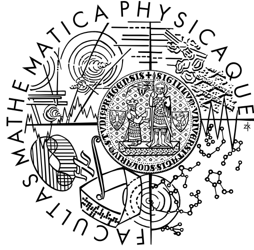


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Fibich

Grid-based Online Multiplayer Strategy Game

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Jakub Gemrot, Ph.D.
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D. for his feedback, suggestions and valuable advice. I would also like to thank my friends who have supported me during my studies.

Title: Grid-based Online Multiplayer Strategy Game

Author: Michal Fibich

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Playing a massively multi-player on-line real-time strategy game is connected with expectation of playing with other players close to your position in the game. Some games of this kind have a very long life cycle where the progress of each player is persistent. A match can take months, even years to finish. However, every match is very dependent on the amount of participants which is not always ideal. Different seasons of the year can cause massive drops in the amount of players. There have already been attempts to incorporate an artificial intelligence to these matches, but the goal was to provide a win condition instead of fighting a decreasing player base. That is why we have started developing a framework which includes the basic mechanics of the game and allows customization of basic game elements such as units, resources, buildings or entire nations. Part of the framework is an artificial intelligence which is capable of playing games created using the framework along with players. The problem was to find a proper behaviour for the artificial intelligence that has a balance between computational complexity and effectivity. A positive outcome of this experiment will influence the critical phases of these types of games by adding entities controlled by mentioned artificial intelligence until the amount of active players returns to acceptable level. This might also increase the chance of success of new game servers that use this framework.

Keywords: video game online multiplayer strategy game grid

Contents

1 Introduction **4**
1.1 Our Goals . . . 5
1.2 Structure . . . 5

2 Problem Analysis **7**
2.1 About Travian . . . 7
2.2 Key Components of Travian . . . 8
2.2.1 Real-time gameplay . . . 9
2.2.2 Length of life cycle of game worlds . . . 9
2.2.3 Persistent game progress . . . 9
2.2.4 Available Tribes . . . 9
2.2.5 Ownership of Multiple Villages at once . . . 11
2.2.6 Buildings and Resource Fields . . . 11
2.2.7 Resource Management . . . 13
2.2.8 Military Combat . . . 14
2.2.9 Village Management . . . 15
2.2.10 Player Strategies . . . 15
2.2.11 Player Interaction and Alliances . . . 19
2.2.12 Victory Condition . . . 19
2.3 Main Differences of Framework . . . 20
2.3.1 Generic Victory Conditions . . . 20
2.3.2 Replacement of Consumable Resource . . . 20
2.3.3 Founding New Village . . . 21
2.3.4 No Loyalty . . . 21
2.3.5 Capturing Enemy Cities . . . 21
2.3.6 No Research . . . 21
2.3.7 Possibility of Controlling Multiple Nations . . . 21
2.3.8 Resource Fields . . . 21
2.3.9 Exchanging Resources . . . 22
2.3.10 Combat Formulas . . . 22
2.3.11 Canceling Actions . . . 22
2.3.12 Events Instead of Oases . . . 22
2.3.13 Upgrades of Military Troops . . . 23
2.3.14 Alliances . . . 23
2.3.15 Artificial Intelligence as Player . . . 23
2.4 Used Technology . . . 23
2.5 Goals Breakdown . . . 23

3 Implementation **25**
3.1 Network Protocol . . . 25
3.2 Data Serialization . . . 25
3.3 Client-Server Communication . . . 26
3.4 Saving the Game . . . 27
3.5 Loading of Game Configuration . . . 27
3.6 Data Storage . . . 28

3.7 World Map . . . 28
3.7.1 Obtaining Map Information . . . 28
3.7.2 Positioning Algorithm . . . 30
3.7.3 Adding Event Tiles To Map . . . 31
3.8 Timed Actions . . . 32
3.9 Artificial Intelligence . . . 32
3.9.1 Random . . . 33
3.9.2 Rule Directed with Parameters . . . 34
3.10 Simulator . . . 35

4 Experiments and Results **37**
4.1 Generating Input . . . 37
4.1.1 Static Game Elements . . . 38
4.1.2 Dynamic Game Elements . . . 38
4.1.3 Picked Strategies and Configurations . . . 38
4.2 Experiment Participants . . . 39
4.2.1 Rule Directed AI . . . 40
4.2.2 Random AI . . . 41
4.3 Running The Experiments . . . 41
4.4 Results . . . 41

Conclusion **44**
Future Work . . . 44

Bibliography **46**

List of Figures **47**

List of Abbreviations **50**

A User Documentation **51**
A.1 System Requirements . . . 51
A.2 Running the Game . . . 51
A.3 Generating Configuration Files . . . 52
A.4 Server Configuration . . . 52
A.5 Main Menu Screens . . . 54
A.5.1 Introduction Screen . . . 54
A.5.2 Server Configuration Screen . . . 55
A.5.3 Join Existing Game Screen . . . 56
A.6 Game Screens . . . 57
A.6.1 Player Configuration Screen . . . 58
A.6.2 Cities Overview Screen . . . 59
A.6.3 World Map Screen . . . 59
A.6.4 Action Reports Screen . . . 60
A.6.5 Player Messages Screen . . . 62
A.6.6 City View Screen . . . 62
A.6.7 Building View Screen . . . 62
A.6.8 Send Attack and Support Screen . . . 67
A.6.9 Send Private Message Screen . . . 68

B Advanced User Documentation **69**
B.1 XML File System 69
B.2 XML Configuration 70
B.2.1 Game Resources 70
B.2.2 Unit Types 70
B.2.3 Nations 71
B.2.4 Event Tiles 76
B.3 Image Configuration 77

C File Attachments **78**

1. Introduction

Our thesis is about development of a framework for creating Massively Multiplayer On-line Real-Time Strategy[1] (MMORTS) games with persistent progress similar to browser games Travian[2] and Tribal Wars[3]. These games are being played on-line by large amount of players. The most common feature of these games is, that the players have to beat their opponents. This is typically achieved by either getting the highest score within set amount of time, researching a secret technology, or similar alternatives of a victory condition. Another feature of these games is, that players own cities. Cities are points in the world and every point has unique coordinates, where each player can own an arbitrary amount of cities. A world is usually a flat grid consisting of points representing a city or an empty ground. World map allows players to view cities all over the world. The visibility of details of enemy cities is minimal. Players can send attacks from their cities at enemy cities, but there is no control over military units during combat. The results of each combat are evaluated automatically by the server and all the participants are informed about the outcome. All the actions of the players are executed in real-time. Even when the player is not currently playing the game, everything is still active and the cities the player owns, can still be interacted with by other players. These games are known for their long life cycle of each game world (12-14 months).

One of the problems of these games is that the gameplay is very dependent on the amount of players actively playing the game. It is important to keep the population stable. However, the length of the life cycle implies that the player activity and world population changes according to many factors, such as holidays, different seasons of the year or weekends. Because of that, the amount of active players can drop to critical levels, leading to loss of additional active players and may cause a game world to end prematurely. If that case occurred, setting a new victory condition according to statistics of players would completely change the tide of the game, making it unfair to those who prepared towards a strategy that would help them win later in the game.

One possible way to fight this problem is to use artificial intelligence (AI) as a replacement for inactive players. There is very little related work for AI in MMORTS games. The main reasons are that the branching factor in these games is very high and the the provided information about enemies is incomplete. An implementation of AI in these games was seen in Travian, called Natars[4]. However, it is used to provide a victory condition instead of providing a substitution for inactive players. The AI only controls a tribe of Natars which has an unfair advantage over other playable tribes, for example, their units do not cost resources. Sometimes, this AI is present in the early stages of the game but does not attack other players and provides special benefits to those who capture their cities. Other implementations of AI in this MMORTS sub-domain have not been seen on live servers yet.

[1] `http://gaming.wikia.com/wiki/Massively_multi-player_online_real-time_strategy_video_games`

[2] `http://www.travian.com/`

[3] `https://www.tribalwars.works/`

[4] `http://travian.wikia.com/wiki/Natars` (Last Access Date: 3rd of May 2018)

In this thesis, we would also like to test whether it is possible to create an AI that is capable of playing any game from a certain MMORTS sub-domain on a decent level. Doing this will make the game world superficially alive without giving the AI an unfair advantage. This can help increase the success rate of new companies which are trying to start their first MMORTS server while they are not well known. Another goal is to seemingly increase the population in case of critical drops in the amount of active players which can stabilize the situation until new players join the world. The AI also opens new possibilities for MMORTS game worlds, where developers can experiment with new victory conditions and game mechanics.

Our main goal is to develop a framework which helps developers create their own alternatives of this type of game in a short period of time without complex development and testing. The implementation of the Server and Client is provided in libraries. The framework allows server administrators to configure the game worlds in a way of changing the world size or change the graphics of the world map displayed by Client. Our second goal is to develop an AI and test its capabilities.

1.1 Our Goals

The goals of this work are summarized in this chapter.

The main goal of our thesis is to develop a framework for persistent MMORTS games. This framework should be capable of creating games similar to browser games Travian or Tribal Wars. The framework will be bundled with a simple client. The client should be capable of providing basic visual representation of the game world and controlling the cities owned by a player.

Our secondary goal is to prepare an artificial intelligence which will be capable of playing the game on sufficient levels of credibility. It should not be provided by more complete information compared to player or any other kind of unfair advantage over players.

1.2 Structure

In chapter 2 Problem Analysis we will describe the reasons of why we have chosen one approach of implementing the features and mechanics and the AI over other possible approaches. We will also explain the importance and possible contributions of this work for the game industry.

In chapter 4 Experiments and Results we will show the results of experiments that have been performed using a simulator. These experiments consist of multiple runs of short matches where the main test subject is the Artificial Intelligence. The results show how minor changes to the parameters of the Artificial Intelligence combined with different games procedurally generated by the framework affect the efficiency of the AI itself. The results also show that the framework works as expected and how the parameters help the AI adapt to games which prefer different types of strategy.

User documentation is covered in Appendix A User Documentation. It will decsribe how the game can be ran, how to change the configuration of a server, how to connect to the server and how to play the game. It will be primarily

focused on describing each screen of the game and components on them. After reading the user documentation, it will be easy to start and play the game.

Advanced User documentation is included in Appendix B Advanced User Documentation. It will be focused on explaining how to create new games using the provided framework.

Conclusion - This chapter is about contributions of the entire thesis and possible options for future work and improvements.

2. Problem Analysis

In this chapter we will describe Travian as a representative of MMORTS games. We will briefly mention the background of this game and focus on the gameplay and mechanics. After summarizing information about Travian, we will describe what our framework should offer. Finally, we will talk about possible obstacles we have to consider when creating AI for the framework and the requirements it should meet.

2.1 About Travian

[Kiran Lakkaraju, 2018, Ch. 8.9.1: Travian was first released on September 5, 2004 by Gerhard Müller with Travian Games GmbH in Munich. The game is written in PHP and is available in forty-one languages. There are more than 5 million players on more than 300 game servers worldwide. In 2006 Travian won the Superbrowsergame Award in the large game category.]

Travian has gained massive community and the gameplay revolves around player interaction. Every year, the developers of the game create a new version of the game which brings something new compared to the classic version. This is one way to keep the attention of the community.[1]

For those, who played the game for years, tournaments are hosted every year. They are intended for the most skilled players. The participants have a chance to win special prizes.

Figure 2.1: Graphical representation of a village in Travian. The action bar on top shows village name, gathered resources, resource maximum, current production, tribe and offers multiple navigation and configuration buttons. The panel on the right side shows incoming attacks and units present in the city. Bottom bar shows in-game time and building and training queues.
Source: `https://www.kingdoms.com/`

[1] `https://www.travian.com/international/game/playerinteraction`

2.2 Key Components of Travian

In this section we will briefly summarize the key components of Travian. We will then describe each component in depth.

The main components are:

(K1) The game is played in real time.

(K2) The game world has a long life cycle.

(K3) The game progress is persistent for each game world.

(K4) Player can play the game as one of three basic tribes. It is possible to pick a different tribe in different game worlds.

(K5) Players can own an arbitrary amount of villages in one game world. All villages have bonuses dependent on picked tribe.

(K6) Each village has a set of buildings they can construct, upgrade or even downgrade and deconstruct. Each building provides different bonuses. Constructing or upgrading a building costs set amount of resources and set amount of time.

(K7) Resource management is an important part of the game-play from start to finish. There are four basic types of resources: wood, clay, iron and crop.

(K8) Military units take time to travel outside the village, but there is no control over the flow of battle during combat. The results of the combat are known immediately after units arrive to their target destination.

(K9) Village management is an aspect of the game that can be handled individually or collectively. In Travian, it is impossible to keep expanding one village indefinitely. To gain more influence and power, it is necessary to establish a new village or conquer a village of another player.

