Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# A computer game with computer-assisted human players

Bachelor thesis

2018
Matej Králik

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

# A computer game with computer-assisted human players

Bachelor thesis

| | |
|---|---|
| Study program: | Informatics |
| Field of study: | Informatics |
| Department: | Department of Informatics |
| Supervisor: | Dominik Csiba, Bc. |

Bratislava, 2018

Matej Králik

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

## ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Matej Králik
**Študijný program:** informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
**Študijný odbor:** informatika
**Typ záverečnej práce:** bakalárska
**Jazyk záverečnej práce:** anglický
**Sekundárny jazyk:** slovenský

**Názov:** A computer game with computer-assisted human players
*Počítačová hra s ľudským hráčom a asistujúcim programom*

**Anotácia:** Cieľom práce je preskúmať potenciál počítačových hier hraných ľuďmi s asistujúcim programom. Hráčom je v takýchto hrách umožnené spolupracovať s programom, ktorý napísali, alebo len upravili a rozširuje ich herné možnosti. Hlavnou výzvou práce je otestovanie konceptu tohto druhu hier, prostredníctvom vývoja a implementácie jednoduchej hry s API umožňujúcim programu asistovať hráčovi.
Navyše, plánujeme hru otestovať na skupine dobrovoľníkov s cieľom zbierania dát o hre, ktoré použijeme na overenie použiteľnosti nášho konceptu.

**Vedúci:** Bc. Dominik Csiba
**Katedra:** FMFI.KI - Katedra informatiky
**Vedúci katedry:** prof. RNDr. Martin Škoviera, PhD.

**Dátum zadania:** 30.10.2017

**Dátum schválenia:** 30.10.2017

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.......................................................
študent

.......................................................
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

## THESIS ASSIGNMENT

**Name and Surname:** Matej Králik
**Study programme:** Computer Science (Single degree study, bachelor I. deg., full time form)
**Field of Study:** Computer Science, Informatics
**Type of Thesis:** Bachelor´s thesis
**Language of Thesis:** English
**Secondary language:** Slovak

**Title:** A computer game with computer-assisted human players

**Annotation:** The goal of this thesis is to explore the concept of computer-assisted games. Players of such games are allowed to use a program they wrote or edited to alter their game experience.
The main challenge is to test the proof-of-concept of such games by developing and implementing a simple game with an API allowing computer assistance for the human player.
Additionally, we plan to test the game on a group of volunteers to collect data about the game in general, which we will use to test the feasibility of our concept.

**Supervisor:** Bc. Dominik Csiba
**Department:** FMFI.KI - Department of Computer Science
**Head of department:** prof. RNDr. Martin Škoviera, PhD.

**Assigned:** 30.10.2017

**Approved:** 30.10.2017 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.................................................. ..................................................
Student Supervisor

**Acknowledgment:** I would like to thank Dominik for the continuous guidance, B. Brejová for the consultations, Bui for design ideas, Sára for the support, and the 36 participants of the Spring Camp 2018 of Correspondence Seminar in Programming for spending their day with this game.

# Abstrakt

Na prelome herného priemyslu a akademickej sféry sú často porovnávané schopnosti ľudských hráčov a umelých inteligencií. Ľudia, rovnako ako umelé inteligencie majú svoje silné aj slabé stránky. Ak by ľudský hráč spolupracoval s umelou inteligenciou, prepojením ich silných stránok by mohlo dôjsť k zvýšeniu herného výkonu. Na základe historických poznatkov môžeme konštatovať, že výhoda ľudských hráčov spočíva v ich intuícii, zatiaľ čo umelá inteligencia vie využiť hrubú výpočtovú silu. Cieľom tejto práce je navrhnúť hru určenú na hranie pre človeka spolupracujúceho s programom - prípadnou umelou inteligenciou. Tento program bude mať prístup k rovnakým údajom ako ľudský hráč a bude schopný ovplyvniť to, čo hráč vidí na obrazovke. Následne po návrhu hry, popíšeme jej implementáciu. Na záver hru otestujeme počas jarného sústredenia pre stredoškolákov so záujmom o programovanie. Očakávaný výsledok testovania, je zaznamenať momenty kedy výstup z programu ovplyvnil priebeh hry, rovnako ako momenty kedy ľudská intuícia mala navrch.

**Kľúčové slová:** hry, herný dizajn, umelá inteligencia

# Abstract

At the intersection of the game industry and academia, the capabilities of a human and an artificial intelligence (AI) are often compared. Both humans and AIs have their strengths and weaknesses. If humans and AIs to cooperate, their performance in games may be even better. Based on historical examples, the strength of human players is their intuition, while AIs excel at using their raw computing power. The goal of this thesis is to design a game dedicated to be played by a human cooperating with a program - possibly an AI. This program will have access to the same data as the human player and will be able to modify what a player can see in the game. After designing the game, we will present our implementation. Finally, we will test the implemented game during a spring camp for high school students with interest in programming. The desired result of the testing is to capture moments where the output of the program changed the course of the game, as well as moments where the human intuition came out on top.

**Keywords:** games, game design, artificial intelligence

# Contents

**Introduction** **1**

**1 Human-computer rivality** **3**

1.1 Historical overview of human battle against AI . . . 3

1.1.1 Chess . . . 3

1.1.2 Go . . . 4

1.1.3 Other games where AIs outperform humans . . . 4

1.1.4 Games that were solved . . . 6

1.1.5 Starcraft 2 . . . 6

1.1.6 Other undefeated games . . . 8

1.2 Strengths and weaknesses . . . 9

**2 Design of our game** **12**

2.1 Design requirements . . . 12

2.2 Description of the rules . . . 14

2.3 Key elements in the rules . . . 20

2.4 Weak points of our design . . . 22

**3 Implementation of our game** **23**

3.1 Implementation requirements . . . 23

3.2 Architecture overview . . . 24

3.2.1 Game server . . . 25

3.2.2 Game client . . . 26

3.2.3 Assisting program . . . 26

3.3 Example assisting programs . . . 27

3.3.1 Simple game counters . . . 29

3.3.2 Previous enemy positions . . . 30

3.3.3 Least dangerous positions heuristic . . . 31

**4 Final testing and results** **34**

4.1 Testing environment . . . 35

4.1.1 Testing schedule . . . . . . . . . . 35
4.2 Results . . . . . . . . . . 36
4.2.1 Testing walk-through . . . . . . . . . . 36
4.2.2 Survey responses . . . . . . . . . . 38
4.2.3 Testing takeaways . . . . . . . . . . 39

**Conclusion** **40**

# List of Figures


2.1 Example of a game situation . . . . . . . . . . . . . . . . . . . . . . . . 14
2.2 Spawn squares and generated times . . . . . . . . . . . . . . . . . . . . . 15
2.3 Created units and their properties . . . . . . . . . . . . . . . . . . . . . 15
2.4 Fog of war around a unit . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.5 Different terrain types . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.6 Board reduction example . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.7 Shooting lasers on enemy units . . . . . . . . . . . . . . . . . . . . . . . 19

3.1 Example usage of an assisting program . . . . . . . . . . . . . . . . . . . 27
3.2 Simple game counters using the assisting program . . . . . . . . . . . . . 30
3.3 Previous enemy positions using the assisting program . . . . . . . . . . . 31
3.4 Least dangerous positions heuristic using the assisting program . . . . . 33

# List of Tables

2.1 All the terrain types present on the board . . . . . . . . . . . . . . . . . . 16
2.2 Key elements in the rules addressing requirements . . . . . . . . . . . . . 20

# Introduction

During the last century the evolution in computing power changed our lives tremendously. We changed the way we think about mathematical and logical challenges we face. One group of those challenges are games, without closer specification - computer games, table games, real-life role-playing games, and so on.

As mankind we naturally incline towards measuring our performance, and more importantly, we compare it to that of other individuals. With the development in the area of computing we began challenging ourselves by our very own devices. We even started to teach those same devices different strategies and observed how they performed. This led to establishing a widely used informal term - Artificial Intelligence (AI). The term stands for device that was taught (programmed) to imitate intelligence as we humans understand it.

Accordingly, we can distinguish multiple types of games, based on the player type, of which the most common are:

- Human vs. Human - the traditional type of a game. Two individuals measure their ability to find and execute a strategy given a certain set of rules to follow.
- AI vs. AI - type of a game dedicated to measuring the performance and the ability to reach a goal under certain limitations of a computer program, which is usually written by a human.
- Human vs. AI - comparison of the two worlds, where we try to recognize which of the players is stronger - the human or a program written by a human.

The motivation behind our thesis lies in the underlying problem in these types of games. The computer and a human brain are both very capable, but yet we use only one of them, or rather use them against each other. As we will show later, the key strengths and the critical weaknesses on both sides are distinct, thus leading us to an assumption.

*„If both of our resources are strong in different areas, then when used in combination, their performance should vastly surpass their individual capabilities."*

An uncommon type of game which will be our point of interest therefore is:

- (Human + AI) vs. (Human + AI) - when the human interacts and cooperates with a written program in order to find an optimal strategy for the game. Together they control one of the players and try to take an advantage thanks to their individual strengths.

In the next chapter we will briefly describe the history of games where humans and AIs battled against each other. In the second chapter we put down design requirements and the rules of our very own game. This game is of the type we mentioned above and designed in a way that it should be challenging sufficiently for a human and an AI as well. The third chapter provides a description of the implementation of our game. In the last chapter we specify the methods we used to test our game on the Spring Camp 2018 of Correspondence Seminar in Programming.

# Chapter 1

# Human-computer rivality

Understanding the historical point of view is a prerequisite for preparing the ground for designing our very own game in the next chapter. The following section contains examples of games where AIs did beat humans as well as examples where the opposite is true. At the end of the chapter we summarize the various observations from the historical examples.

