William B., and Alan J. Vayda. 1992. Using superimposed coding of n-gram lists for efficient inexact matching. In *Proceedings of the fifth usps advanced technology conference, washington d.c.* Washington D.C, USA.

Franz, Alex, and Thorsten Brants. 2006. All our n-gram are belong to you. Accessed: 28/03/2012. Aug. `http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html`.

Ganapathiraju, M., D. Weisser, R. Rosenfeld, J. Carbonell, R. Reddy, and J. Klein-Seetharaman. 2002. Comparative n-gram analysis of whole-genome protein sequences. In *Proceedings of the second international conference on human language technology research,* 76–81. HLT '02. San Diego, California: Morgan Kaufmann Publishers Inc.

Griffiths, Thomas L., and Joshua B. Tenenbaum. 2001. *Randomness and coincidences: reconciling intuition and probability theory.* Stanford, CA 94305-2130 US.

Jon, Orwant. 2010. Find out what's in a word, or five, with the google books ngram viewer. Accessed: 28/03/2012. Dec. `http://googleblog.blogspot.com/2010/12/find-out-whats-in-word-or-five-with.html`.

Jurafsky, Daniel, and James H. Martin. 2008. *Speech and language processing (prentice hall series in artificial intelligence).* 2nd ed. Prentice Hall.

Keselj, Vlado, Fuchun Peng, Nick Cercone, and Calvin Thomas. 2003. *N-gram-based author profiles for authorship attribution.* Dalhousie University, Canada.

Kukich, Karen. 1992. Techniques for automatically correcting words in text. *ACM Comput. Surv.* 24, no. 4 (Dec.): 377–439. ISSN: 0360-0300.

Laramée, François Dominic. 2002. Using n-gram statistical models to predict player behavior. *AI Game Programming Wisdom* (Rockland, MA, USA):596–601.

Mantegna, R. N., S. V. Buldyrev, A. L. Goldberger, S. Havlin, C. K. Peng, M. Simons, and H. E. Stanley. 1995. Systematic analysis of coding and noncoding DNA sequences using methods of statistical linguistics. *Physical Review E* (Boston University, USA) 52:2939–2950.

Mariòo, José B., Rafael E. Banchs, Josep M. Crego, Adrià de Gispert, Patrik Lambert, José A. R. Fonollosa, and Marta R. Costa-jussà. 2006. N-gram-based machine translation. *Comput. Linguist.* (Boston, USA) 32, no. 4 (Dec.): 527–549. ISSN: 0891-2017.

Millington, Ian. 2006. *Artificial intelligence for games (the morgan kaufmann series in interactive 3d technology).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Ponsen, Marc, and Ir. P. H. M. Spronck. 2004. Improving adaptive game ai with evolutionary learning. In *University of wolverhampton,* 389–396.

Unity, Technologies. 2012. Unity - game engine. Accessed: 03/04/2012. Apr. `http://unity3d.com/`.

Wang, Kuansan, Christopher Thrasher, Evelyne Viegas, Xiaolong Li, and Bo-june (Paul) Hsu. 2010. An overview of microsoft web n-gram corpus and applications. In *Proceedings of the naacl hlt 2010 demonstration session,* 45–48. HLT-DEMO '10. Los Angeles, California: Association for Computational Linguistics.