(K10) Strategy is the key to success. There are multiple strategies that can be played and they can yield different results on different game worlds.

(K11) The MMO aspect of Travian allows the game to place a big importance on player interaction. Player interaction combines all of the components mentioned above together and making them gain on importance. The most important example of player interaction in Travian are Alliances which are established by players and consist of players having the same intentions.

(K12) Travian as many others games offers a victory condition. As the game keeps going for long enough, normally for about 280 days, tribe of Natars enters the world. They own plans for a building that leads to victory. Players need to start attacking villages owned by Natars in order to retrieve these plans. Once a player manages to retrieve the plans, players from their alliance start working together on helping the player finish building this structure. This typically includes sending resources. They also have to protect it so that other alliances do not destroy it in order to buy more time to build their own structure to achieve victory.

2.2.1 Real-time gameplay

This concept of the game ensures that all actions performed by players are processed and executed as soon as they are received by the client. This mostly matters in terms of combat between players. It allows players to plan ahead and execute an attack that will catch their enemy unprepared. It also allows to execute a set of consecutive attacks where each attack has different goal, for example to lower the enemy defenses, destroy his defensive army and immediately after that, conquer the city of enemy by last few attacks.

2.2.2 Length of life cycle of game worlds

The length of the life cycle of a game world is typically about 280 days. However, it is possible to play on so called SpeedTx[2] worlds. The letter T can be replaced with numbers 2, 3, 5 and 10. The number indicates that the events on this world occur T-times faster in comparison to common worlds. On speed worlds the length of life cycle of the game world decreases drastically.

The shortest life cycle of a game world was running for 42 days. It was a test run of a Speed10x speed world on German servers in 2010.

2.2.3 Persistent game progress

Because of the length of life cycle of the classic game worlds, having a persistent progress is a necessity. When a player stops playing the game for one night or even couple of days, they are still able to log back in to the game and continue playing.

However, the villages of the player are present in the game even when he is not currently playing. This may lead to the player getting attacked or even annihilated by other players during his absence. Players who lose their progress by getting conquered prefer starting out in a new, different game world. The reason is, that older game worlds already has players who have already spent a long time playing the game and already own multiple villages or even formed powerful alliances with other players. This gives the player who just joined the world a very small chance of becoming one of the strongest players on given game world without getting conquered once again.

2.2.4 Available Tribes

[3] In Travian, each playable tribe offers different special features. The Romans, the Gauls and the Teutons are the three basic playable tribes in the classic version of Travian. However, there are two additional playable tribes in the special version, Travian Fire and Sand. These tribes are the Egyptians and the Huns. To keep it simple, we will only break down the features of the tribes present in the classic version.

The Romans are recommended tribe for a player who just started playing Travian. Their troops are balanced in both attack and defense. On the other hand, their defense against cavalry is the lowest of all tribes.

[2] http://travian.wikia.com/wiki/Speed_server

[3] https://t4.answers.travian.com/?view=answers&action=answer&aid=7

The special features of the Romans are:

- Simultaneous construction of resource fields and buildings.
- Higher defense bonus from city wall.
- Merchants can carry moderate amount of resources while traveling at average speed.
- Exceptionally powerful infantry, but below average cavalry.
- Training of troops is long and expensive.

The Gauls train units that are very strong defenders. Their cavalry is above average in terms of speed which makes them viable for surprise attacks, fast pillaging, even sending out a quick army of powerful defenders to support a nearby village. This tribe is also recommended for beginners as they are harder to beat when they are defending and they are capable of hiding their resources better than other tribes to prevent pillaging.

The special features of the Gauls are:

- Fastest units in the game.
- Moderate defense bonus from their walls.
- Merchants can carry an above average amount of resources while traveling at above average speed.
- Ability to hide more resources from the attackers to prevent pillaging of resources.
- Expensive siege weapons.
- Cheap settlers.

The Teutons are the most offensive tribe out of all three of them. Their units are the cheapest out of all tribes and can carry the most resources after a successful fight. However, the movement speed of their units and their fighting capabilities are worse than those of the Gauls or the Romans. This tribe is recommended to experienced players who play offensively.

The special features of the Teutons are:

- Their walls are very hard to destroy, but they also offer very little defense.
- Merchants can carry the most resources out of all tribes, but they also move at the slowest pace.
- Their troops are unequivocally cheap, fastest produced, and have the best resource carrying capacity.

2.2.5 Ownership of Multiple Villages at once

In Travian, players typically start out with one village. The players can construct and upgrade every building available in this village and they are also able to produce every resource and train every unit. However, buildings, building upgrades and units use up Population. The population needs one of the resources called Crop to be able to survive. Therefore there is a limit of Population a player can sustain in one village without external help. Because of that, the players are forced to expand their territory by getting more villages under their control.

There are two ways of getting more villages. One way is to found a new village. To found a village, it is necessary to train set amount of units called Settlers. To be able to train these units, the player needs to fulfill certain prerequisites. When the player is ready to get a new village, they pick an empty spot on the map where no other village is present. Then, the player sends out the Settlers and once they arrive to their destination, a new village is established and immediately becomes controlled by the player.[4]

Another way is to conquer an existing village owned by another player or a neutral village. Compared to founding a new village, this process is more complex but may yield higher rewards. First, the player needs to recruit an administrator. This unit has different name for all three tribes and the attributes and resource cost differs, but the use remains the same. To be able to recruit such unit, the player must fulfill certain prerequisites. A village of another player can only be conquered when only when specific conditions are met. One of those conditions is that the attacked village is not the capital and not the only village owned by the victim. Every attack which includes an administrator reduces loyalty of the enemy village by set amount of points. The maximum loyalty for every village is 100 points. To gain control over the new village, the loyalty of the targeted village must fall down to zero. The only way to bring the loyalty of a village back up is to construct and keep upgrading a building called Palace or Residence. After a village gets conquered, all units owned by the previous owner in that village will disappear and the walls that were protecting the village will get destroyed. Any research or tribe specific buildings will also disappear. It is possible to conquer a village owned by the same player who sends the attack to perform the reset mentioned above. This approach enforces many different strategies which may vary from player to player and it can also involve actions of entire alliances. [5]

2.2.6 Buildings and Resource Fields

Resource fields[6] are the most basic way of harvesting resources in the game. These fields are located outside a village. For each resource out of the four available resource types in the game, there is a unique resource field which can be used for the production. These resource fields are:

- Woodcutter
- Clay Pit

[4] `https://t4.answers.travian.com/index.php?aid=108`
[5] `https://t4.answers.travian.com/index.php?aid=101`
[6] `http://wiki.kingdoms.com/tiki-index.php?page=Resource+Fields`

- Iron Mine
- Cropland

Wood, clay, iron and crop is used to construct and upgrade buildings and resource fields. However, wood, clay and iron can be used to research and train troops, while crops is used to feed troops and the population in the village.

Figure 2.2: Graphical representation of resource fields in game. In the middle of the picture, there is a village seen in figure 2.1. Everything around the village are the four types of resource fields.
Source: http://blog.travian.com/2014/08/alpha-diary-settling-has-the-highet-priority/

Buildings[7] are divided into 5 categories.

- Infrastructure
- Military
- Resources
- Wonder of the World
- Special Buildings

Infrastructure buildings is a category which is focused on storing, trading and hiding resources. Additionally, these buildings are intended for spreading the influence of the village, including founding new villages around the world. It is possible to improve the construction speed and the durability of all buildings in the village. Durability of buildings matters only during defense against enemy catapults and rams.

The strength of army and training of individual units is maintained in buildings from the military category. It is possible to research new troops, even train the units to be stronger and more agile.

This category of buildings increases the base resource production that comes from resource fields.

[7] http://wiki.kingdoms.com/tiki-index.php?page=Buildings

Buildings from this specific category are only available in village which starts constructing Wonder of the World. This building is typically used to determine a victory of an alliance in the game world. The category includes the Wonder of the World itself. However, they also allow building multiple specific buildings to store more resources and build stronger walls around the village to make it easier to protect the Wonder of the World from getting destroyed by Natars and enemy alliances.

While the resource fields and all buildings from the four categories of buildings and identical for each tribe, buildings from Special Buildings[8] category differ for all tribes. This category provides tribe specific walls to protect villages from incoming attacks. Moreover, there is one unique building in this category for each tribe which adds a tribe specific bonus to the village.

An example of how the buildings are represented in the game can be seen in figure 2.1.

2.2.7 Resource Management

Many RTS games utilize resource management and Travian is not an exception. As we mentioned above, there are four resources that can be harvested by the player. There are multiple ways the resources can be obtained.

The most common way of obtaining resources is to keep upgrading resource fields, which are part of each village. The resources from these fields are passively gained over time. The production of resources does not change depending on whether the player is actively playing the game or he went off-line for a couple hours. The only moment when the production of resources from resource fields stops, is when the resource hits the maximum limit. The maximum limit is controlled by a building, typically a warehouse or granary. The efficiency of resource fields can be affected by constructing and upgrading buildings specifically oriented for production of resources.

The production from resource fields can be increased even further with bonuses from special currency purchased for real money or from bonuses of oases. An oasis is a special type of tile in the world which can randomly spawn close to a village owned by a player. After raiding an oasis and conquering it, the village that performed the final attack receives a bonus to production depending on the type of oasis.

Besides the passive ways of resource acquisition, it is possible to actively trade resources with villages owned by other players or with non-player characters. Trading is performed from building called marketplace. Players have the ability to send resources from one village to another village as reinforcement without requesting anything for exchange.

The most active and strategically the most difficult way of obtaining resources is by raiding enemy villages. Military units have a variable amount of resources they can carry after a successful raid. However, to be able to raid enemy villages for resources, player has to destroy the defensive units guarding the village. After all defensive units are eliminated, the player has to raid the village periodically to make sure the player does not gain enough resources to train new defensive units and prepare to defend against next raid. There are many factors that can

[8] `http://wiki.kingdoms.com/tiki-index.php?page=Special+Buildings`

stop a player from raiding a village. As an example, we will mention two different ways. It can be done by requesting a defensive reinforcement from players from alliance to provide protection from incoming raids until the threat is eliminated or by diplomatically talking to the player who is raiding the village.

2.2.8 Military Combat

Because the combat in Travian is automated and the players can only see the time of arrival of units to their destination and the outcome of the battle afterwards, it is crucial for the player to know how the outcome is calculated in order to be able to maximize the efficiency at which they can use their army.

Each unit has four attributes which are used in combat:

- Attack Power
- Defense Power against Infantry
- Defense Power against Cavalry
- Carry Amount

[9] In the calculations, siege units like catapults or rams, are classified as infantry unit type. When a player attacks a village with cavalry and infantry units, the total attack power is calculated as follows:

$$AP = \sum_{u \in U} u(C) * u(AP)$$

Where U is a set of different kinds of attacking units. Element of the set u, contains information about the count *(C)* of that kind of unit, their attack power *(AP)*, defense power against cavalry *(DP[C])* and defense power against infantry *(DP[I])*.

When a village is defending against an attack, the situation becomes more complex. We will calculate the attack power of attacker granted by cavalry *AP[C]* and infantry *AP[I]* by including only the information of units that belong to these categories. Let *P[C]* be proportion of *AP[C]* and *AP*. P[I] is proportion for infantry units. The sum of these two proportions will always be equal to one and one. Using these variables we can calculate the defensive power *DP* of the besieged village. Let *U'* be the set of different kinds of defending units.

$$DP = \sum_{u' \in U'} u'(C) * u'(DP[C]) * P[C] + \sum_{u' \in U'} u'(C) * u'(DP[I]) * P[I]$$

The value of *DP* is further influenced by the level of walls in the village. Therefore, this case is relevant for case when the walls are not constructed at all.

Comparing values of *AP* and *DP* will indicate the winner of the battle. In case *AP* is greater, attacker wins. If the value of *DP* is higher, it is the defender who wins. All units of the defeated player die. The amount of survivors of the winner is calculated from the proportion of the power of the defeated village and

[9] `https://wbb.forum.travian.com/thread/75248-combat-system-formulas/`

the power of the winner. This proportion is applied to the total number of units of the winner.

When the winner is the attacker, the units that survived will take all resources the defender has in his village that are not hidden from attackers. The amount of resource that can be taken by each unit is limited by the carry amount attribute.

There is a very small chance that the values of AP and DP are equal. In that case, all units of both attacker and defender are killed.

2.2.9 Village Management

When players receive their first village after joining a new game world, they have to focus on multiple aspects of the game at once. They need to create a strong army to be able to protect their own village, they need to have at least an average passive production of resources from resource fields, they need to perform research of new units and prepare their city for capturing or founding a new village.