## 1.1 Historical overview of human battle against AI

In order to properly design a game of the mentioned kind we first have to understand how far can we go as humans and how smart are the programs we are creating. To understand our limits, as well as the limits of our computers, let us take a quick look back through history.

From the human perspective, one of the most known board games are chess, go, backgammon, poker, and a few others. All of these games have a professional scene with people dedicating their whole lives only to become the best players in the world. Therefore it is no surprise that these games have been used to measure the performance of humans and AIs since the birth of this concept.

### 1.1.1 Chess

Undoubtedly Magnus Carlsen can be considered to be one of the greatest players in chess history. He is the current holder of the World Champion title in chess, champion in World Rapid & Blitz Chess Championship more than once and he earned his grandmaster title(currently the highest title a player can get) at the age of 13. In an interview [7] taken in 2009 for the magazine TIME, when asked how many moves he calculates ahead he answered:

"Sometimes 15 to 20 moves ahead. But the trick is evaluating the position

at the end of those calculations." -Magnus Carlsen

Thinking that much moves ahead means considering at least hundred different resulting positions in your head. That number may sound like a lot for a computer, but with the human brain it is different. The human brain can very easily rule out all the impossible moves. Afterwards it can use a few simple heuristics to focus only on those positions which make at least a little sense in the traditional understanding of the game. For a computer, to think 15 to 20 moves ahead is a lot more work.

A famous historical example of a Human vs. AI match goes back in time more than 20 years. In 1997 a match between Garry Kasparov and IBM's Deep Blue took place. Garry Kasparov was the world champion during that time and one of the best players in the world during the following years as well. A chess-playing computer - IBM's Deep Blue was the result of over 11 years of development at IBM. Deep Blue defeated Kasparov $3\frac{1}{2} : 2\frac{1}{2}$, making it the first AI to be better than world's best human player in a well known game. Even though Deep Blue was never open sourced and IBM was even accused of cheating, we can still make a brief comparison. It claimed to process about 200 million possible positions every second and still search only to a similar depth than the one mentioned by Carlsen.

### 1.1.2 Go

The game of Go is accompanied with a similar story but in a lot more open and less controversial version. Google's Deepmind developed a computer program called AlphaGo [15], which was the first program to defeat a Go world champion and arguably the strongest Go player in history. In March 2016, AlphaGo competed against the winner of 18 world titles and considerably the greatest go player - Lee Sedol. This time, the match was a worldwide event with over 200 million viewers and resulted AlphaGo's victory $4 - 1$.

But the interesting part of the AlphaGo project did not stop there. AlphaGo was trained using thousands of amateur and professional games, leading to the next challenge for it's creators. A year later, another paper was released about AlphaGo Zero [16]. AlphaGo Zero was an upgraded version of AlphaGo, which instead of learning from human games, learned only from playing against itself. This step made it surpass the performance of all previous versions, including those which defeated Lee Sedol.

### 1.1.3 Other games where AIs outperform humans

Chess and go were only the most famous examples when a computer program outplayed humans in a game. There are numerous examples of this happening among other games as well. The problem with lesser known games is often the lack of acknowledgment.

This results in the lack of testing of such programs and their performance. In these games, we may only wonder, whether these programs could really outperform the best human players.

More than two decades ago, the game of **backgammon** was challenged by AIs. The zero knowledge approach (similar to the one used by AlphaGo) turned out to be very successful and the level of play of our programs rivaled that of the best humans [20] [19]. The interesting difference between Backgammon and chess or go is that backgammon introduces a random factor to the game. Throwing dice creates a great amount of possible states and an unpredictable element of the game. Despite this difference, AIs were successful in defeating humans.

Another of such games is **poker** [4]. The distinguishing part of poker against previously mentioned games is that the player has only imperfect information, since the other player's cards or the cards in the deck are not known. This complication requires a rather different approach from the programs. Poker has plenty of different game variants, the simpler of which are solved by computers, whereas the most complicated ones are not. In one of the most known poker variants - No-limit Texas Hold'em, AIs are on top of humans recently [5].

Imperfect information is also present in **Scrabble**, a game where the goal is to score points by placing tiles with letters onto a board. The amount of possible turns in a position is greater than in chess, yet AI did take the trophy from humans in this game as well more than a decade ago [13]. Back then, the most known Scrabble AI Maven was able to beat the best human player about two thirds of the time. One could argue that this is not much of an achievement, because a computer can easily remember a whole dictionary of words and abuse that in the game. But the game of scrabble is not only about words from a dictionary. The players interact with the board and the count of different letters in the game is limited, therefore a lot of other different factors need to be taken into account.

A very interesting game in this sense is **Arimaa**[1] [17] which was particularly designed to be hard for computers. Rules of Arimaa are very simple and it can be played with a traditional set of chess figures. Randomness or imperfect information are not present in this game as in the previously mentioned ones. The complexity is being achieved rather through a non-trivial amount of starting positions and complex turn possibilities. At the beginning of the game, each player can arrange his 16 pieces in any way he likes on two rows of the board. In each turn a player can make up to four moves with his pieces. The amount of unique possible moves in some positions can even be as high as tens of thousands. In 2003 even a competition with a prize was set out. The first AI to defeat humans shall win the prize. In 2015, after twelve years

[1] Rules of Arimaa available at `http://arimaa.com/arimaa/`

on of unsuccessful results, a computer program called SHARP has finally defeated the best human players [23].

### 1.1.4 Games that were solved

The games we mentioned in previous subsections have in common that the computer AIs are better than humans on average. In these games even a better player can sometimes lose to a beginner due to random factors of the game or due to some kind of luck of the beginner. But there are games where this is simply not true. If a game does not have too many total positions or possible moves in every position it is possible to simply check all of them for a winning strategy. Some games are solved in a way that from any given position we can tell who will win and some games are solved only for the starting positions.

An example of a very famous game solved for a starting position is **checkers**. Checkers has been solved 2007 and if both players play perfectly the game leads to a draw [12].

Another game of this type is **Connect-Four**[2]. The game has many variations like poker and even little kids can understand it's rules. On small boards, the game presents an unsurprisingly small amount of positions. The more interesting variation if it is played on an infinite board, where the number of possible positions are infinite. In 2012 it was shown that both players have never-losing strategies, meaning that if both players play perfectly the game is a draw as well [24].

Some solved games present a winning strategy for one of the players. **Pentago**[3] is a particularly challenging game for the human brain. Humans tend to picture the possible future positions in their minds, therefore moving and rotating parts of the board are a great challenge for sure. In 2014 it was shown that the first player has a winning strategy [9].

Qubic, Gomoku (also known as five-in-a-row) and many others have been solved or partially solved as well. Several publications [1] [21] offer a comprehensive survey in this area.

### 1.1.5 Starcraft 2

The situation is rather unfortunate for mankind in all the examples listed above. However, starting from this example the tables are turned. Starcraft 2 differs from the

[2]Rules of Connect-Four available at `https://www.hasbro.com/common/instruct/ConnectFour.PDF`

[3]Rules of Pentago available at `https://webdav.info.ucl.ac.be/webdav/ingi2261/ProblemSet3/PentagoRulesStrategy.pdf`

previous games dramatically. For starters, it is a computer game. In 2018, it is considered to be the next target for machine learning projects like AlphaGo, with a goal to build a bot better than any human player.

In Starcraft, players build buildings and create units of their own, which they control. They use these units to destroy other players, units, and buildings to win the game. The game has plenty of game modes, but we will focus on the one-on-one format. As with Go, Google's Deepmind published a collaboration with Blizzard(company that created Starcraft 2) [22] which provides a framework for bots as well as explains the key challenges for computer programs in this game. As stated in the article, current best AIs for Starcraft can be defeated on a regular basis by amateur players.

An interesting measurement in this game is Actions Per Minute (APM). APM denotes the average number of actions (unit or building selections, movement commands, ...) the player makes during a minute of the game. Typical APM of a beginner player is around a hundred, while professional players reach APM between 300 and 500.

Let us try to sum up some of the key differences between Starcraft 2 and the other games we mentioned. Some of these differences are only of technical nature and some of them are the reason for the game being a challenge for researchers. Differences:

- Players are provided only with partial(imperfect) information about the map.
- The amount of possible states is a higher than in previous examples. The player can build tens of buildings and control hundreds of units which can take thousands of steps.
- As the units move almost continuously, the number of turns in a typical game is orders of magnitude higher than for turn-based games.
- Starcraft is a real-time game so the time to decide on a turn may be shorter (fraction of a second compared to minutes in other games).

However we should also note that the game of Starcraft 2 is similar to chess or go in several aspects:

- The short term rewards are not important, only the result of the game matters. Simply put, sacrifices of a piece on a chess board or a unit on a map may lead to significant advantages later in the game.
- Most of the positions are unreachable or information about them does not add value to our strategy. This is where humans seem to be better, at evaluating what are the interesting turns and positions to think about.
- The best humans are trained professionally, several hours a day. This can be considered the best we can reach with current training techniques.

### 1.1.6 Other undefeated games

We presented a few examples of games where a program performs better than a human and a single game where humans still hold the victory. Truth is, there are several disciplines, which can be considered games, where humans have significant advantages over computers. We just did not consider wide enough range of what can be a game yet. Here we provide a short list of such challenges.

- Bridge is one of well known card games, where artificial intelligence has not yet outperformed human players. The need to cooperate while having access to only imperfect information is what makes bridge so difficult. Multiple approaches are being tested against humans [3] [25].

- Certain computer games involve a lot more than making turns and moving pieces. First person shooters like Counter Strike[4] often involve certain level of team play. A perfect aim may be an easy job for an AI, but top players execute team strategies that can overwhelm individual performance.

  Another non-trivial element of computer games can be their narratives. The player is often expected to make a choice or solve a puzzle, based on his understanding of the environment's narrative. Therefore game like World of Warcraft[5] or The Elder Scrolls[6] are unchallenged by AIs.

- The human language still is a great challenge for programmers and AIs. Translation, correction or synonym seeking are very simple tasks that today's AIs complete with decent quality. Yet, if we were to linguistically conceal a word, we would effectively create a riddle. And no other game resembles riddles more than a crossword puzzle. Crossword puzzles are still nowhere closed to be solved. Attempts to change that can be seen in publications [8].

- Mao is what we could call an extreme game [6]. Beginner players are forbidden to be told the rules and have to guess them by trial and error. During the game, additional custom rules can emerge from the players. Every match can be very different. The game is full of imperfect information. The players need to successfully guess, cooperate, deceive, and observe to win a round. The game can be as complex or as simple as the players make it, hence it may take a very long time before an AI will beat the human players.

- Identifying what is on a picture. Even though computer programs are able to distinguish between a few basic types of images or can name a few objects in a

[4]Official website of Counter Strike - http://www.counter-strike.net

[5]Official website of World of Warcraft - https://worldofwarcraft.com/

[6]Official website of The Elder Scrolls - https://elderscrolls.bethesda.net/

picture, the human brain of a child can still do this better. Reading a text is a problem of a similar kind.

- Going even further, most of the interaction with physical world is still in it's infancy for computers. Although these examples are not what we traditionally perceive as games, we can consider them like ones. Driving a car, cooking a meal, cleaning a room, changing a diaper, or a collection of other often crucial or life endangering tasks still cannot be entrusted to a computer. These tasks simply involve too much complexity, require a lot of sensors and have close to infinitely many possible states. Robots can usually only repeat simple tasks in a factory. More recently, they are also able to solve bin picking tasks using a lot of sensors.

## 1.2 Strengths and weaknesses

The purpose of this section is to summarize the findings from the previous section full of historical examples. We listed several games where AIs outperform humans as well as plenty of those where humans are still on top. The observations below are mostly linked to a few crucial terms, we are about to define. These terms are inspired by those defined in Chapter 5 of [11]. It's chapter 5 is mostly focused on games and a lot of it's concepts are very helpful in formalizing the term game and other related terms.

**State space** of a game is the set of all possible positions in a game. It may involve the positions of pieces on a board, units on a map, some partial score or information from previous turns that has impact on the rest of the game based on the rules.

**Move** is a transition from one state to another. A move has to be valid according to the game rules.

**Game tree** is a tree where the nodes are the possible states of the game and the edges between them are represented by moves. The size, depth, branching factor or any other distinguishing properties of a game tree are often a consequence of the rules.

Here is a list of the strengths of an AI that come as obvious from the examples listed above:

- If the state space is too small, AI can explore most or even all of it. In this case it can easily happen that the game will be solved by a computer using a brute force search. The computer program can then find exact strategy (either winning or never-losing) without even requiring any advanced AI techniques.

- If there is little to no variance in starting positions, by observing thousands or more games an AI is able to look up certain opening techniques and strategies, which are more likely to result in a winning position. The same is true for the end of a game. Game endings can be usually classified into categories, with similar

strategies when the amount of possibilities is small enough. A computer can learn a single strategy for each of these categories. For example, in chess, we call a list of these strategies tablebases[7].

Creating a set of opening or closing short-term strategies is possible by observing a set of provided games. The same is true in the case of a zero human knowledge AI, learning only by playing against different versions of itself.

Those were the strengths of an AI, so let us turn the table and take a look at the weaknesses. We could say that the key advantage of a human being against AI is his intuition. Human intuition can of course be misleading or very wrong, but in most cases it provides a healthy overview of good and bad game states or moves. Even in complex games, like Starcraft a human can tell in a blink of an eye if the game situation is very imbalanced for one of the players. These simple evaluations are what drives our intuitive approach.

A more experienced professional player relies less on the simple evaluations and rather observes a lot more parameters of the current game state to make better decisions. An AI behaves in a similar way. If it is told, or has learned what are the important parameters to observe on the current game state and how to react to them, it can easily outperform a human professional.

In other words when it is unclear what are the important pieces of information or a general lack of information is present, an intuitive human approach leads to plausible results. When the opposite is true and all the information has a clear structure, an AI can use raw computing power for better results.

Here is a list of a few examples when the raw computing power is less effective. We could call these the strengths of humans.

- A certain amount of randomness requires preparedness for several situations, and the required computing power can be orders of magnitude higher. This does not necessarily imply that an intuitive approach is better, it rather remains unaffected by this phenomenon.

  From a long-term perspective proper randomness should not affect the game excessively. But in the way it is used in traditional games, it results in spikes of advantage which are short-term with long-term outcomes.

  Forms of presence in games: throwing dice, shuffled deck, generated maps, critical hits, ...

- Imperfect information about the game state is similar to randomness in a way that it creates multiple magnitudes of possibilities. But conversely it is possible

[7] Tablebases for up to seven pieces can he found at `http://tb7.chessok.com`

to make certain predictions about the information we do not know. These predictions may come from previous observations, assumptions of certain play style of the other player or just the premise of a worst case scenario.

Forms of presence in games: hidden cards on other player's hand, fog of war, partial information on enemy pieces, ...

- In certain games, the starting conditions of each player may be different. In a very light form we see this even in games as chess and go, where one of the players takes the first turn, while the other starts with the second turn. But this aspect can be taken to a next level when a player can choose a fraction of his own (strategy games), invest some points into a talent tree (role-playing games) or choose his initial position from a given set.

  This starting position variation enables players to pick up a playstyle of their own and come up with a plenty of different strategies.

- Human beings are of deceitful nature which is often reflected in their playstyle. In games where deception, deceit and bluff are incorporated, an AI may have to go through a lot of trouble to predict a player. In these cases it can be very unclear what is the best play to execute. That is again, where the human intuition kicks in and may result in better than random gameplay.

  Forms of deception are present in games like poker, bridge, mao, Starcraft 2 or rarely even in chess in several different forms. Player may try to trick the other player into making a wrong move by persuading him about what is the information he knows. Making a bet, playing a card, moving a unit or taking a turn in a way that seems like a mistake at first glance is often abused by human players to deceive their opponents.

Whether a game contains more elements from the first or the second list presented above often decides if humans or computers hold the trophy. Nevertheless the amount of years of human expertise or the amount of available computer resources can shift the result one way or another.

# Chapter 2

# Design of our game

In this chapter, we will describe the design requirements for our game, it's rules, and how the requirements were projected into the rules of the game.

Firstly, we require the design to preserve the nature of the game itself. The rules of the game should be as simple as possible, with a clear goal. One match should not take too long to play and it should be easy to visualize all the game positions.

Secondly, our design should build on top of what we described in the previous chapter. As a result, our game will be challenging for a human, for an AI, and for a cooperation of both. We describe these requirements in detail in the next section.

At the end of this chapter, we will look at the designed rules of our game from a critical point of view and list a few problems that arise. These problems are not crucial in order to achieve the goal of this thesis, yet it is helpful to mention a few places where improvements could be made in case of a different design.

## 2.1 Design requirements

As described in the previous chapter, **(Human + AI) vs. (Human + AI)** is the type of game we are designing. Each player interacts and gets help from an assisting program of his own before taking a turn. Both the human and the AI can execute the same set of actions, but the player should determine which resources to use for each decision. In order to make the game challenging for the cooperation of a human and an AI, it has to meet certain requirements, which we will try to formulate based on the conclusions we stated in the previous chapter.

For the game to be challenging for a human, not much is required. It should just present enough possibilities and states with an unclear winner and it should not resemble any process that humans meet in their daily lives. In the end, if a human can fully understand and solve a game, it should not be much of a problem to program a computer to execute a similar process. Exceptions to this rule are the simple tasks

like recognizing what is on a picture, which were mentioned in the previous chapter, however those break our requirements of no daily life resemblance we just stated.

Therefore, the interesting part of the design is to introduce a certain level of complexity to the game in the interest of making it difficult enough for an AI, while still allowing a human player to make a difference. More specifically, our game should meet these requirements:

- The state space has to be big enough to make it very hard to do a brute force search.
- There should be a plenty of different starting configurations, from which each player may choose one, altering the course of the rest of the game.
- Significant amount of imperfect information needs to be present in order to engage the power of a human player.

During the design we also need to keep in mind a few other important points:

- The rules of the game should be intuitively understandable and easily implementable on a computer. The former is a good prerequisite for a human to play well and the latter is required for contributions from the AI.
- The rules of the game should contain as few special cases as possible. They add unwanted complexity, which does not go in hand with our goals.
- One match of the game should not take longer than a few minutes. If one match of the game is too long, we will not be able to test it properly.

As game designers we also decided to incorporate one more requirement:

- The game should be symmetrical with respect to the players. The player should be able to accomplish exactly the same set of moves on either side. Such asymmetry would represent unwanted complexity.

Based on previously mentioned ideas and the typical setup of well known games, certain aspects were chosen for the game environment. The game will be played by two players against each other on a board consisting of squares. Each player controls several pieces. Fog of war will be present, so that players will have imperfect information about opposing player's pieces. The specific board properties will change between games and players will be able to affect their starting position.

A distinct feature we decided to use in contrast to typical board games is the fact that players decide on their turns simultaneously within a given time frame. This effectively reduces the game time to half and presents a new layer of player symmetry in exchange for an increase in the complexity of the turn evaluation.

## 2.2 Description of the rules

The game takes place on a $(2k+1) \times (2k+1)$ symmetrical board consisting of unit squares. Each square has a certain terrain type and some of them are marked as spawn squares. The exact position and count of spawn squares, as well as the placement of terrain can be different in each game. Two players will battle to defeat each other with their units. Their units are spawned only at the beginning of the game, exactly one on each spawn square.