After getting their second village, they usually start thinking about utilizing their new village to fulfill a specific purpose. Some villages become filled with defensive units, ready to be sent to alliance members in need of help. Other villages become offensively oriented and are used to pillage surrounding villages and making enemies defenseless. Sometimes the village becomes focused on production of resources or gets fortified by walls and serves as a point of strategic advantage on the map in case of war between alliances. There are many ways a village can be used and specialized.

2.2.10 Player Strategies

Strategy is essential to become a successful player of Travian. The play style is very often the main factor that determines the final strategy. There are many strategies that cover all phases of the game from start to finish. However, they are usually not very successful, because the success is dependent on what players are near villages owned by a player trying to follow a strategy guide. For example, using a strategy that revolves around domination of surrounding villages and focusing on raiding and pillaging is very ineffective when all villages around the player are owned by members of his alliance.

Because it is very hard to create such strategy, there are entire threads on forums, discussions, even entire websites dedicated to trying to create new strategies. Most of the better strategies consist of use of smaller strategies depending on the flow of the game. In this section we will describe some of the small strategies and how they are performed.

Defensive Strategies

Attack Dodging is a strategy where a player who is getting attacked orders all units to move out from the village until the attack is done. This is being done when a player is not sure if he can defend against the attack or the player wants to spare his offensive units which are ineffective in defense and would die easily. This can be achieved by sending the units to attack or reinforce other village.

However, that can make the army become unavailable for a long period of time because the troops need to complete the whole trip. This trip can take about one hour on short distance.

To prevent the troops from having to finish the whole traveling sequence, a player can perform a dodge. In Travian, it is possible to revert an order to attack or reinforce a village within the first 90 seconds. Using this mechanic, it is possible to send the army away a couple seconds before the attack is about to happen and once the village is safe again, a player can revert the order and the units will safely return back to village within seconds instead of minutes to hours.

Timed Counterstrike also utilizes timing to achieve desired results. Because it is possible to see the time of the impact of incoming attack by both players, the defender can see how long it takes for the enemy to travel the distance between those two villages. The defender can prepare an attack that will arrive one second after the units of the attacker arrive back to their home village. This way the attacker will only have one second to send his offensive units away to dodge the counterattack, making the defender turn the tide of the battle.

Double Conquer becomes useful when it becomes too difficult to defend against a conquering attack. When a village gets conquered, the loyalty becomes zero. It needs a new infrastructure to gain back loyalty to prevent it from being conquered again. This is used in this strategy. A player can give up on the village and then conquer it back by timing a conquering attack on their own village so that the attack arrives immediately after getting conquered by the enemy. Losing a village however causes it to reset all the researches, troops and army upgrades as mentioned in subsection 2.2.5 Ownership of Multiple Villages at once.

Oasis Protection is important because it grants bonus resources. Additionally, oases carry ten percent of the resources owned by the main village and the amount is replenished every ten minutes. Because oases do not have walls or any other way of increasing the defensive capabilities, the best way to protect the oasis is to attack an oasis and take the resources a short time before the enemy.

Defense Villages are involved in almost every strategy that tries describing every phase of the game from the first day until the end. These villages are a specific specialization mentioned in subsection 2.2.9 Village Management. This type of specialization is usually unit type specific. It can focus on defense against infantry units or cavalry units. How many of these villages a player establishes and when he starts preparing them is entirely up to the player and the flow of the game. The goal of these villages is to provide reinforcements for other villages owned by the same player or entire alliances. It is recommended to convert all villages into defense villages as soon as a player from the same alliance starts building a World Wonder.

Offensive Strategies

The Blacksmith has a name after a building. This building is used to upgrade combat capabilities of troops trained in the village where upgrades were

performed. The strategy revolves around upgrading offensive troops as much as possible. The bonus from upgrades does not apply for units that are present in the village as reinforcement. This strategy is not suggested as defensive strategy, because a player cannot spread influence or capture cities using defense and the likelihood of getting attacked, when the area around a player is eradicated by force, is small.

Fake Attacks are important part of offensive play in Travian. We already mentioned that attacker and defender can see the time of arrival of an attack. What the defender cannot see is what units are coming. Therefore it is possible to assume what types of units are in the incoming attack but the quantity cannot be guessed. This opens a possibility to fake attacks. Fake attacks usually consist of only 1 unit of chosen kind of troops. There are multiple reasons why a player wants to fake an attack. Most common examples are:

- To alert an enemy and to make him aware about the activity of the player to make the enemy reconsider their actions.
- To demoralize a player. Waves of fake attacks that take hours or even days to arrive can put lots of pressure on a player receiving the fake attacks and he might expect them to be conquering attacks coming from a strong player. This situation can make the player completely quit the game world leaving his villages unprotected.
- To force a player to spread his defenses and ask for reinforcements into multiple villages at once. Faking attacks on four different villages and attacking only one of them results in significant reduction in lost units during the real attack. This way it is easy to hide the real target of attack and the victim might lose a village without being given a chance to properly protect his village.

When sending out fake attacks, it is the best idea to send slow units like catapults because travel speed of attacking units is defined by the slowest moving unit in the attack. This can help hide even more information from the defender. Additionally, it is much easier to spot a fake attack if the travel speed appears to be too fast or equal to a movement speed of a defensive unit.

Cleaner waves are sent as attacks at enemy villages with purpose to kill all or most of the defenders in the village. A cleaning wave should always be accompanied with rams to lower the enemy walls making their troops less effective. Sending a cleaning wave without rams would result in much greater loss of troops in the attack. Sometimes the difference is so big that it can determine if the attacker manages to kill all defenders. Optionally, cleaning waves should contain some catapults to bring down infrastructure, however, they are usually sent in separate attacks as part of the waves.

The Conqueror is based on trying to conquer every village around the player which seems appealing based on the displayed population. Capturing a village owned by an active player can be difficult, because the village might receive reinforcements from alliance the player belongs to. When picking the next target

to be conquered, it is important to consider the distance between both attacker and defender, how many villages the defender owns and how far away they are from each other to be able to predict the speed at which the targeted village can be reinforced. As an offensive player it is crucial to pick villages with as many crop resource fields as possible so the new village can feed as many troops as possible.

Once the target has been decided, the next step is to start preparations for the attack. Each successful attack containing an Administrator decreases loyalty of a village by 20 to 30 percent. The chances of successfully capturing a village are much higher when the village is captured in one run. This can be achieved by training and preparing multiple Administrators responsible for lowering the loyalty of a village. On average a player needs between 5 to 7 Administrators to completely conquer a village. In the early stage of the game, when it is too hard to be able to train multiple administrators in single village, it is a good idea to ask alliance for help in decreasing the loyalty. However, from mid game onwards, it should be easy enough to have more Administrators than required.

When the preparations are ready, it is time to start planning and executing the attack. Typically, a player needs to perform these steps in order to successfully conquer an enemy village:

1. Estimate the number of successful Administrator attacks required to fully conquer the village.

2. Send scouts to reveal the defense present in the village while the player is not alarmed.

3. Attack when the defending players is likely asleep to lower the chances of the defender to get reinforcements.

4. Attack the village with catapults in order to destroy the infrastructure making the village more vulnerable to being captured.

5. Send the Administrators in separate waves of attacks to make sure they are performed as quickly as possible.

The last two steps include attacking in waves. The waves can differ depending on likes of the attacker and the assumptions about the enemy. The most common way of forming the waves is as follows:

1. Send 4 or 5 waves from the main village. Make the first few waves fake waves. The remaining waves should consist of cleaner waves and Administrators. Catapults should be ordered to attack infrastructure.

2. Send another 3 waves from each village containing Administrators. It is suggested to time those waves to be about one minute apart. The reason for that delay is to make sure the incoming attacks do not hit and kill the units sent by the attacking player. This can happen in case the village gets captured sooner than expected because of random chance or miscalculation.

3. Make sure that each wave with Administrators has enough offensive troops to prevent the Administrators from getting killed by units from dodged performed by the defender.

Immediately after conquering a village it is important to build back the infrastructure to gain back the loyalty and to get as many reinforcements in the captured village to prevent it from being captured back.

2.2.11 Player Interaction and Alliances

In the early stage of the game, there are usually many alliances. They can be either small or large, however they are formed by random players who are grouping together without knowing too much about each other. Majority of these alliances are typically dissolved because they get defeated by larger alliances who make the strongest players join them. Later in the game, these stronger alliances tend to merge with other alliances to form bigger alliances. Eventually, they become multi winged alliances, which are known as meta alliances. These alliances form within each quadrant of the game world and many of them do not even make it to the end game because they fight other meta alliances. The winning meta alliance in each quadrant is then able to join the race in building the World Wonder in the final stages of the game.

The most dangerous type of alliance is the one formed by players who played together on previous game worlds. These people know each other, know majority of mechanics, tricks and dangers of the game and are not willing to accept an inexperienced player to join their ranks.

Alliances make collective decisions and then they cooperate to achieve their goals. However, players are just people and a behavior is not restricted by the game rules. Therefore it is possible for some players to be official members of one alliance, but their intentions are to spy for a different alliance that plans an ambush. Spies are a very common strategy in Travian and it is hard to find a spy in an alliance since they look exactly the same as every other member of alliance. The alliances that are formed from previous game worlds tend to place a spy in alliances even immediately after launch of a new game world.

Diplomacy is a very common factor which affects the outcome of the game. In the early stages of the game it is not recommended to form a friendly alliance with as many players as possible so there are players that can be raided for resources and villages to conquer. Later in the game during the mid game phase, it is suggested to start forming alliances with those that managed to become strong to be able to create even stronger alliances. This still leaves the weak alliances around for raiding and capturing. By the end of this stage of the game there should be many wings under the strongest alliance. A stage right before the end game begins is called late game. During the late game stage of the game players want to finalize their diplomatic agreements and prepare for end game and World Wonder. Because of that, diplomacy in end game is typically non-existent.

2.2.12 Victory Condition

The first player to construct their World Wonder to level 100 is declared to be the winner of the game world. The players with the highest total population and the most successful attacking and defending streaks are also mentioned in the declaration.

After the winner is declared, the game stops and players can no longer build,

trade or initiate attacks. After a period of time, the next game world begins, and the game starts from the beginning.

2.3 Main Differences of Framework

In this section we will describe differences of our framework in comparison with components described in section 2.2 Key Components of Travian. It is worth mentioning, that compared to Travian, player does not start with a **village**, but with a **city**. However, the functionality of the entity remains the same.

2.3.1 Generic Victory Conditions

The framework does not come with a default victory condition. The game world goes on until it is stopped manually. The reason behind that is, that there is no victory condition we know of that would work for all games created by the framework.

Examples of generic victory conditions would be:

1. Certain amount or proportion of players are left with no village to control.
2. A player reaches specific score milestone.
3. A player managed to capture or establish certain amount of cities.
4. World runs for certain amount of time.

In simulation we will describe in **?? ??**, we used a combination of the first and last victory conditions mentioned above, which worked well, but had some problems which can be seen in chapter 4 Experiments and Results.

2.3.2 Replacement of Consumable Resource

Travian utilizes a resource called **Crop** to maintain the army and workers in buildings. The resource is passively gained from resource fields, and the army and workers decrease the production. Once the amount of crop required to maintain the village exceeds production, the amount of stored crop starts decreasing and the troops will starve and eventually die. Death of troops would cause the required crop to maintain the village to get back to acceptable levels. The amount of crop required to maintain a village is affected by reinforcements.

Our framework does not use this method of limiting the amount of troops and workers in one city. Instead, we directly limit the amount of population each city can reach.

The difference is, that upgrading buildings and training more units increase the population of the city. Trying to upgrade a building or train units that would bring the population above the value of maximum, is an action that cannot be performed. However, receiving reinforcements from other cities does not affect the amount of population.

Downgrading buildings that require population and losing units in battle brings down the amount of used population.

2.3.3 Founding New Village

As mentioned in 2.2.5, Travian allows training units called Settlers. These are being sent to empty positions on the map to establish a new base. With greater distance, the traveling time of the settlers increases.

In the framework, there are no units that have similar use compared to Travian. However, it is possible to open world map interface to find an empty spot. When a position is picked, it is possible to establish a new city using resources in the city.

The action of founding new city in the framework is instantaneous and the resource cost increases with greater distance between the two positions on world map. The process of founding resources is covered in user documentation in subsection A.6.3 World Map Screen.

2.3.4 No Loyalty

The framework has no support for such feature. Every city can be captured after first successful attack containing a unit capable of capturing enemy cities which survived the fight.

2.3.5 Capturing Enemy Cities

In Travian, it is impossible to capture a village which is marked as capital city. This feature is not present in the framework and a player can lose all of his villages. When a player has no village left to control, they are defeated and there is no action left to be performed other than starting a new city under new name.

When a player captures a city, the player inherit all progress that was done in the city by the other player. Moreover, all ongoing actions are also inherited with all units trained in the city that are still alive. The only thing that changes is the ownership of the city.