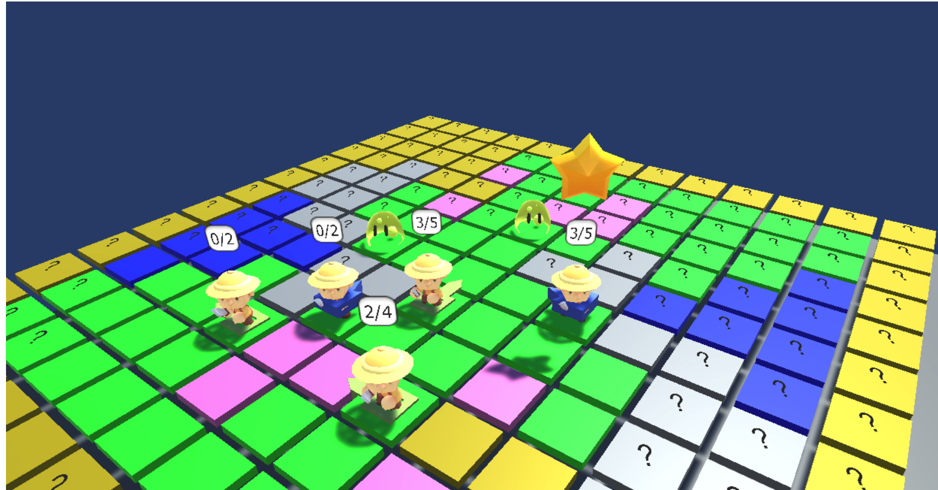


Figure 2.1: The board consists of squares of different colors. Each color corresponds to certain terrain type. Friendly units and enemy units can be are present on certain squares.

Let $s$ be the count of spawn squares belonging to one of the players (the total amount of spawn squares on the board is therefore $2s$). At the beginning of each game, $s$ positive integers are randomly generated and known to both players. Each of the players distributes these positive integers among his spawn squares in a one to one manner and chooses one of his spawn squares to be his king spawn square secretly. In this phase the other player does not know the distribution of numbers, neither which square is the king spawn square. The numbers distributed to the spawn squares will determine the recharge time of unit spawned on that square, which will be discussed later on. An example distribution of these recharge times is shown in figures 2.2 and 2.3.

**The goal** of the game is to keep one's king alive, while slaying the opposing king.

A single game consists of several rounds, in which both players plan to move and/or attack with their units. The plans are executed simultaneously, leading to a new position on the board and the next round. The movement of units is affected and limited by the type of terrain on the board and each player has only partial board information at any given time.

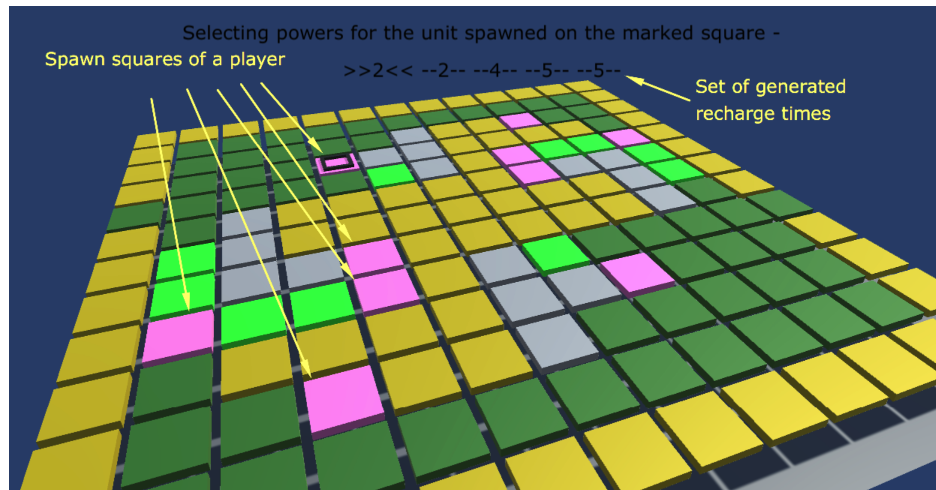


Figure 2.2: Spawn squares are distinguished by pink color. The player is able to select a recharge time for the unit spawned on each spawn square.

Figure 2.3: After selecting all the recharge times, all units are visible and their recharge times are displayed.

**Fog of war** is created on squares which are not close enough to any of the units of a player. Both players have partial information about every square on the board, allowing them to know the type of the terrain on that square at any given moment. Each unit creates an area with radius of 3 in Manhattan distance around itself, which is not affected by the fog of war. The size of this area can be changed by certain terrain types as shown in figure 2.4. The owner of a unit has full information about these squares - knowledge of presence (absence) of an enemy unit, as well as some other minor information about enemy units we discuss later on.

**Neighboring squares** of a square are those immediately next to it in one of the four directions - up, down, left, right (each square has at most four neighboring squares).

Figure 2.4: The unaffected area is marked, fog of war is indicated by a question mark.

Table 2.1: All the terrain types present on the board

| Type of terrain | Passable | Transparent | Slippery |
|---|---|---|---|
| Mountain | No | No | - |
| Water | No | Yes | - |
| Forest | Yes | No | No |
| Desert | Yes | Yes | No |
| Ground | Yes | Yes | Yes |

Five **terrain types** can be found in the game, each of them having different properties as shown in table 2.1, and different colors as shown in the figure 2.5.

*Transparent* type of terrain does not affect the default radius of fog of war around units. On the contrary, opaque (opposite of transparent) type of terrain reduces the radius of non-fogged area to 1 (fog of war is not present on these squares, only if a unit is on one of the neighboring squares).

*Passable* type of terrain can house a unit of a player. Obstructed (impassable) type of terrain can never house a unit.

*Slipperiness* of a terrain affects how units move on it. Details are described under unit movement.

**Unit movement:** Every unit can be either currently executing a command, or waiting to receive one (movement or a battle command). There are four possible movement commands, corresponding to four directions - up, down, left, right. When a unit is assigned a movement command, it moves in that direction by one square every round, until one of the following happens:

- The unit cannot move any more in that direction because an obstructed terrain is on the next square.

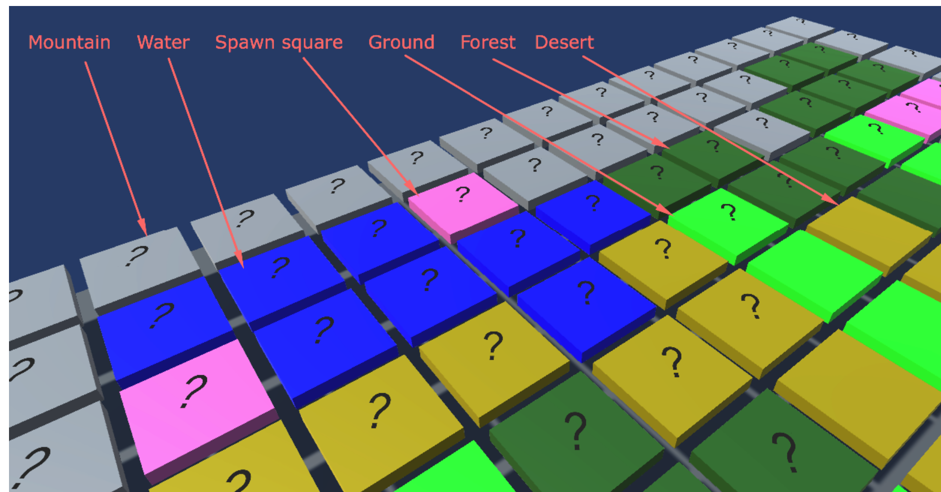


Figure 2.5: All the terrain types are shown here.

- The unit is on a square with non-slippery type of terrain.
- The unit would end up on the same square as another unit (more details in movement collision).
- The unit dies.

The direction of a unit's movement cannot be changed after it is set. When the unit finishes executing one movement command, it can receive another command (this time with a different direction) right away.

**Movement collision** happens when two units (either from the same player or not) try to execute conflicting movements. Two movements are conflicting if:

- More than one unit would end up on the same square.
- Movements are between the same pair of squares in opposing directions.

In case of a movement collision, neither of the units involved move. This may result in additional movement collisions, and the units involved in them will not move either (all the additional units involved are recursively calculated and will not move).

**Board reduction:** On the end of every $5^{\text{th}}$ round (starting with the fifth) the board is reduced. Reducing a board results in the change of terrain on it's circumference. The left-most column, right-most column, top row and bottom row have their terrain type changed to Mountain. This effectively reduces the passable part of the board by 2 in each direction. Any square of Ground terrain on the new circumference of the reduced board has it's terrain type changed to Desert.

Any units residing in the squares whose terrain type has been changed to Mountain are destroyed. At the beginning of the game, all squares of Ground terrain on the

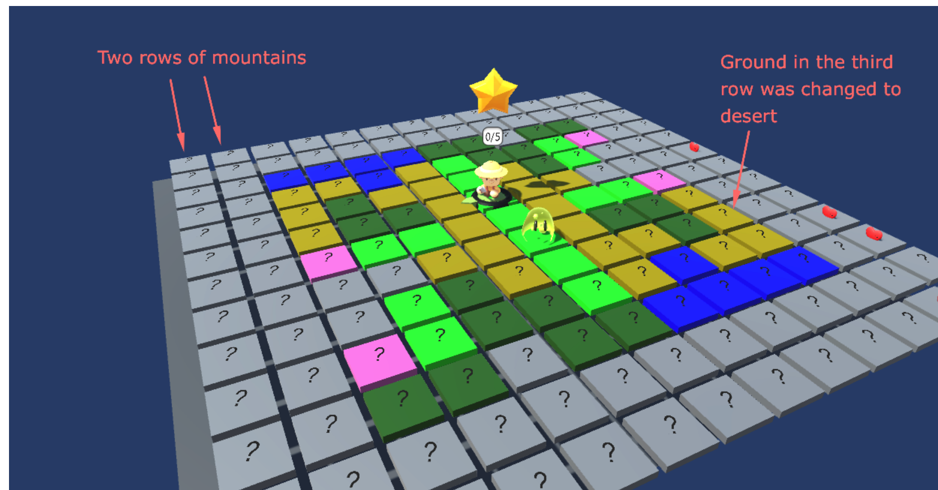


Figure 2.6: This board was reduced twice and all the ground on the third row changed to desert.

circumference of the board have their terrain type immediately changed to Desert. A board that was reduce twice is shown in figure 2.6.

**Battle:** There are four types of battle commands a unit can receive, resulting in shooting a laser in one of the four directions - up, down, left, right. A unit can execute a battle command only if it is charged. Once a battle command is executed the unit is fully discharged and requires a certain amount of rounds to charge.

Every unit (or the spawn square, which it spawned on) has been assigned a number and this determines it's recharge time in rounds. At the beginning of the game all of the units are fully discharged. The current and total recharge time of own units is known to a player at any time as well as which unit was chosen to be the king. On the contrary this information is unavailable for enemy units. A player can see enemy units on squares he has full information about, but their current recharge time, total recharge time or the king status are not known.

Shooting a laser means a laser beam is created on all squares in a given direction between the unit and the next obstructed terrain. The laser beam and it's direction (horizontal or vertical) can be seen by players having full information on the affected squares as in figure 2.7.

Any units (enemy or friendly) on squares affected by a laser beam are destroyed.

**Order of execution** During each round, multiple actions happen and their order of execution determines the result. The order of actions every turn is as follows:

1. All units awaiting orders receive orders from players.

2. All movements are executed.

3. Movement collisions are resolved.

Figure 2.7: Horizontal as well as vertical lasers are shown. If a friendly unit is hit, we may even lose complete information about the squares around it, if no units are nearby.

4. Board is reduced if board reduction happens in this round.
5. Units positioned on the circumference after the reduction are destroyed.
6. If a king was destroyed by board reduction the game terminates.
7. All lasers are shot.
8. All units hit by lasers are destroyed.
9. If a king was destroyed by a laser the game terminates.

Final notes and tips:

- There is no movement command that orders a unit to stay put, but a movement command into a neighboring square with obstructed terrain can have the same effect.
- On every round a player is only asked to give commands to units that are not in the middle of a movement.
- Friendly movement collision are one of the methods to position a unit on a square that would not be reachable otherwise.
- The order of execution can cause multiple units to destroy each other simultaneously by their lasers.
- The game ends when one of the kings is destroyed. A king can be destroyed by a laser or by the board reduction.

- The map is symmetrical with respect to the player's starting positions in spawn points and terrain types.
- Spawn points act as Ground terrain type.
- If a unit requiring a command was not assigned a command, a random movement command is chosen for it.

## 2.3 Key elements in the rules

In this section we shall quickly take a look back at our rules and the requirements we set for them. Each of the requirements may be addressed in multiple elements of the rules, as well as some elements of the rules may address multiple requirements. In the table 2.2 we provide a comprehensive overview.

Table 2.2: Key elements in the rules addressing requirements

| Requirement | Addressed as |
|---|---|
| robust state space | size of the board, count of pieces |
| | fog of war |
| | unit recharge times |
| | variable maps and terrain types |
| starting configurations | random set of unit recharge times |
| | secret distribution of unit recharge times |
| | variable maps |
| game symmetry | symmetrical map |
| | equal set of unit recharge times for both players |
| | simultaneous turn execution for both players |
| imperfect information | fog of war |
| | recharge time of enemy units |
| | choice of the king |
| simplicity & intuitiveness | unit collision resolution |
| | simple goal |
| few special cases | unit collision resolution |
| | only difference between the units is recharge time |
| match length | board reduction |
| | slippery terrain |
| | time limit for each turn in our implementation |

Before we move on to the next section, one part of the rules is worth revisiting. Each

type of terrain has three different properties - passability, transparency and slipperiness. The first two may seem as a natural choice when choosing differences between terrain types and are often present in different games as well. All of their combinations give us four terrain types - mountain, water, forest and ground/desert. From this point of view, why would we decide to add another element of complexity in form of slipperiness to the game?

The answer is a little surprising. We did list slippery terrain as one of the points we use to address match length. The fact that for some units the player will not have to choose a command every turn, will in fact reduce the game length. But this seems rather like an overkill. The game length could be reduced in several other ways and while we want to keep the rules as simple as possible, it does not sound like a wise rule.

The true motivation elsewhere. To understand it, let us try to look at the game without this element from a perspective of an amateur player, who understands the rules and played a few games. Due to the board reduction, the action in the game will incline towards the center of the map. Keeping your king as far as you can from the center, hidden behind a mountain, while peeking out with other units to briefly shoot lasers at as many squares on the other half of the map as possible, seems to be a decent strategy. At this point, the game becomes dull. It is often very obvious which mountain to use for cover or how many steps you should take in some direction before shooting. Picking the optimal movement direction is a short-term and non-challenging decision for a human most of the time.

With the addition of terrain slipperiness, the dynamics of unit movement are very different. If you want to move your unit to a certain square, it may be unclear what is the best path to take. Some movement commands may be long-term and have great impact on the game, so you need to think twice before executing them. Sometimes movement commands may even take your unit into an inescapable position. As with the game of Pentago, mentioned in the previous chapter, this kind of movement is harder to visualize for a human.

This affects the way a human and an AI can attribute to a strategy. More and less skilled human players can be distinguished based on how well can solve given movement situations. An AI can use it's raw computing power to rapidly find the quickest ways to a square or to find the right one out of the possible positions in a few turns.

We increased the game difficulty for a human player and increased the possible attribution of an AI in order to encourage their cooperation. Furthermore, the change should not be too dramatic to make the game too hard for a human or too easy for an AI, leading us to a golden middle way.

## 2.4 Weak points of our design

In our design, we wanted to minimize the probability of the game ending up in a tie. Tied game does not give us any meaningful information about the skill of the two matched players. If we were to eliminate the possibility of a tie entirely, we would need to decide what would happen if two deterministic players were to play a game against each other. In that case, the only remaining possibility seemed to be to add some random factor in the game. However, leaving the result to be a flip of a coin seems as an even worse option than allowing a tie.

Given the current set of rules, the tie can happen with non-trivial probability. Two players can shoot each other's kings simultaneously or the kings can be killed by board reduction at the same time.

Another drawback of the rules may be their complexity. Although we are confident that during the design phase we took several steps in order to eliminate the special cases and increase the overall intuitiveness of the rules, the result is not as neat as expected. It may require an entirely different setup than a board of squares with units on them to create a game fulfilling our requirements with lesser complexity.

The overall verdict on the design of our game is positive, as we have met all the presented requirements, while creating only a few minor drawbacks.

# Chapter 3

# Implementation of our game

After we defined the requirements for our game and it's rules in the previous chapter, in this chapter we will describe our specific implementation of the game. The implementation is an important step to demonstrate the core concepts of this game type.

We will have a certain set of requirements for the implementation. These differ from the design requirements from the previous chapter in a way that they are more of a technical nature. The challenges include playability, usability and attractiveness. The role of the AI cooperating with the human player is represented by what we call an assisting program. Each player can write his assisting program, which can alter the course of the game.

The testing phase in the next chapter targets high school students, with little to moderate algorithmic programming skills which affects the implementation heavily. Most of the students have no former experience in the field of installing or configuring a multi-part application. Therefore the technical setup is ought to be as simple as possible.

## 3.1 Implementation requirements

We are creating a game, which is to be played by (but not necessarily only) high school students. The participants of Spring Camp 2018 of Correspondence Programming Seminary will be the first ones to test it thoroughly. Our game may serve as a programming teaching tool and motivation resource.

It is therefore required that the setup process is minimal. Running and playing the game, along with programming a player's very own assisting program to guide him through his turns should be straightforward. The following list does summarize our requirements:

- The game has to run under Windows, Mac OS X and most of traditional Linux

distributions. The students are using different operating systems, and in 2018 this is an expectable prerequisite for any software. It is not important whether we will create native executables or a web application, as we are going to use a game framework which will take care of this for us.

- A player's game client has to be able to connect to a game server, and be matched with the desired opponent. The server will serve as an intermediary between the clients verifying integrity of the data sent between them. For logging purposes the server should be able to record all the game data.
- The gameplay should follow the rules described in the previous section as closely as possible. Other than that, the server should be able to set a time limit for the turn of a player.
- To support the benefits of using an assisting program, it should be possible to generate random maps, meeting some basic playability requirements. If we were to play only on a small set of maps, the players could easily adapt to their characteristics and learn certain hardwired strategies for them.
- The protocol used to communicate with an assisting program should be available to players. Thanks to this, it is possible to program an assisting program in any language, that can produce some form of executable. A template with a simple example should be provided in order to make everybody able to pick up the main concepts easily.

Given that our implementation meets these requirements, the program should be prepared for testing.

## 3.2 Architecture overview

After specifying the requirements, we will describe the specific architecture we have decided to use and briefly go over the process behind the decisions.

On top of the game being fully functional, we also require it to be attractive for both beginners and experienced programmers. These requirements easily rule out a bunch of setups. For the concept itself, it would be sufficient to have a console interface to the game, or the assisting program. However, that would hurt the attractiveness. In order not to reinvent the wheel, writing low-level graphics library, or a decent game management library is not an option either.

As developers we had a bunch of different framework options, from which we considered LWJGL [2], Unity3D [18], PyGame [14] and Godot [10]. We have decided to use Unity3D, which provides the highest level of control out of the given options.

Unity3D can be used as a mature game engine allowing us to focus on the important parts of the implementation. The engine supports 2D content as well as 3D content. We implemented the game in 3D (even though it's concept do not necessarily imply it) in a hope that this choice will lead to an increase in attractiveness among young beginner programmers. Thanks to Unity3D, little to no overhead is required for this step.