2.3.6 No Research

This feature is not available in the framework at all. There is no similar feature that would be capable of mimicking the same behavior.

2.3.7 Possibility of Controlling Multiple Nations

Compared to Travian, where capturing a village of different tribe, it becomes a city of the same tribe the player who captured the village is using.

In the framework, it is possible to capture a city controlled by different nation and gain the ability to construct the buildings and units specific for the nation of captured city. This adds a new component for strategies that can be used while playing the game.

2.3.8 Resource Fields

Passive income of resources is handled using buildings instead of resource fields found in Travian. Additionally, there are no buildings that provide bonus

to the total passive resource production from buildings in the framework.

2.3.9 Exchanging Resources

Travian uses merchants to send resources to other cities. They are also used for performing exchanges of resources. Merchants require certain amount of time to travel between cities to deliver the resources from one city to another.

The framework uses a different approach. It is possible to exchange resources from a building directly. The exchange happens instantly, however, the exchange ratio does not change depending on the demand of resource.

2.3.10 Combat Formulas

When a player attacks with only one type of unit, the calculations are just like in Travian.

The calculations when a player attacks with multiple types of units are different. In such case, the attack happens in waves. Every wave contains only one type of units, resulting in attack having the same amount of waves as the amount of unique unit types of attacker.

Because each wave contains only one type of units, the calculations become the same as for the case of one unit type combat in Travian. The order in which the waves are executed is defined by the order of unit type definitions described in subsection B.2.2 Unit Types of attachments.

2.3.11 Canceling Actions

While Travian allows canceling construction and military orders, the only action that can be canceled is to cancel the order of reinforcements to protect another city and to return them back to the city they were sent from.

2.3.12 Events Instead of Oases

In the framework there are no oases that can be captured to provide bonuses for a city they were captured from.

As a replacement, there are events that can appear around the game world. Events appear on the world map and can be seen by everybody. They can be attacked and supported like every other city. Events are not controlled by artificial intelligence or another player. They are passive and they are not a threat and they only prevent players from establishing a new city on the position where the event is placed. On the other hand, attacking the events can yield extra resources, but they can be very heavily guarded by neutral units.

They can also serve as a trap, when players decide to send reinforcements to the specific event tile which will make it more dangerous when a player decides to attack it, especially if the event can only be attacked by a special type of unit.

Events are further described in subsection B.2.4 Event Tiles of attachments.

2.3.13 Upgrades of Military Troops

This feature is completely missing from the framework and there is no replacement for it.

2.3.14 Alliances

Because there is no default victory condition, alliances are also missing. There is no reason to team up.

2.3.15 Artificial Intelligence as Player

In Travian, the only villages that are controlled by AI are villages of Natars. These appear in the end-game phase of the game as part of the preparations for the World Wonder leading up to the victory condition and the end of game world.

Our framework allows adding cities which are controlled by generic AI immediately after creating the game world. It is possible to make the AI controlled cities appear periodically so that individual AI controlled cities have different progress and can be attacked even by new players joining the world.

How the AI works and what behavior of the AI is provided by the framework will be covered in section 3.9 Artificial Intelligence.

2.4 Used Technology

Our goal is to be able to see whether it is possible to create a generic AI. We also wanted to have a simple client-server application. This played a vital role in choice of the used programming language.

Because of that, the framework is running on .NET platform and C#. The main reason of choosing this platform is that it comes with Windows Forms class library which allows high variety of customizations and custom event handlers. It was used to create the first client to visually represent and allow the users to play the games created in the framework.

We have successfully avoided using external database. Instead, we used .NET language integrated query (LINQ) combined with instances of generic classes. There were two main reasons for this approach. One was to prevent the in-game information to be loaded in memory by both external database and the server. The second reason was to make running the server as simple as possible. However, it would be much easier to manage the game data with an external database such as MySQL.

2.5 Goals Breakdown

A. Main goal. Create a Framework for MMORTS games which allows to:

1. Define new and modify existing nations
2. Add and modify unit types
3. Add and modify resource types

4. Add and modify available units for each nation
5. Add and modify available buildings for each nation
6. Modify the price of establishing a new city
7. Modify the starting resources of each city
8. Add and modify events which can appear on the world map
9. Perform all the modifications listed above from XML documents
10. Modify the graphics used by Clients without modifying the Client itself
11. Play any game created using the Framework using the provided Client

B. Secondary goal. Develop an Artificial Intelligence which can:

1. Play games created with the Framework
2. Build, upgrade, downgrade owned buildings
3. Train military units
4. Attack surrounding cities of other players
5. Defend against attacks of other players
6. Establish new cities
7. Conquer cities owned by other players
8. Win the game.

3. Implementation

In this chapter, we will cover the implementation of algorithms, features and mechanics of the framework and game elements. We will also include possible improvements of these algorithms.

We will discuss the choice of networking protocol and how it is used to transfer game data during communication of server with client, including an example. Later, we will describe how we implemented save game system and how we load different configurations of the game elements.

Then, we will talk about the data stored by the server. A description of world map, including how sharing of map information with clients is handled. We will describe an algorithm that chooses positions on map for new players, why the approach is important and how it is related to placing new events on the map.

We will define timed actions and how we implement them. Then, we will list what artificial intelligence is present in our framework and how the AI works. This will include definitions of action and selection of actions for AI.

We will conclude this chapter with a description of simulator, which utilizes the implementation of the game to provide a testing environment for generated games and the created AI.

3.1 Network Protocol

The server is using Transmission Control Protocol (TCP) to establish connection with clients. The frequency of sending packets between server and client is not high. Additionally, latency is not important part of games that can be run by the framework. Therefore, TCP suits our needs better than UDP.

The game is played in real-time, but once the client receives information about currently viewed city, everything is being calculated without help of the server. The next packet containing new information from the server is received again once the player performs some server-side action.

Because the client relies on receiving every packet from the server after each performed action, while the communication between client and server is not very frequent, TCP fits our purposes better than UDP.

3.2 Data Serialization

The framework utilizes two types of serialization provided by .NET libraries.

1. **Binary Serialization**
2. **XML Serialization**

Binary serialization is an important part of the framework which allows converting objects of the game into binary information that can be stored for later use or sent to another system. It is used in communication between server and client that will be described in section 3.3 Client-Server Communication, where

both client and the server are required to support binary serialization. It is also used to save and load the game progress described in section 3.4 Saving the Game.

XML Serialization is used to load and store definitions of configurable game elements covered in section B.2 XML Configuration of attachment called Advanced User Documentation. The process of XML serialization is used exclusively by the server.

3.3 Client-Server Communication

The communication between server and client is essential in the framework. Both client and server use string messages to specify their communication. The client sends only string messages that use UTF8 encoding. The server is capable of sending a serialized object containing string message and another serialized object relevant to the type of request received from client. The string with relevant object is encapsulated in our implementation of **Packet** class.

The objects are serialized by the server using binary serialization to make it possible to transfer all information over network. After a client receives the response, client proceeds to deserialize the packet, identifies the string message and if required, deserializes the appended object to expected class or type.

In future, it would be better to use request IDs instead of strings, because string comparison is slower. However, the string usually contains additional information about the request. The additional information is passed using a chosen separator which is >. The typical response of a client is in format *COMMAND>INFORMATION*.

We will describe an example of the communication when client sends a request to retrieve a new version of world map.

1. The client sends a string request *GETNEWWORLDMAP>103 104*. The message before the > symbol is the identifier of the request. The values that come after > are the values for the request. In this case, the client is asking for new information about world map around city on position with coordinates X = 103 and Y = 104.

2. The server receives the request, checks if it contains the symbol > and splits using the symbol.

3. The server proceeds to check if the request is a valid string message.

4. The server identifies the values and prepares required information about the world map for the answer. Then, the information information is serialized and inserted into an instance of **Packet** class. The packet also receives a command used for response *NEWMAP* and values for the command. A response contains three values, which are the total size of the map with the X and Y coordinates of the upper left corner of the part of map the server sends data about. The string message contained in the **Packet** is *NEWMAP>200 98 99* and the appended serialized object is a two-dimensional array of instances of **BasicTileInfo** class. The purpose of the class will be covered in subsection 3.7.1 Obtaining Map Information. The server serializes the **Packet** and sends it to the client.

5. The client receives the response, deserializes the packet and identifies the string message. Based on the information received in the string message, the client knows that the appended object contains a 2-dimensional array of **BasicTileInfo** and deserializes the object accordingly. With the information provided, the client stores the information and displays the world map.

The server stores information about currently connected clients in instances of **ClientDetails** class. The objects store information such as the name of the player and the TCP socket which is used for communication.

Static class **CommConstants** contains constants used by both client and the server. It contains information such as size of buffer, number of port the server is listening on, message used by client when trying to establish connection and the message the server answers with. The list of all commands can be found in class **Commands**. Any command that is not listed in this class and is used in communication will be handled as undefined command.

3.4 Saving the Game

Currently, the save game system only allows maximum of one save game file. Implementing more save slots is not demanding. However, in a game with very long life cycle, it typically does not make sense to create and continually manage more than one saved game progress.

Games are saved using binary serialization process so that it keeps all used data structures (e.g. Dictionary) intact. XML serialization process does not support serialization of some of the more complex data structures by default. Another reason why we have chosen binary serialization over XML serialization for save game system is that we do not want to allow easy manipulation with saved game data by a user who does not have enough experience with programming.

3.5 Loading of Game Configuration

To load and store XML documents, we use XML serialization technology provided by .NET libraries. It is important to mention, that this method does not support serialization of queues and some other data structures by default.

XML documents are used to add, modify and remove game elements such as buildings, troops, types of in-game resources, types of units, game events or entire nations that can be played. The documents are very important part of the framework and in case they are missing, they are automatically generated back after an attempt to start the server.

Because the contents of files generated by XML serialization technology are easy to read by human, they can be modified by any file editor. How to properly edit these files, how the folder hierarchy works and what exactly can be configured can be found in section B.2 XML Configuration of attachments. Different configurations of the XML documents will be intensively used for multiple runs of simulation during experiments.

3.6 Data Storage

The runtime data containing information about players, their progress and the status of the entire game world is stored by the server. The server distributes the data to clients depending on their requests. However, clients can only perform requests to change the obtained data. The server will then perform required checks, perform the change to the data if possible and sends the modified data to the client. This way, third-party modifications done to client data cannot affect the data stored by the server.

What data the server stores and how they are related to each other is covered in the source code provided in Appendix C File Attachments in *ServerImplementation Namespace.*

We will briefly summarize the most common data structures used to store game elements.

- **World Map** is a two-dimensional array that can contain classes extending **WorldMapTile** abstract class.
- Players, buildings, units and actions of units, such as attack, support, return, are stored using generic **List** class.
- Screens of game client are stored in **holder** classes that store each formular and allows switching between them without having to display more than one screen at the same time. Each screen is described in section A.6 Game Screens in attachments.
- Building upgrades, downgrades and unit training are stored in generic **List** class, but the list is being treated as queue. This is being done because of the limitations of XML serialization process.
- World events and cities are extensions of **WorldMapTile** abstract class. The **WorldMapTile** class includes information about basic features of a tile on a map. These include name of the owner, nationality of the tile, available buildings, units, resources and production. A city provides very little additional features making it identical with **WorldMapTile** abstract class. However, events have ability to regenerate units, fighting the event yields additional resources to attackers and the rewards and units can be generated based on predefined sets of rewards and units including their amounts.

3.7 World Map

The world map has been implemented based on an example of old representation of map in Travian, with an example shown in figure 3.1.

3.7.1 Obtaining Map Information

Client sends a request to retrieve an information about part of map. The server processes the request and sends the client information about surrounding

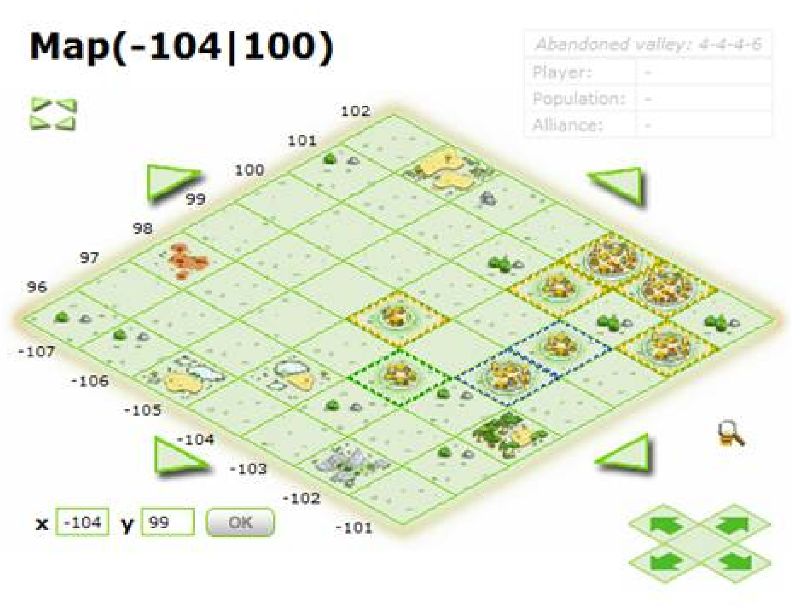


Figure 3.1: An old visualization of map in Travian. The controls and layout were used as an example when designing map for our framework.
Source: `http://www.glenzippo.co.uk/travian/Hero2.htm`

tiles on the map. The information is passed using instances of special class called **BasicTileInfo**.

This class stores information about the position of the tile in the game world, name of the owner, nation of the city and the amount of points of the city. In case of event tile, it also contains the description and name of image specific for the type of event.

Empty tiles in the world are represented with null value. Wrapping it in instance of **BasicTileInfo** allows us to store the information about the position of the tile.

However, cities and events contain lots of information that we do not need to send to the player. Having to serialize and send data about each city is unnecessary and it would result in much higher consumption of server resources when communicating with each client. Because of that, we have decided to include only the necessary information in instances of **BasicTileInfo**.

We mentioned that the client only receives part of the map, which is also important to reduce the network traffic between client and server, especially since the information about surrounding positions on the map is constantly being updated. The part of the map received by the client is stored in a two-dimensional array of this information. Whenever a client encounters a null value when trying to display the map to player, it immediately sends another request to the server asking for another part of the map. To be able to perform the update easier, checking for null values in stored array of world map information is always performed one step ahead. When the client receives the new part of map, it adds the information to the two-dimensional array.

3.7.2 Positioning Algorithm

In section 3.6 Data Storage we described the data structure used by this algorithm. In this section, we will cover the algorithm which uses the mentioned data structure and how it works.

We already mentioned that the life cycle of each game world can be very long. However, new players can join the server as long as there is an empty position where the player can start. Here we encounter a problem of placing a new player with small city in case there are already players with multiple cities.

Placing the player randomly in the game world would be problematic especially in the early stages of the game world, when there is not enough players. First few players in the game world would appear far away from each other. In the later stages it still would not prevent the new players from being placed close to players that have been playing for a few months, which would give the player no chance to be a part of the competition. Another problem is, that players should be grouped depending on their progress and when they joined the world, while keeping them close to players that have joined only a couple days earlier.

A solution to this problem is, to start adding new players in the center of the map and keep spreading them towards the edges of the map in circles, similar to growth rings of a tree. This way the players will be sorted from the oldest players, in the center of the map, to the player who recently joined, closer to the edges of the world map. For this, we will need a data structure that simulates the growth rings of tree with respect to cardinal directions. Because of the cardinal directions, the data structure is implemented as a special kind of tree structure.

The construction of the data structure is performed at the start of the server. An algorithm tries to find the center of the world map. There, it creates a **block** covering tiles in a square of configured dimensions. This block becomes the root node of the data structure. To make it simple to explain, let the dimensions of the **block** be set to one, making it cover exactly one tile on the world map per block. From here, we will refer to each **block** as a **tile**.

Each tile, which is not on the edges of the map, has eight other tiles around it. These tiles are in eight different cardinal directions. These are the eight child nodes of the root node of data structure.

Every child node has up to three child nodes. These are:

1. **MiddleBlock** has all three children, one is in the same cardinal direction as parent node and the other two are the left and right branch neighbors of the cardinal direction. For example, if the parent node has cardinal direction *NORTH*, the MiddleBlock will have children from *NORTH*, *NORTHEAST*, and *NORTHWEST* from the perspective of parent node.

2. **LeftBlock** only has one child node, which is the left branch described in **MiddleBlock**.

3. **RightBlock** only has one child node, which is the right branch described in **MiddleBlock**.

The final data structure covers the world map as shown in figure 3.2. If we take *EAST* marked by letter **E** in the picture, the part of map marked with **M** is covered by **MiddleBlock** nodes. The **L** and **R** are covered by **LeftBlock** and

RightBlock nodes. As we can see, **NE** and **E** partially cover the same piece of map.

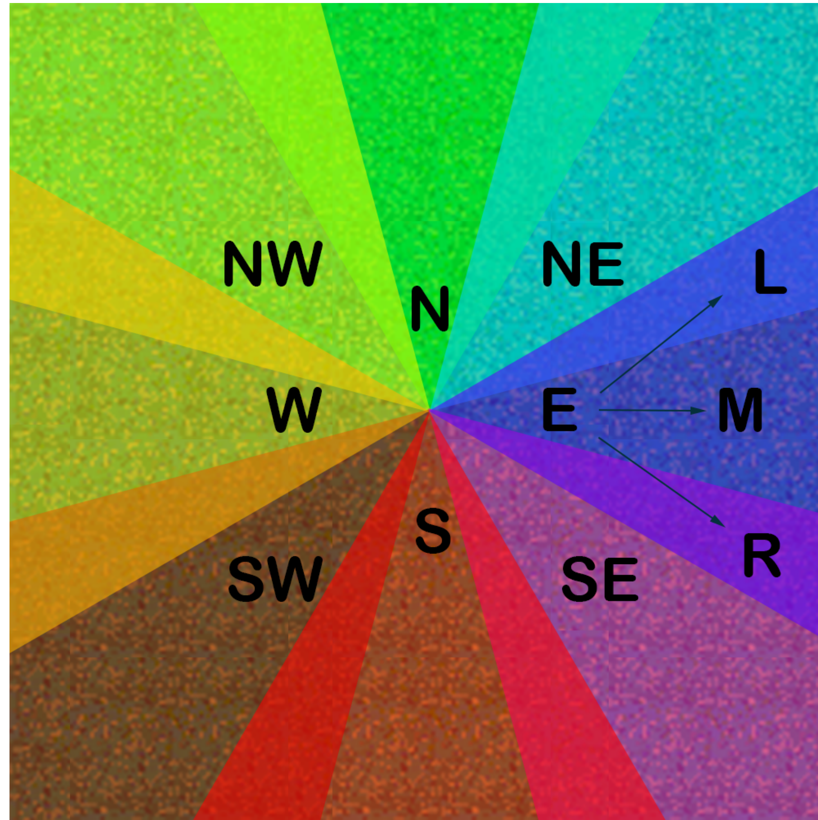


Figure 3.2: Visualization of the data structure used by positioning algorithm. The whole picture is entire map with highlighted cardinal directions. The colors represent children of each node. The middle of the map is the root node of the data structure.

It is worth mentioning, that the representation in the figure 3.2 is not identical in game. There are blank spots near the edges of the world map, in case it is not possible to fit in another block of specified dimensions.

Tiles covered by the **blocks** from neighboring cardinal directions slightly overlap, which is intended. It helps create quadrants of players on the world map.

The algorithm that picks the position using the data structure uses a simple BFS algorithm. Based on the cardinal direction chosen by the player connecting to the server, the one of the eight child nodes is chosen as start of the BFS search for the first empty spot on the map in a block that has less than configured amount of cities that can be placed on the same block.

3.7.3 Adding Event Tiles To Map

Events are generated periodically from the start of server. The position for an event tile is found using the positioning algorithm from 3.7.2. New events are only added to block that already contain player controlled cities. The purpose of events has been mentioned in subsection 2.3.12 Events Instead of Oases.

What kind of event is generated is determined using following algorithm.

1. From the list of all events, make a list of those that would manage to appear after applying their chance of appearing.
2. If there are no events left in the list, no event is placed in the found position.
3. If the the new list of events contains at least one event, randomly choose one of them.
4. Generate the units that will be present in the event tile, their amount and everything described in subsection B.2.4 Event Tiles based on the values in specific intervals.

3.8 Timed Actions

Some actions are not finished instantly. Instead, they require certain amount of time in order to be finished. This includes upgrades and downgrades of buildings, training of units, attacking enemy cities, sending reinforcements or waiting for units to return back to their city of origin. We call these action **timed actions**. The amount of time required needs to be accounted with, especially when the server shuts down an the progress is saved for later use.

We decided to remember the time required to finish each of these actions. Once the timer reaches zero, the specific action is executed.

However, implementing timed actions as a discrete simulation would require much computer performance. Instead, the server supports updating cities on demand. Human players who are currently not actively playing do not require their cities to be updated. Same applies for players controlled by artificial intelligence, which is currently not doing anything.

The update occurs when a player performs an action which involves the city or if some other action somehow affects another city. An example of such action is a player attacking reinforcements present in other city.

Every city is also updated every set amount of time. This amount of time can be configured by modifying the value of **CityTileUpdatePeriod** described in section A.4 Server Configuration.

3.9 Artificial Intelligence

As we already mentioned in subsection 2.3.15 Artificial Intelligence as Player, artificial intelligence can be a vital part of the experience from start to finish of the game world.

In our framework, AI has a set of **basic actions** The behavior of AI is defined by **algorithms for action selection**. The implementation of the basic actions is provided by the framework and the actions are identical for each type of AI. The algorithms for action selection are implemented separately and always use the implementation of basic actions provided by framework.

Every AI consists of a collection of arbitrary amount of algorithms, which are represented by an instance of class implementing an **IExecutable** interface. How the algorithms are executed and in which order depends on individual implementation of AI.

We will list the basic actions below and provide name of the file containing the definitions of these actions in brackets.

- Enqueue Building Upgrade (*UpgradeBuilding.cs*)
- Enqueue Building Downgrade (*DowngradeBuilding.cs*)
- Enqueue Unit Training (*TrainUnits.cs*)
- Send Attack (*SendAttack.cs*)
- Send Reinforcements (*SendSupport.cs*)
- Return Reinforcements (*ReturnUnits.cs*)
- Establish New City (*EstablishNewCity.cs*)

If an AI uses any action from the list mentioned above, it should definitely use the predefined actions to prevent undefined behavior. More about the methods provided in the files can be found in the source code in Appendix C File Attachments.

Every action of AI is performed per city for both types of AI implemented in the framework. This means that no action is executed as attempt to capture one city of enemy from multiple cities owned by the AI.

We implemented two types of AI in our framework. One of them is using random actions, which was used to see if the basic actions can be properly used by AI. We named this type of AI **Random**.

The other AI is an attempt to implement a logic that controls the flow of executed actions based on generic rules with configurable parameters. This type of AI is presented as **Rule Directed with Parameters**.

3.9.1 Random

Random AI does use any logic when playing the game. It has a basic set of algorithms to upgrade buildings, establish new city, train units, support cities owned by the AI and to attack surrounding cities that are not owned by the AI.

All these algorithms are performed every time the AI is updated and if it cannot afford the action specific for the algorithm, it skips it and continues with another algorithm.

All actions performed by the algorithms are also randomized. They try upgrading random building from list of all affordable buildings, train any affordable unit, attack any enemy around and establish a new city as soon as the AI has enough resources to do so. The only difference is the supporting algorithm which supports only cities that are being attacked.

This type of AI serves as a test of the implementation of basic actions. However, in games with very low branching factor, where there are very few buildings to upgrade and kinds of units to train, this type of AI becomes very effective.

3.9.2 Rule Directed with Parameters

This type of AI is using more complex algorithms than the Random AI. It also performs some calculations which are stored in the memory and frequently accessed afterwards. What calculations are being performed and what kind of rules the AI utilizes will be described in this section.

Parameters

Because the branching factor is affected by the configuration of the game created by the framework, using rules for specific situations will not always give us positive results.

We decided to try defining the rules based on configurable parameters. The parameters are as follows:

- Military Ratio
  - Total Percentage
  - Aggressiveness
  - Defensive Percentage
  - Offensive Percentage
- Building Ratio
  - Total Percentage
  - Production Percentage
  - Defenses Percentage
  - Construction Building Percentage
  - Recruitment Percentage
- Expanding Urge

The sum of **Total Percentage** of both ratios should always be 100. The percentages of each ratio should always add up to 100 per ratio.

The AI will try making decisions that will keep the ratios in every owned city similar to ratios provided in its parameters. For example, if an AI is parametrized to have 40% building and 60% military ratio, it will try keeping the population used up by buildings and military troops in 40:60 ratio. The subsequent percentages are handled in the same way.

Parameters of **Aggressiveness** and **Expanding Urge** are handled in a different way. The aggressiveness increases the likelihood of the AI to perform an attack on a city which is stronger than the city owned by the AI.

Expanding urge decreases the levels of progress in current city required before the AI tries establishing a new city.

Some of the actions use special calculations of these three classes with singleton instances containing required results.

- **City Memorizer** is being used by actions to attack enemies and establish new cities. The instance of this class remembers data about discovered enemy positions and empty positions. It is used to quickly access corresponding cities so that the AI does not need to search for these positions around his city using BFS algorithm, which can get very slow once the world becomes overcrowded by cities owned by the same AI.