The game is supposed to be multi-player, possibly over a network, which lead us to a server – client architecture. To maintain simplicity and adhere to language paradigms, the communication between the server and clients uses TCP sockets. We will define a simple protocol to transfer required data.

Last part of the pipeline is the assisting program. In order to provide as much flexibility as possible, it communicates with the client asynchronously with respect to the game state. Communication uses standard input and output and a simple string protocol, similar to the one used between the client and the server. This allows the players to write an assisting program in any language as long as it supports standard output and input streams. As part of this thesis, we provide a fully prepared python wrapper.

In the next subsections we take a brief look at the specifications and properties of each component of our pipeline.

### 3.2.1 Game server

Capabilities of the server:

- accept TCP socket connections from various clients
- allow the clients to be matched against anybody in parallel by selecting a room name
- authenticate the clients against a user list and pass board information between them
- keep an internal game state structure, execute the requested commands on it, and send appropriate responses to players
- track the time limitation for a turn
- send end game responses to clients
- allow a single client to connect in a single-player test mode

The main parts of the internal game state are: board information, unit positions and information, and board reduction information. The need for the last item emerged

during the testing. In our implementation, any room whose name begins with #(hash-tag) is considered as a single-player testing room by the server, and the game is starts immediately after a single player connects to it. There is no time limit tracking in the testing mode and the second player mirrors the commands of the first player. Powers selected are mirrored as well, but randomly chosen actions (in case the player does not select a command for a unit) can be different.

### 3.2.2 Game client

Capabilities of the client:

- open TCP connection to the server
- allow the player to choose a game name, unit powers and assign commands to the units
- display an appropriate visualization of the game situation including the board, units and lasers
- allow the player to access visualization controls (e.g., camera)
- respond to the server with selected commands
- run a separate process for the assisting program and communicate with it
- appropriately react to assisting program commands

### 3.2.3 Assisting program

In the design phase we stated a few requirements for the assisting program in order to make it easy to use. On top of these, we aimed to avoid making the program too overwhelming or abstract, as that would lead to difficulties for beginner programmers. Each player can make use of three main functions of the assisting program - displaying a text on the screen, on a square, or displaying a prepared 3D model on a square. All of these three functions are shown at 3.1.

Other than that, we wanted to allow the players to create fully functional automated players - bots. For this purpose, the assisting program can set commands for units and advance game turns.

Capabilities of the assisting program:

- receive messages from the client about the game state
- display text with given properties on the screen of the client

- display text with given properties on any of the squares of the game board
- place an object chosen from a predefined set of objects on any of the squares of the game board
- set commands for the units and advance game turns

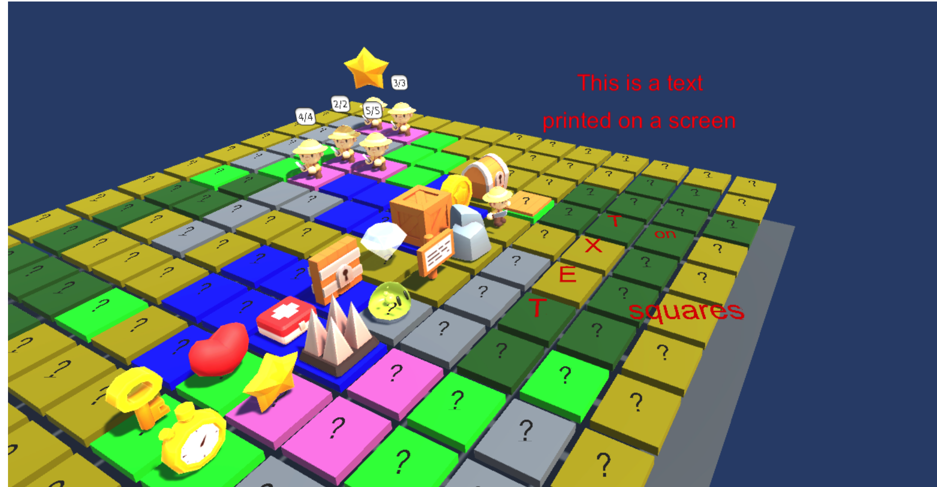


Figure 3.1: The figure shows us text printed on the screen, and one or more characters of different sizes printed on squares. All of the sixteen objects from the predefined collections are placed as well.

## 3.3 Example assisting programs

In this section we will take a closer look at the structure and demonstrate the capabilities of the assisting programs. On the first look it may be hard to come up with useful ideas to implement in an assisting program. Coming up with such ideas is one of the challenges the players of our game will need to face.

To illustrate some of these ideas, let us take a look at some examples of usage of the assisting program. We will accompany these ideas with examples of python code, that can be used with very little effort and changes in our framework.

Before jumping straight into the examples, let us take a look at the code structure of the program into which we are going to write. Examples may be specific for python, but one should keep in mind, that any language can use standard input and output to communicate with the client.

Listing 3.1 is a simplified version of the example code that greets a new player. We provide the code in two python files - helper and the assisting program itself. We will show only parts of the assisting program down below.

The helper contains definitions of methods used for creating and removing texts or objects (`create_text`, ...), as well as parsing of the input from the client. This code is pretty straightforward and players need to make only slight improvements to it when needed.

Listing 3.1: Assisting program structure

```
# ... imports left out

state = GameState()
# ----- Write your code under this line -----

# ----- Edit this method to execute code every second -----
def everySecond(seconds_from_start):
        pass

# ----- Edit this method to execute code every turn -----
def everyTurn():
    log('Logging␣state:')
    log(str(state))

    # Examples of available commands
    text_id = create_text(100, 100, 'Hi!', color='#FCF323', size=35)
    remove_text(text_id)
    sqText_id = create_text_on_square(5, 6, 'X', color='red', size=1.2)
    remove_text_on_square(sqText_id)
    sqObject_id = create_object_on_square(2, 3, 4, 1.3)
    remove_object_on_square(sqObject_id)

# ----- Management code under this line -----
# ... management code left out
```

Without advanced modifications the user will be typing his code into one of the two methods shown - `everySecond` and `everyTurn`. The provided log method will write it's input to a log file inside the same directory as our assisting program. This is a useful way to debug the assisting programs, as the standard input and output are already used for communication with the client.

Any time a python error happens in our template, it logs the precise error to the log file and displays an indicator text in the client. If a more serious crash happens, such that even our exception catching routine cannot handle it, the python program crashes. The client is monitoring crashes in the assisting program and tries to restart

the assisting program after a given time delay.

### 3.3.1 Simple game counters

There are two challenges probably every new player will bump into after first few games. It is important to keep track of when the board is going to reduce in order to evade the death of our units. Secondly it is crucial not to exceed your given time limit for a turn, otherwise you will be greeted with a losing message next turn.

The assisting program interface provides a simple solution to both of these problems, as shown in figure 3.2. Every second we print the remaining turn time to our screen as well as the current turn number. We know that board reduction happens every fifth round, so now we can prepare for it.

Listing 3.2: Example with basic game counters

```
# ... imports and definitions left out

current_turn_time = 0
turn_time_id, turn_count_id = None, None

def everySecond(seconds_from_start):
    global current_turn_time, turn_time_id
    current_turn_time += 1

    if state.turn_id >= 0:
        if turn_time_id:
            remove_text(turn_time_id)
        turn_time_id = create_text(-200, 200, str(state.turn_time-
           ↪ current_turn_time) + '␣seconds␣remaining', 'white', 20)

def everyTurn():
    global current_turn_time, turn_count_id
    current_turn_time = 0

    if turn_count_id:
        remove_text(turn_count_id)
    turn_count_id = create_text(-200, 150, 'Turn␣no.' + str(state.
        ↪ turn_id), 'white', 20)

# ... management code follows
```

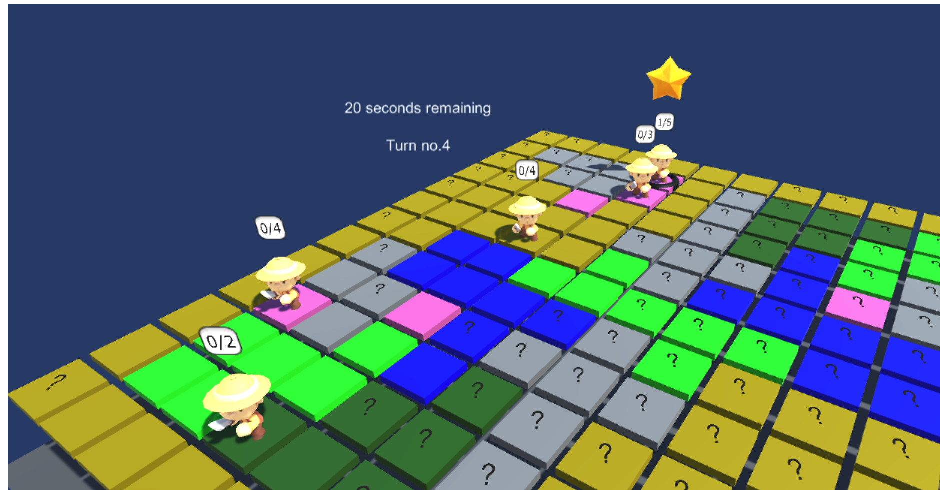


Figure 3.2: Both the remaining time for the current turn and the turn number are shown.

### 3.3.2 Previous enemy positions

The fog of war is an ever changing element of the board. When a player is deciding when and where he should shoot a laser, it may be useful to remember the previous positions of enemy units. They may help to make a more educated guess of an enemy position or even rule out some possibilities.

It would be certainly unfortunate to lose the ability to see where the fog of war still is and where it is not due to this strategy. Fortunately, we can use color of the text to differentiate between the two cases.

The last implementation simplification will be the usage of custom IDs for our displayed texts. These IDs hash the square position so before printing out the new numbers we can easily delete the old ones without creating a structure to remember them. Calling object removal on a non-existent ID is expected to do nothing and is valid.