- **Building Cost Memorizer** contains information about resource cost of buildings at all levels of upgrade. It is used for fast access to increase the response rate of the AI.

- **Efficiency Table** is being used when picking which units should be trained. The data stored in the instance of this class contains information about how effective each unit of each nation is against certain units. Using this table, the AI tries recruiting the most effective units currently available. The data is also used when picking units that should attack the enemy.

The effectiveness of the rule directed AI has been tested by the simulator described in section 3.10 Simulator and the results will be shown in chapter 4 Experiments and Results

3.10 Simulator

Creating a balanced game using a framework is not a trivial task. Additionally, the created artificial intelligence has to be tested and their difficulty and performance is hard to be measured because the game world lasts for long period of time.

To test the AI and how balanced the design of configured game is, we decided to develop a simulator.

The simulator uses the implementation of world server with different configuration and does not accept any connections. It is used to run the game with AI controlled players only.

The simulation does not start the default server loop. Instead, it is replaced by the simulation loop. The loop of simulation is turn based and has five phases:

1. Perform an update of every city.

2. Prepare sets of actions every AI wants to perform during this turn.

3. Execute all prepared actions. An action returns a value which is not relevant outside of simulation. The value represents a sleep time of the AI, which is calculated as time required to finish the performed action or the expected time until the action can be performed. The simulation remembers the shortest waiting time required after execution of this phase is finished.

4. Perform a check of victory condition. For simulation, we made the simulation to end once $\frac{2}{3}$ of all players are left with no village to control, or the server time reaches five days. If the victory condition is reached, the simulation loop ends.

5. Add the value remembered from phase three to the time of the server and continue with phase one.

Using the simulator, it is possible to simulate entire life cycle in much shorter amount of time. This lets us collect data about the AI and the game in a short period of time.

4. Experiments and Results

We have decided to run some experiments to confirm our hypothesis that there is no optimal configuration of the rule directed AI, with respect to all game configurations. In other words, our hypothesis is that each game configuration calls to a different play-style and thus for different parameterization of our rule directed AI, described in subsection 3.9.2 Rule Directed with Parameters.

We also want to show, that different configuration of game elements, which changes the way the game should be played, affects the performance of rule directed AI with different configuration of parameters.

We have chosen simulator described in section 3.10 Simulator as our testing environment. In the next sections of this chapter, we will describe how we generated the different configurations of the game and what configurations of the rule directed AI were used. Then, we will talk about how we ran the experiments and how we collected the data. The chapter will be concluded by discussion of the results and how they confirm our hypothesis.

4.1 Generating Input

The input of the experiments for the simulator consists of different sets of configurations of one type of game. Using a game containing many different game elements would allow us to collect more data, which would make the results harder to observe. To be able to collect data that can be observed, we had to generate as simple type of game as possible. If we manage to confirm our hypothesis in a simple type of game, the hypothesis would likely be confirmed in a more complex type of game. We decided to prepare a game with the following elements:

1. Only one type of units. More types of units only provide higher diversity, which we want to prevent.
2. Only one type of resource. More types of resources also increase the diversity of the game.
3. Cities start with no resources.
4. Only one playable nation. More nations would mean higher diversity.
5. Each city has six buildings. There are eight different types of buildings but AI does not exchange resources, leaving us with only seven types of buildings. Action Center type of building is not affected by upgrades, so we can merge it with any other building.
6. Each city can train only three unique units. One offensive, one defensive and one used to conquer enemy cities.
7. Disabled random events.

There are static and dynamic elements of the game we are generating for the input. We will describe those in subsection 4.1.1 Static Game Elements and subsection 4.1.2 Dynamic Game Elements. What influenced the configuration will be covered in subsection 4.1.3 Picked Strategies and Configurations.

4.1.1 Static Game Elements

Static elements always remain unchanged while generating new configuration for the simulator input. In this case, the static elements are types of resources, starting amount of resources, playable nations, buildings and types of units.

We created a game with one type of resource called **Gold**, and a city would always start with no resources. The game only offers one nation of **Barbarians** which can be played with predefined set of building and units.

- Buildings
  - **Construction Building** which is used to give orders to troops and perform building upgrades and downgrades.
  - **Population Building** increases the limit of maximum population of the city.
  - **Production Building** increases the production of **Gold** of the city.
  - **Protection Building** increases the defensive bonus of troops in city.
  - **Storage Building** increases the maximum limit of **Gold** that can be stored in the city.
  - **Training Building** allows training of troops.
- Units
  - **City Attacker** is a basic offensive unit.
  - **City Protector** is a basic defensive unit.
  - **City Conqueror** is a unit capable of capturing enemy cities.

Attributes of the buildings and units listed above are part of the dynamic game elements.

4.1.2 Dynamic Game Elements

Dynamic elements are directly changed by the process of generating new configuration of the game. These elements are resources required to establish a new city, attack power and defensive power of units, salvage amount of resources for units, resource cost of unit that captures enemy cities, resource production speed of buildings and the defensive bonus of walls. All of these attributes are being generated by the project in namespace *GameGenerator*.

4.1.3 Picked Strategies and Configurations

How the game configurations should be generated became a problem. We had to try generating the games so the play-style promoted certain strategies that can be played out by our tested AI.

In Travian, there are many different strategies, from which we have listed some in subsection 2.2.10 Player Strategies. Most of the strategies can be performed by players in games created by our framework. However, our AI does not utilize them and it is hard to defend against some of the strategies. In spite of that,

there is a concept we can use for the AI. The strategies fall under offensive and defensive categories.

We decided to use these two strategy categories as strategies. If the AI focuses on offensive units and attacking other cities, it is playing an offensive strategy. If the AI builds its own defense, it plays defensive strategy. We picked third strategy based on the parameters of the AI, which is expansive strategy. If the AI tries spreading its influence as often as possible, the play-style falls under this strategy.

We started generating the values of dynamic game elements depending on the three picked strategies - offensive, defensive and expansive. We placed an equilateral triangle into a three-dimensional space. Each vertex would gain maximum on different axis and other values would be minimum. These vertices are games that are in favor of one strategy and other strategies would be on their minimum. An example of such triangle is shown in figure 4.1.

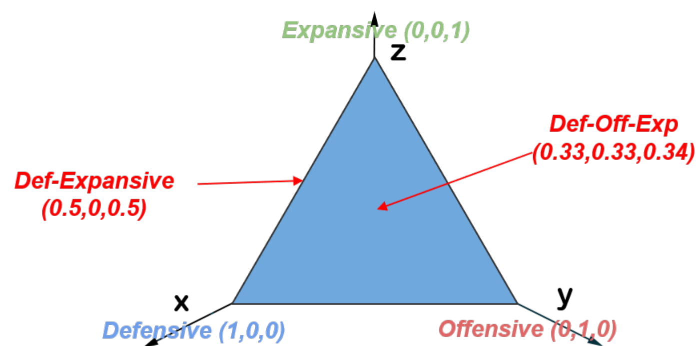


Figure 4.1: A 2-dimensional triangle in 3-dimensional space. Each vertex represents a specific strategy. Picking any point on the triangle is a new configuration for the simulator input.

Configuring a game to be in favor of offensive strategy means increasing the attack power of offensive units. It would also increase the amount of salvage that can be carried by offensive units.

Defensive game would instead increase the defensive capabilities of defensive troops. It also increases the defensive bonus provided by protective buildings.

Expansive strategy would influence the production of resources from buildings. It would also decrease the resource cost of training a conqueror unit and the resource cost of establishing a new city.

We recursively generated points withing the triangle with certain distance between every two points. We passed the values of the points to a method that would set the dynamic elements of the game based on the values. Using this method we recursively generated exactly 66 different configurations of the game.

4.2 Experiment Participants

Because we need to perform multiple tests and the life cycle of the game is still estimated to take hours per game world, there cannot be any human participants.

The only participants in the experiments are players controlled by artificial intelligence.

4.2.1 Rule Directed AI

Because there are many parameters of the rule directed AI and we want to try out as many combinations of parameters as possible, while keeping the simulation time per game world at acceptable levels, we need to generate the configurations of parameters with low density.

We recursively generated different parameters for rule directed AI and made sure there are no repetitions. We placed following restrictions on the parameters during the process of generating.

- **Expanding Urge** cannot exceed 100%.
- Values of **Total Percentage** of **Military** and **Building** ratios change by 25% and the sum of their totals always has to be 100%.
- Values **Expanding Urge**, **Construction Building Percentage** and of **Production Percentage** change by 25%.
- Values of **Defensive Percentage** and **Offensive Percentage** change by 25% and their sum always has to be 100%.
- Value of **Aggressiveness** changes by 35% and cannot exceed 100%.
- Value of **Defenses Percentage** changes by 50%.
- The sum of production, recruitment, defenses and construction building percentages has to be 100%.

Then, we manually configured the parameters of the first AI to following values:

- **Building Total Percentage** was set to 100%, from which the **Recruitment Percentage** was also set to 100%. Other building percentages remained zero.
- **Military Total Percentage** was set to 0%, **Aggressiveness** to 0% and **Offensive Percentage** to 100%.
- **Expanding Urge** was set to 0%.

The first parameters of the AI were used as start of recursion, which used the limitations. Once the parameters were generated, we received exactly 106 different configurations with very high differences between parameters. Because the input games are very simple, the big differences between parameters will not cause us problems. The process of generating the AI parameters can be found in *Simulation10x10.Configuration* namespace in class **RuleDirectedAiParametrizer**.

4.2.2 Random AI

The simplistic nature of the games used as input for experiments allows us to add one more AI to the game, which will be implementation of random AI.

The chances of this AI picking a good strategy is relatively high. This can help us estimate the efficiency of different configurations of our parametrized AI. However, the results of random AI is used as a baseline for the results. The AI with parameters that fits the configuration of the game the most, should perform better than random. On the contrary, the worst fitting parameters of the AI for the game should perform worse than the random AI on average.

4.3 Running The Experiments

The game contains a score system that collects data based on multiple actions performed and player interaction. However, the system collects data of more than one unique type of action into one score category. This makes it hard to see why the player performed well or worse than usual.

We prepared a special scoring system used only for the simulation. Instead of counting score based on how important the actions were, we counted the amount of actions instead.

We placed the 66 generated game alternatives in one folder and let the simulation load each one of them. The world was configured to be able to provide enough space for all 106 players controlled by rule directed AI with different parameter configuration and 1 player controlled by random AI. All of them were randomly placed in the game world.

Each game was played 100 times by the AI and the output of each result of game world was saved into a different comma-separated value (CSV) file. This resulted in 6600 files containing results of 107 players controlled by AI.ontaining results of 107 players controlled by AI.

4.4 Results

Each file from the collected data contains the following information about each AI at the time it was stopped:

- Time of the end of the match and the limit at which the match would end prematurely.
- Nation played by the AI.
- Type of the AI - **Rule directed** or **Random**.
- The amount of cities owned, captured, lost and established.
- All parameters of the AI, for random we used a value **N/A**.
- Amount of attacks sent and received.
- How many units were killed during attack and defense.

- How many units were lost during attack and defense.
- How many offensive, defensive and conqueror units were trained.
- Amount of buildings upgraded and downgraded.
- Amount of gold produced, stolen by attackers and stolen from attackers.

We collected 100 examples of results of each game configuration we generated. We have picked the ones that were the most interesting and counted the average statistics of these games for each AI that participated.

There are multiple ways of estimating which AI performed the best. It is possible to measure it with the amount of cities the AI owned when the game world ended, or the amount of units the AI defeated in combat, or even the amount of gold produced. Depending on different information used to measure the efficiency of the AI, we get different results. Because each game prioritizes different play-style, we decided to measure the efficiency the AI based on different information.

We inspected results of multiple game configurations and picked three of them that provide interesting observations.

We selected games, that ended sooner than the 5-day limit ran out, with the following Offensive:Defensive:Expansive configuration ratios:

1. 20:20:60 - This game prioritizes the strategy of expansion while keeping threats coming from other strategies low and balanced. For this game configuration, we decided to measure the efficiency by the amount of cities the AI established by the end of the game. For this configuration, random AI ended up being on 37th place. Rule directed AI with 50:50 ratio of buildings and military ratio ended up being the most efficient based on the information used to measure the success of the AI. The expanding urge of the most successful AI was mostly 25% and above. An interesting observation from this type of game is, that the most successful AIs had the same same percentages of buildings ratios.