Listing 3.3: Example with previous enemy unit positions

```
# ... imports and definitions left out

old_enemy_positions = {}

def everySecond(seconds_from_start):
        pass

def everyTurn():
    for position in old_enemy_positions:
        old_enemy_positions[position] += 1
```

```
    for position in state.enemy_units:
        old_enemy_positions[position] = 0

    for position in old_enemy_positions:
        remove_text(1000+position[0]*100+position[1])
        create_text_on_square(position[0], position[1], str(
            ↪ old_enemy_positions[position]), color='red' if position
            ↪ in state.visible_squares else 'black', id=1000+position
            ↪ [0]*100+position[1])

# ... management code follows
```

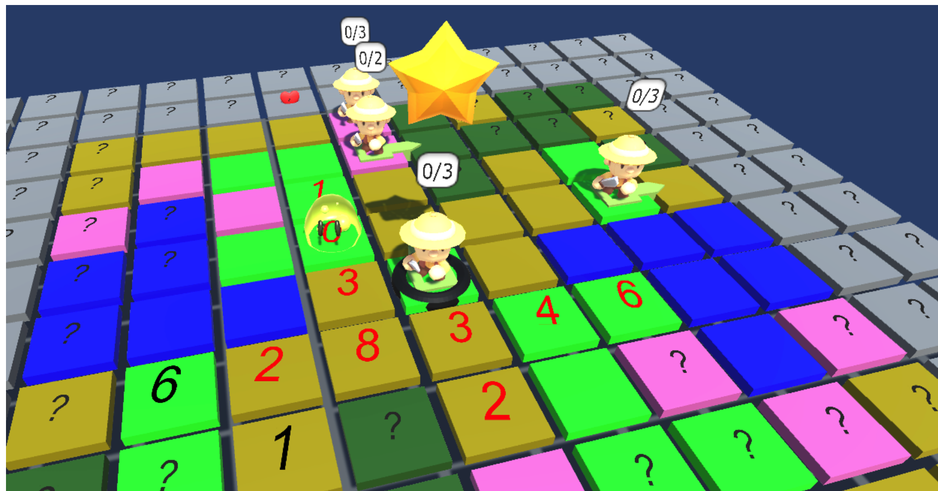

Figure 3.3: The numbers on the squares denote how many turns passed since we last saw an enemy unit at those squares. Squares unaffected by fog of war have their numbers colored in red. Numbers on the squares in the fog of war have black colored text.

### 3.3.3 Least dangerous positions heuristic

In the last of our examples we will try to mark the current least dangerous squares on the board. To approximate square's danger we use a simple heuristic technique we made up, along with the constants it uses.

For each passable square, we inspect all squares that could possibly shoot a laser at it. Based on the terrain type of these squares, we increase the so called danger of this square. Afterwards, the squares with the lowest danger should approximate the least dangerous positions on the map.

Listing 3.4: Example calculating least dangerous squares

```
# ... imports and definitions left out
top_positions_ids = []

def everyTurn():
        for id in top_positions_ids: # Remove old indicators
            remove_object_on_square(id)
        top_positions_ids = []

        danger_storage = []
        for i in range(state.row_count):
            for j in range(state.column_count):
                if state.board[i][j] == 'M' or state.board[i][j] == 'W':
                    continue

                square_danger = 0
                # range, rows True - columns False
                ranges = [((i, state.row_count, 1), True),
                          ((i, -1, -1), True),
                          ((i, state.column_count, 1), False),
                          ((i, -1, -1), False)]
                for r in ranges:
                    for k in range(*r[0]):
                        square = state.board[k][j] if r[1] else state.
                            ↪ board[i][k]
                        if square == 'M': break
                        if square == 'W': square_danger += 1
                        if square == 'P': square_danger += 7
                        if square == 'D': square_danger += 10
                        if square == 'F': square_danger += 11
                danger_storage.append((square_danger, i, j))

        danger_storage.sort(key=lambda x:x[0])
        worst_danger = danger_storage[-1][0]
        # Create indicators
        for i in range(min(10, len(danger_storage))):
            b, r, c = danger_storage[i]
            top_positions_ids.append(create_object_on_square(r, c, 3,
                ↪ max(0.25, 2-(worst_danger-b)/50)))
```

```
# ... management code follows
```

The provided strategy is very simple and there are numerous things we do not consider and can affect what we call danger of a square. The strategy serves rather as a starting point for similar heuristics. In some situations, the marked squares really do match the seemingly least dangerous squares, as shown in figure 3.4.

Heuristic approaches are the middle ground between very simple calculations, like those shown earlier, and advanced algorithms. As a result, heuristics are a way, how even an algorithmically unexperienced programmer can make an attempt for a strategy.

Figure 3.4: The little diamonds are placed at positions that are supposed to be least dangerous

# Chapter 4

# Final testing and results

After designing and implementing our game, we decided to thoroughly test it with a group of volunteers. As mentioned in the previous chapters, we aimed to design a game, which is challenging for a computer and for the human brain as well. During the testing, we have expected to observe human-only players as well as some players, which rely on their programs to do most of the work. Our main goal has been to capture moments, when an assisting program changed the course of the game in some way. These moments serve us as examples of game positions, at which the raw computing power could somehow help a human player to make a more educated decision. Analyzing these moments we would try to compare them to what we proposed as strengths and weaknesses of a human or an AI.

Testing took part during the Spring Camp 2018 of Correspondence Seminar in Programming (CSP) [1]. CSP is a correspondence competition primarily aimed at high school students in Slovakia. The competition periodically presents students with challenging problems, which they solve both theoretically and practically. CSP motivates high school students to become proficient programmers and skilled problem solvers with analytical and critical thinking. Semiannually the best 36 students are invited to a camp like the one we have tested our game at. Most of the successful participants of Slovak Olympiad in Informatics (OI) [2] are trained through CSP. Moreover the best of the students tend to participate at the International Olympiad in Informatics (IOI) [3].

In order to properly understand the testing environment, we briefly describe the skill level of the students and their engagement in similar activities. In the following sections we outline the schedule for the testing day and report the results of the testing. Afterwards, we examine the student responses to our survey about the game and it's concept. At the end of the chapter we summarize the results.

[1]Correspondence Seminar in Programming - `https://ksp.sk/`

[2]Slovak Olympiad in Informatics - `http://oi.sk`

[3]International Olympiad in Informatics - `http://ioinformatics.org/`

## 4.1 Testing environment

During the camp, one whole day was reserved for the testing of our game. That is, about 11 effective hours of time with breaks for lunch and dinner. The participating students highly differ in their skill in programming. Most of them understand the basic concepts of simple algorithms and time complexity. Yet, practical programming tasks, like creating their assisting program can be very overwhelming for them at first. For some, it might even be the first program they would try to write, that does not just solve a given task at a competition, or a homework at school.

On the other end of our skill spectrum, we have a few well educated and prepared programmers. They are very capable in algorithmic thinking and have previous experience in similar competitions. This experience in setting up development environments and writing their made up programs full of heuristics makes them very well equipped for the task.

In order to promote experience exchange and make the task more attractive, students were allowed to form teams of up to 4 people, which participated in the final competition. Other than that, the beginner programmers, even when teamed up, would need a lot of support and assistance. For this purpose, most of the organizers joined one of the beginner teams to be their member. During the day, they have helped to motivate the students and provided them with technical support.

### 4.1.1 Testing schedule

Based on the previously described assumptions about the testing environment we created a time schedule for the day. Participating teams would play two types of matches. First, each team would play three to four best of three matches against randomly chosen teams. These qualification matches would provide us with base ranking to create a single elimination tournament bracket, which would be played after dinner.

09:00 - 10:00 - Game rules, programming and competition description
10:00 - 12:30 - Initial setup, familiarization with the game
12:30 - 13:15 - Lunch break
13:15 - 18:00 - Main coding time, qualification matches
18:00 - 18:45 - Dinner break
18:45 - 20:00 - Finalization of programs, qualification matches
20:00 - 21:30 - Tournament matches
21:30 - 22:00 - Finale
22:00 - 22:15 - Feedback collection

Approximate of 7 hours of time was planned for the participants to get to know the game, play it a few times and try to create their own assisting programs.

## 4.2 Results

In the end, the testing itself was not as smooth or successful as expected. We did not capture any of the moments where a computer changed the course of the game, neither did the students come up with impressive ideas. The students did not spend most of the time as we originally planned, which was due to various reasons. From this point of view, we could say our testing was not a success. During the testing, we found out a lot of ways to improve the implementation of the game. Some moments of the testing even indicated that our participants were not chosen very wisely. A more experienced set of programmers might be required for our purpose. The provided time frame may have been a bottle-neck as well.

Below, we recap the the testing day. We will take a look at it from multiple different perspectives and try to find it's interesting moments. In the following subsection we cite some of the survey responses from the participants. In the last subsection we compile a list of take away messages. These things are either implementation changes or overall conceptual differences.

### 4.2.1 Testing walk-through

Multiple unprecedented factors affected our testing. They were both of technical and personal nature. In the following paragraphs a chronological overview of the testing is provided along with specific moments.

At nine, we started to explain the rules of the game and competition that the participants are going to take part in during the day. It took nearly twice as much as originally planned. A lot of the time was consumed by a seemingly never stopping stream of questions.

The students were given a link to a website with all the necessary materials. The website included off-line installers of handy IDEs and languages, all the required documents and compiled client binaries. Among the required documents they could find the full description of the rules, the full description of protocols between the server and the client as well as between the client and the assisting program. Compiled binaries included versions for Windows and Linux, both 64-bit and 32-bit. Based on the team's choice, the binaries were incorporated with example assisting program in Python or C++. On the website a configuration guide along with basic tips and tricks was present.

Even though all the required materials with descriptions were provided, the students often failed to set up their environments. This may be caused by their lack of experience in comprehending a written configuration tutorial. The configuration was as simple as we have managed to provide. Excluding the most experienced teams, each team

had needed technical assistance. Due to lack of upfront preparation even some of the organizers failed to help their teams in setting up their environments smoothly.

Our network architecture was not well designed either. Moments after the beginning we started to have problems with the network being overloaded. Due to this, we were forced to quickly edit the server code and distribute it among the students, to run their own servers in different rooms. This stripped down version of the server did not support logging or some other advanced features. After the lunch break, most of the students had successfully set up their clients and proof-of-concept assisting programs. Distributed throughout the building, groups of teams were hosting their own servers, playing the game against each other.

The teams struggled with basic assisting program strategies for the most of the afternoon. Most of the students did not code before in more than a single language (for most of them - C++). Adapting themselves to Python was not something they could do in such a short timespan. But for our assisting program purposes, python was a very well equipped language, therefore the client - assisting program protocol did focus around it. Parsing the strings from the client and formatting them back again can be a problematic task in C++.

On the contrary, teams which did code in Python did lack debugging experience. Inspecting the Unity3D client logs was a big challenge and the idea of wrapping the whole assisting program in a try / except block was an idea suggested by one of the students only after the competition. Without the possibility of debugging to standard output most of the teams had a rough time making their code work.

Overall, the lack of functionality in the wrappers for both languages along with non-trivial debugging, made the experience very unsatisfying for most teams. Before dinner, they just finished familiarizing themselves with the core concepts of the game and were not able to create truly helpful assisting programs.

The network problems were nearly solved by the transition to local game servers, so the teams played quite a few matches between themselves. Playing the game itself was entertaining, but writing an assisting program was simply too challenging. Most of the teams did manage to write an equivalent of our game counters example from previous chapter, but nothing more.

After dinner, each team played a few quick matches and we have determined the best two teams using single elimination bracket. These two teams played best of three, while their screens were projected to the wall for others to watch. The finale was a pretty enjoyable end of the day.

Overall verdict seemed to be that the game is well designed and fun to be played as a human player game. But given the short timespan, technical problems, and lack of programming experience the purpose of assisting programs was not fulfilled.

### 4.2.2 Survey responses

After the finale, the participating teams were kindly asked to fill out three questions and give us the permission to use this data in the thesis. The questions asked about the realization of the game, concept of a game with an assisting program and what was the most challenging part for them. Below are a few translated excerpts from these anonymous responses. We picked the most surprising, informative, and wise responses to present here. All of the responses are included along with the code with the thesis.

A few of the responses were similar to this one:

**What was the greatest challenge for you?**
"To make up and code something that would be useful and help us in the game. And of course to make it work."

We tried to design the game in a way, that an assisting program could help the player somehow. These responses question whether we were successful in achieving this goal.

Due to missing features of the Python wrapper and lack of experience, most of the responses stated facts like this:

**What was the greatest challenge for you?**
"Everything. (laughter) Probably that, how is it called ... Parsing!"

In the question about the concept of a game with an assisting program, most of the answers were positive.

**How would you rate the idea of a game, where you can code some additional functionality into your client?**
"Never heard of the concept before, but I really like it. Would love to try more similar things."
"I believe this idea is really great, I like it and we need more such games. I can imagine such modification in a lot of strategic games like Starcraft or MOBA [4] games. What I would need is a way of coding inside the game, so it would be accessible for more people ... "
"... the idea is interesting and very engaging. It certainly is an excellent way how to introduce programming to less experienced kids."

From these responses it seems that we were on point with the design but lacked in implementation quality. We also tried to design the game to be engaging for beginner programmers, which has seemed to work out well. The idea of coding an assisting program directly inside a game, could eliminate the undoubtedly most challenging part for most participants - setup and parsing.

[4] MOBA stands for Multiplayer online battle arena

### 4.2.3 Testing takeaways

After the testing, we have learned a lot. Had we known all of it in advance, we would do a lot of things differently.

Firstly, the testing also served as a thorough bug testing for the implementation of the game. There were no major bugs or unexpected behavior in the game itself. In contrast, the assisting program engine and it's wrappers contained a few bugs. The python and C++ wrappers also lacked in features.

A very useful idea one of the participants provided was to implement a single-player testing mode to the game, which we later did implement. Beforehand a player had to start two instances of the game to test his assisting program, which complicated his progress.

The Python wrapper did not contain the code to fully parse all of the input from the client and the students had to write this part themselves. We did not consider it to be a big of a problem and expected them to parse only the data they actually wanted to use. Yet, most of the participating teams did not parse the data successfully, or even got stuck in trying to parse data, they did not even want to use in the first place. The idea to wrap the core functions of the assisting program in a try / except block also occurred only after the testing. That, along with a simple logging interface outputting to a file, made debugging orders of magnitude easier.

Secondly, based on the progress during the day and some feedbacks, we have developed a general feeling, that the target audience was not appropriate. Despite them being high school students with at least moderate interest in programming, our participants may have simply lacked the programming skill required. The testing would have required a lot more time to achieve the goal set specified at the beginning of this chapter. On the other hand, a long term competition usually suffers from a lack of motivation.

As a result, we believe that testing with a more experienced group of programmers would end up rather differently.

# Conclusion

In our work, we have listed the most important historical examples of games, where humans and AIs were matched against each other. In some games, the AIs can outperform the best human players by a large scale. In others, the best AIs can barely challenge a beginner human player. We have also included examples of games, where raw computing power allowed us to simply search for the best strategy.

We have shown, that imperfect information, randomness, deceit, and complex rule set make the game a lot harder for an AI. On the contrary, small state space and little variation create the ideal environment where humans can get outperformed by AIs.

With the knowledge of strengths and weaknesses of both human players and AIs, we have recapped the design requirements for our game. The design goal for our game was to make it challenging both for the human player and an AI, while allowing them to cooperate.

Our two-player game is a computer game, taking place on a board consisting of squares. Each square has it's properties and each player controls his own units. The units are able to move across the board and shoot lasers periodically. These lasers can eliminate the enemy units. A player wins if he eliminates the unit his opponent has secretly chosen as his king. The players have imperfect information in the form of fog of war. The board and unit recharge times are generated randomly, leading to a high starting position variance and a large state space.

We have implemented the game using Unity3D for the game client and Python for the server. Players can write their assisting programs to improve their game-play. These assisting programs communicate with the client through standard input and output. An assisting program is able to show text on clients screen or on one of the board squares. Bots for the game can be programmed as well, due to assisting program being able to assign commands for the units and advance through the game.

The possible usages of an assisting program are numerous, so we have listed a few of them with complete code examples. Along with the thesis we provide a full fledged Python wrapper for the assisting programs.

Our game underwent thorough testing on the Spring Camp 2018 of Correspondence Seminar in Programming. Even though we had technical issues and the implementation did not contain as much features as it does now, we have learned a lot. After the testing,

based on the feedbacks from participants, we have improved the client, as well as the provided Python wrapper. These changes allow even a less experienced programmer to easily play the game and write his own assisting program.

Unfortunately, due to unpolished testing methods, we were not able to fulfill our initial testing goals. We did not capture a moment, where an assisting program would help the human player and change the course of the game dramatically. Most of the created assisting programs contained only basic functions such as simple game counters.

On the other side, we did prove the concept of a game where a human cooperates with a program to be interesting. It can motivate beginner programmers to engage in programming as well as bring a lot of experience. Altering ones own game client to his needs during the game requires creativity and critical thinking.

Given the current state of society inclining towards information technologies, we believe this type of a game may become a lot more popular in the future. It may be well presented at the competitive scene as well as at educational institutions.

# Bibliography


[1] Louis Victor Allis. Searching for solutions in games and artificial intelligence, 1994.

[2] Antonio Hernández Bejarano. *3D Game Development with LWJGL 3*. GitBook, 2017.

[3] Paul M Bethe. The state of automated bridge play, 2009.

[4] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1):201 – 240, 2002.

[5] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 2017.

[6] Public Domain. Mao - the game. `https://boardgamegeek.com/boardgame/4213/mao`.

[7] Eben Harrell. Magnus Carlsen: The 19-Year-Old King of Chess. `http://content.time.com/time/world/article/0,8599,1948809,00.html`, 2009.

[8] Felix Hill, Anna Korhonen, Yoshua Bengio, and Kyunghyun Cho. Learning to understand phrases by embedding the dictionary. *Transactions of the Association for Computational Linguistics*, 4:17 – 30, 2016.

[9] Geoffrey Irving. Pentago is a first player win: Strongly solving a game using parallel in-core retrograde analysis. *CoRR*, abs/1404.0743, 2014.

[10] Juan Linietsky, Ariel Manzur, and The Godot Engine Team. The godot game engine. `https://godotengine.org/contact`, 2011.

[11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2nd edition, 2003.

[12] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[13] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241 – 275, 2002.

[14] Pete Shinners and PyGame development team. Pygame. `http://pygame.org/`, 2011.

[15] D. Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 10 2017.

[17] Omar Syed and Aamir Syed. Arimaa: A new game designed to be difficult for computers. *ICGA Journal*, 26:138–139, 2003.

[18] Unity Technologies. Unity 3D - Personal edition. `https://store.unity.com/products/unity-personal/`, 2018.

[19] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[20] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, mar 1995.

[21] H.Jaap van den Herik, Jos W.H.M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1):277 – 311, 2002.

[22] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[23] David J. Wu. Designing a winning arimaa program. *ICGA Journal*, 38:19–40, 2015.

[24] Yoshiaki Yamaguchi, Kazunori Yamaguchi, Tetsuro Tanaka, and Tomoyuki" Kaneko. *Infinite Connect-Four Is Solved: Draw*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[25] Chih-Kuan Yeh and Hsuan-Tien Lin. Automatic bridge bidding using deep reinforcement learning. *CoRR*, abs/1607.03290, 2016.