2. 50:20:30 - This game prioritizes offensive strategy and keeps defensive and expansive strategies low. This game configuration had a very easy choice for the information used to compare the efficient with. We picked the amount of captured cities. For this configuration of the game, the random AI finished as 11th, which captured about 5-times lower amount of enemy cities than the best three rule directed AIs. This time, the parameters of the most efficient AI were in 75:25 ratio of military and buildings. Their aggressiveness was 70% and above and they did not prefer upgrading their defensive structures. Surprisingly, the ratio of offensive and defensive units was 50:50 for half of the most efficient AIs.

3. 30:50:20 - This game prioritizes defensive strategy and keeps the offensive and expansive strategies low. The choice of information to estimate the efficiency with for this configuration was to use the amount of enemy units killed during defense. In this case, the random AI was the 6th most efficient. Since this type of game is mostly based on military units, just like the

previous game, it is not a surprise that the most efficient rule directed AI also used 75:25 ratio of military and buildings. Their aggressiveness in this case was 35% and above. Three out of two most efficient rule directed AIs used 25:75 offensive to defensive military unit ratio.

The tables of these three game configurations have many different columns, we included the results of the experiments Appendix C File Attachments of this thesis.

To make sure the different choice of information used to estimate the efficiency of the AI was not affecting the results, we tried comparing these three games using the same information. We did not get the same results, however, the parameters of the most efficient AI were changing as expected but with less significant differences of the parameters.

It is possible that testing more types of AI with different configurations in one game, meaning that the differences of parameters of each AI would be lower, would increase the precision of the results. In spite of that, the results of these three games, which are not very different, have confirmed our hypothesis by showing even slight changes in the most efficient parameters of the most efficient types of AI.

Conclusion

In this chapter, we will provide a conclusion of this thesis. We will compare the results of our work with our goals described in section 1.1 Our Goals.

We have developed a framework which allows creating and configuring persistent MMORTS games. The games created by the framework are capable of emulating most of the basic features offered by these types of games nowadays.

The framework contains two different implementations of artificial intelligence. These were used in a simulator we have created, to test the capabilities of the AI, as well as provide a testing environment for the different types of games created using the framework.

The simulator was later used to perform various experiments, which were used to test our hypothesis that different configuration of the games created using the framework calls to a different play-style, which should result in different configuration of AI required to beat its opponents.

From the set of experiments we performed, we have picked three different configurations of the game to statistically confirm the hypothesis. While it was not easy to determine what kind of information should be used to check the efficiency of AI in games that prefer different play-style, we tried comparing the results for two cases.

In one case, we compared the performance of AI in the game using different piece of information collected from the game, which makes sense because different play-style calls for use of different game features. In this case, the results of experiments have proven that the parameters of AI change in a significant way.

In the other case, we tried comparing the performance of AI using the same piece of information. Even in this case, the results have shown that the different parameters of AI influence their effectiveness. However, the differences were no longer as significant.

We have successfully created a client which directly communicates with the server, is capable of retrieving dynamic information about the game and provides basic visual representation of the game world, mostly using text representation. The client allows performing actions with cities owned by the player and allows basic interaction with other players in form of private messages and coordinated attacks on their cities.

Future Work

This thesis provides multiple opportunities for future work. As the MMORTS industry keeps growing and the existing games keep being expanded with new features, it is possible to implement the features that are missing in the framework, but are present in similar games.

Another way would be to provide wider variety of configuration for the games that can be created using the framework. For example, making some of the features possible to be turned off and on from the server configuration files, or adding more parameters to the XML documents.

Making the client more graphically appealing and dynamic is another way of improving the framework. That would include creating a new design of the client

and providing an art style for visualization of units, buildings and entire cities, including the background and panels.

There is a possibility of adding a more clever artificial intelligence to the game, which would provide a greater challenge for the players. Since it is very simple to add new AI to the game, it would be possible to compare multiple types of AI in a simulator.

Bibliography

Donald Francis Beal. The nature of minimax search. *Dissertation thesis*, 1999.

Marsland Tony Bjornsson Yngvi. Multi-cut alpha-beta-pruning in game-tree search. *Theoretical Computer Science*, 252(1-2).

Hyun Sung Chu. Building a simple yet powerful mmo game architecture. 2014.

Rolf T. Wigand Kiran Lakkaraju, Gita Sukthankar. *Social Interactions in Virtual Worlds.* First Edition. Cambridge University Press, Cambridge, United Kingdom, 2018. ISBN 978-1-107-12882-8.

Peter Norvig Stuart Russell. *Artificial Intelligence.* 3rd Edition. Prentice Hall Press Upper Saddle River, NJ USA, 2009. ISBN 0136042597 9780136042594.

List of Figures

2.1 Graphical representation of a village in Travian. The action bar on top shows village name, gathered resources, resource maximum, current production, tribe and offers multiple navigation and configuration buttons. The panel on the right side shows incoming attacks and units present in the city. Bottom bar shows in-game time and building and training queues. 7
2.2 Graphical representation of resource fields in game. In the middle of the picture, there is a village seen in figure 2.1. Everything around the village are the four types of resource fields. 12

3.1 An old visualization of map in Travian. The controls and layout were used as an example when designing map for our framework. . 29
3.2 Visualization of the data structure used by positioning algorithm. The whole picture is entire map with highlighted cardinal directions. The colors represent children of each node. The middle of the map is the root node of the data structure. 31

4.1 A 2-dimensional triangle in 3-dimensional space. Each vertex represents a specific strategy. Picking any point on the triangle is a new configuration for the simulator input. 39

A.1 Console window after starting the world server from a console. It contains information about IPv4 and IPv6 addresses that can be used to connect to the server and which port the server listens on. Additional output information is configurable. 52
A.2 Default contents of a WorldServer.txt file. It contains configuration of basic server settings and how server actions should be logged. . 53
A.3 Part of the screen with all components that shows right after running the game. Components that have cursor over them are highlighted by changing text color to black. Every other component is written in white color. 54
A.4 Options that are offered when a user hits an option to close the game marked by letter 'X' while viewing screens from section A.5 Main Menu Screens. 55
A.5 Options that are offered when a user hits an option to close the game marked by letter 'X' while viewing screens from section A.6 Game Screens. 56
A.6 A server configuration screen that allows configuration of most of the attributes from section A.4 Server Configuration. 56
A.7 Lets the user write down an IPv4 or IPv6 address of the machine that hosts a server. The IPv4 address in the picture is used to connect to server running on the same computer. 57
A.8 Main components of game screens highlighted on an example of **City View Screen**. 57

A.9 Player setup screen allows the player to set up their user name, name of their first city, position in which the player would prefer being placed in and which nation they would like to play. 59
A.10 After a player sets up his credentials for the game, this is the first screen that pops up. The aim of this screen is to list all cities owned by the player to be able to access any of them as fast as possible. 59
A.11 A world map screen showing surrounding cities and three event tiles nearby. 60
A.12 Action reports screen showing multiple attacks performed on a city owned by selected player. The action report always states which city has been attacked and in case of multiple cities have the same name, world map coordinates are provided to distinguish the cities. 61
A.13 An example of an opened action report. It displays details about units attacking the city and units protecting the city at the time of attack. It also contains information about stolen units and how many units survived the fight on both sides. 61
A.14 A screen with list of private messages. Each message can be selected and opened for reading. 62
A.15 An opened message from list of messages in figure A.14. 62
A.16 Information that appears when a construction building is selected. It shows the resource cost, population requirement of upgrade of each building in the city and amount of time it takes to finish the upgrade. The seconds building cannot be downgraded anymore and because of that, the **Downgrade** option is disabled. 63
A.17 When a building has reached its maximum upgrade level available, it becomes maxed out. 64
A.18 A window of **LumberJack Hut** which is a combination of two building categories. These categories are **Resource Production** and **Military Troop Recruitment**. 64
A.19 A window that allows exchanging resources. These values and the exchange ratio is specific for a **MarketPlace** on upgrade level 1. . 65
A.20 An example of window from section **Military Troop Recruitment**. 66
A.21 An example of a building from military action category. It provides details about ongoing military actions that involve the city. 66
A.22 When a player selects an option to attack a player, this is the screen that opens. It is possible to specify the amount of units and the values will not go above the amount of units the player owns. 67
A.23 A screen that shows immediately after selecting an option to send a private message to a player. 68

B.1 Contents of **StartingResources.xml** file showing all relevant elements of the XML tree. The contents are identical to contents of file **EstablishCityCost.xml**. 70
B.2 Contents of **UnitTypes.xml** file showing all relevant elements of the XML tree. 71

B.3 Contents of **f_Lumberjack Hut.xml** file showing elements of the XML tree. The building in this file is a combination of two different building types. 72
B.4 Contents of **b_Spearman.xml** file showing elements of the XML tree. 74
B.5 Preview of all images included in the default game provided by the framework. 77

List of Abbreviations

MMORTS Massively Multi-player On-line Real-Time Strategy
AI Artificial Intelligence
AP Attack Power
DP Defense Power
LINQ Language Integrated Query
SQL Structured Query Language
XML Extensible Markup Language
DLL Dynamic Link Library
GUI Game User Interface
IP Internet Protocol
BFS Breadth-First Search
TCP Transmission Control Protocol

A. User Documentation

A.1 System Requirements

The server has been implemented using features of .NET Framework 4.5. We did not use any external database and managed to be able to use and save all data using C# serialization.

The only requirement to be able to use and run the framework and games created for the framework is to have .NET Framework of version 4.5 or above.

A.2 Running the Game

In this chapter we will describe how the game server can be started and how players can connect to it.

The game server can be started in two ways. Each approach requires seven libraries (*.dll files) to be able to successfully start. The names of these libraries are:

- Common.dll
- NetworkingConstants.dll
- NetworkingPackets.dll
- SerializationBinary.dll
- SerializationXml.dll
- ServerConfiguration.dll
- ServerImplementation.dll

First approach to run the server is by starting it from console. To do that, all that has to be done is to run a file called **WorldServer.exe** from a folder containing all required libraries. This opens a console and the server automatically starts. The console window will contain an IPv4 and IPv6 address on which the server is listening including a specific port. The console window will look similar to figure A.1.

Depending on the configuration of the server, the console will periodically display more information about what is happening on the server. Which messages appear in the console of the first approach and the configuration of the server can be configured in a file **WorldServer.txt** generated and mentioned in A.3. The process of configuration is covered in A.4.

The second option to run the server is directly from the client by running file **WinFormsGameClient.exe**. The major difference is, that there is no **WorldServer.txt** configuration file that can be used to change the settings of the server. The configuration is set up directly from the interface. Additionally, this approach does not show information about IP addresses, port or what actions are happening on the server. This approach still generates all other files described in A.3.

```
Establishing Server...
Server Established.
Searching for Available IP Addresses for Connection.
Server Host is Listening on IPv6 Address : fe80::69c6:81a2:4393:5e4c%12
Server Host is Listening on IPv4 Address : 192.168.0.102
Server Port : 27015
Expecting Connections...
[00:00:00] [City of Skcwxhuywrortfj [X:101 Y:98]] Has joined the world!
[00:00:00] [City of Skcwxhuywrortfj [X:101 Y:98]] Type of Articficial Intelligence : Random Choice
New AI Created :
  Name : Skcwxhuywrortfj
  Nation : Persian Empire
  City Coords : X 101 Y 98
  City Name : City of Skcwxhuywrortfj
  AI Type : Random Artificial Intelligence
[00:00:00] [City of Hgtyfqhyractxap [X:99 Y:98]] Has joined the world!
[00:00:00] [City of Hgtyfqhyractxap [X:99 Y:98]] Type of Articficial Intelligence : Random Choice
New AI Created :
  Name : Hgtyfqhyractxap
  Nation : Persian Empire
  City Coords : X 99 Y 98
  City Name : City of Hgtyfqhyractxap
  AI Type : Random Artificial Intelligence
[00:00:00] [City of Tudooawinkwuhur [X:97 Y:100]] Has joined the world!
[00:00:00] [City of Tudooawinkwuhur [X:97 Y:100]] Type of Articficial Intelligence : Random Choice
New AI Created :
```

Figure A.1: Console window after starting the world server from a console. It contains information about IPv4 and IPv6 addresses that can be used to connect to the server and which port the server listens on. Additional output information is configurable.

A.3 Generating Configuration Files

When the server starts, it checks for files required by the framework. If there are no files or some of them are missing, the framework generates those and saves them on a disk. The location of these files is strictly defined and changing the file system may result in the framework being unable to find the files making it generate them again.

In the root folder of the framework, up to four new folders can be created and a new file called **WorldServer.txt** appears when the server starts from the console. The meaning of the folders in Appendix B Advanced User Documentation and the configuration of **WorldServer.txt** is covered in section A.4 Server Configuration.

A.4 Server Configuration

As mentioned in section A.2 Running the Game, the game world can be configured from a file called **WorldServer.txt** which can be opened and it will look like figure A.2. This file allows configuration of the following:

- **WorldSize** specifies the dimensions of the game world. The world will have [**WorldSize**]x[**WorldSize**] dimensions.
- **AIAddSpeedInMinutes** specifies the frequency of adding new players controlled by artificial intelligence to the game world. The frequency is in minutes.
- **InitialAICount** specifies the amount of players controlled by artificial intelligence to the game immediately after a server is launched. Half of this value will be used to add Random AI player. The other half is used to add Rule Directed AI players.
- **AutoSpawnCount** is an amount of AI players added periodically while the server is running. Similar to the value of InitialAICount, half of this value will be used to add Random AI player. The other half to add Rule Directed AI players.

```
WorldSize=200
AIAddSpeedInMinutes=15
InitialAICount=8
AutoSpawnCount=5
BlockSize=5
MaxPlayersPerBlock=9
EventTilesPlacedPerPeriod=3
EventTilePlacingPeriod=90
EventTileUpdatePeriod=1
AiRandomUpdatePeriod=20
AiRDirUpdatePeriod=15
CityTileUpdatePeriod=10
LogLevel=131071

**COMMENTS**
Logs are being performed for Artificial Intelligence only.
LogLevel values (can be combined) :
1  - Log Building Upgrades
2  - Log Building Downgrades
4  - Log Start of Unit Training
8  - Log Establishment of New Cities
16 - Log Sending of Supports
32 - Log Sending of Attacks
64 - Log Requests to Return Supports
128 - Log Results of Defense Actions
256 - Log Results of Attack Actions
512 - Log Cities Getting Captured
1024 - Log Units Returning Back To Owner City
2048 - Log Supporting Units Arriving to Target City
4096 - Log Score Gains
8192 - Log Finished Building Upgrades and Downgrades
16384 - Log Finished Military Recruiting
32768 - Print AI joining the world Military Recruiting - Always to Console.
65536 - Log To Console Instead Of File
65535 - Log Everything To Log Files
131071 - Log Everything To Console
```

Figure A.2: Default contents of a WorldServer.txt file. It contains configuration of basic server settings and how server actions should be logged.

- **BlockSize** specifies size of a block used by new player position finding algorithm. The dimensions are [**BlockSize**]x[**BlockSize**]. The value is not recommended to be changed unless there is a good reason for it and the user knows how the algorithm works.

- **MaxPlayersPerBlock** says how many player cities can be placed in every block by the positioning algorithm.

- **EventTilesPlacedPerPeriod** is an amount of event tiles placed in each block occupied by at least one player.

- **EventTilePlacingPeriod** value specifies how often should the server generate event tiles on the map. The value is the amount of minutes.

- **EventTileUpdatePeriod** is a frequency in seconds at which event tiles are checked and updated.

- **AiRandomUpdatePeriod** is a frequency in seconds at which players controlled by artificial intelligence are checked, updated and try performing some actions.

- **AiRDirUpdatePeriod** has the same meaning as AiRandomUpdatePeriod but it applies for Rule Directed AI controlled players.

- **CityTileUpdatePeriod** is a frequency in seconds of checking and updating all city tiles.
- **LogLevel** specifies what kind of messages should be logged by the server. All values and their meaning is listed in the file itself. These values are considered flags, which means that they can be combined in order to be able to use arbitrary amount of the values for this attribute.

None of the values in the configuration files should be negative. All of the values should be natural numbers.

A.5 Main Menu Screens

In this section, we will go through each screen of the Game User Interface (GUI) which visualizes the games runnable by the framework. The GUI shows when file called **WinFormsGameClient.exe** is started. We will describe each screen that we can get to during the runtime and describe their components up to the point where a game is entered.

A.5.1 Introduction Screen

The first screen that shows up after running **WinFormsGameClient.exe** is a screen with four possible options for selection. Part of the screen is shown in figure A.3 with a highlighted option.

1. Start New Game
2. Join Existing Game
3. Load Game
4. Close Game (X)

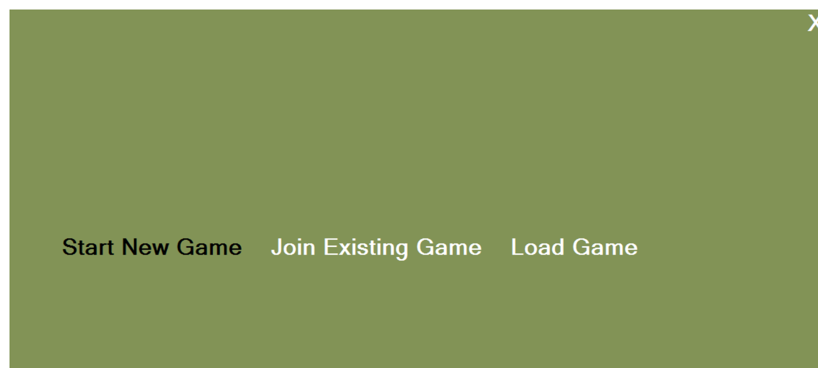


Figure A.3: Part of the screen with all components that shows right after running the game. Components that have cursor over them are highlighted by changing text color to black. Every other component is written in white color.

Start New Game

This choice immediately brings the user to a new screen seen in figure A.6 and described in subsection A.5.2 Server Configuration Screen.

Join Existing Game

This brings the user to a new screen shown in figure A.7. The new screen is further described in subsection A.5.3 Join Existing Game Screen.

Load Game

This option attempts to search for a saved game in a default folder. If the folder is not found or some of the save files are missing, the game is not loaded and the user can choose another option.

Close Game (X)

An option to close the game is present on every screen. This option shows a small window shown in figure A.4 which asks if the user wants to quit the game and the user can choose an answer from two offered options.

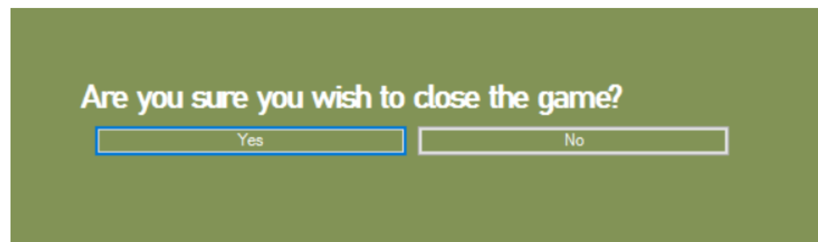


Figure A.4: Options that are offered when a user hits an option to close the game marked by letter 'X' while viewing screens from section A.5 Main Menu Screens.

The window changes when the user enters screens from section A.6 Game Screens to what it looks like in figure A.5. The selection of options changes to three options.

- **Save and Exit** which saves the progress of the game server and then closes the game.
- **Exit and Don't Save** which closes the game without saving the progress.
- **Cancel** option closes the window with the three options without saving the progress or exiting the game.

A.5.2 Server Configuration Screen

Allows setting the server configuration attributes of **WorldServer.txt** for the console version of the server, which is not present in the client. The screen allows setting everything except of update periods and the level of logging, which is disabled by default.

Figure A.5: Options that are offered when a user hits an option to close the game marked by letter 'X' while viewing screens from section A.6 Game Screens.

When everything is set, hitting **Confirm** button will proceed to the next screen described in subsection A.6.1 Player Configuration Screen. Additionally, clicking the button will create a server in the background and the client connects to it. Because of that, playing the game is still based on communication between client and server.

Figure A.9 shows how the screen looks like without modified game files.

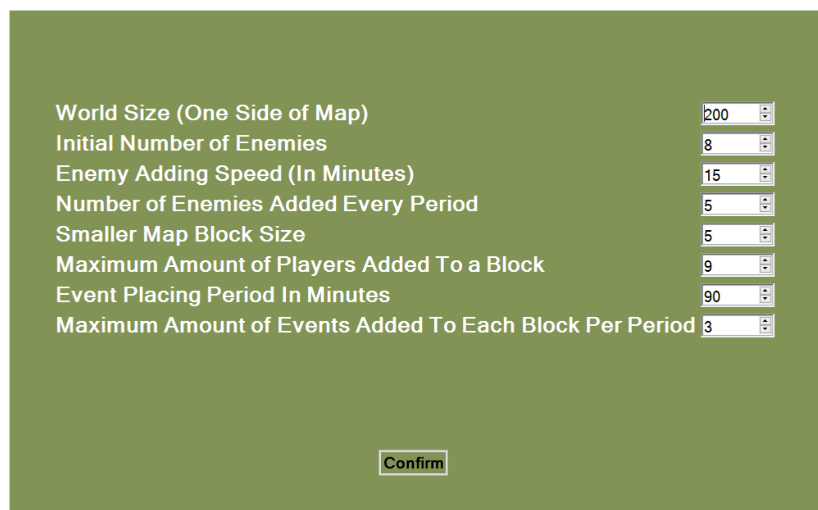


Figure A.6: A server configuration screen that allows configuration of most of the attributes from section A.4 Server Configuration.

A.5.3 Join Existing Game Screen

This screen allows the user to write an IPv4 or IPv6 address of a machine that is hosting a server the user wants to connect to. By default, the field containing IP address is set to the one seen in the figure which is used to connect to server running on the same machine.

Once the IP address field is filled with desired address, hitting the **Connect** button will attempt to connect to the server. If the connection fails, the user will remain on the same screen. Otherwise, the user will be redirected to a new screen described in subsection A.6.1 Player Configuration Screen.

An IP address of a server can be seen when a user starts the server from console. Running the server from console is described in the first approach of section A.2 Running the Game.

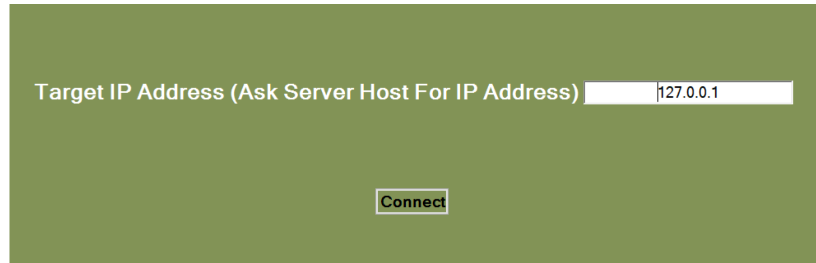


Figure A.7: Lets the user write down an IPv4 or IPv6 address of the machine that hosts a server. The IPv4 address in the picture is used to connect to server running on the same computer.

A.6 Game Screens

Screens in this chapter receive additional components that are present on each screen, but there are some exceptions which will be listed in this chapter.

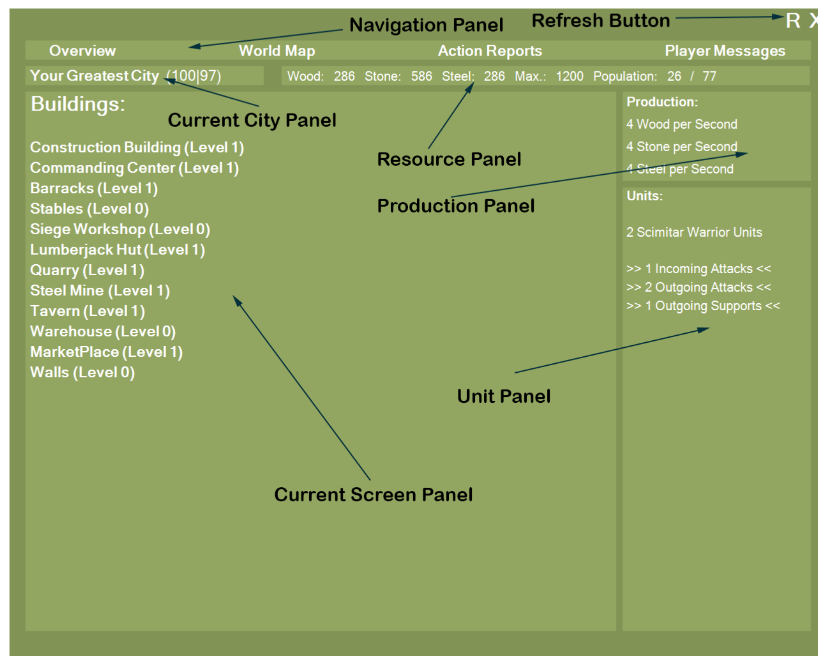


Figure A.8: Main components of game screens highlighted on an example of **City View Screen**.

The main components of game screens are highlighted in figure A.8.
The new components that are the same on all game screens are:

- **Refresh Button** can be used to refresh the current screen by requesting most recent information about the currently opened screen.
- **Navigation Panel** contains links to four other screens. **Overview** redirects to screen described in subsection A.6.2 Cities Overview Screen, **World Map** redirects to subsection A.6.3 World Map Screen, **Action Reports** option redirects to subsection A.6.4 Action Reports Screen and **Player Messages** option redirects to screen described in subsection A.6.5 Player Messages Screen.

- **Current City Panel** switches the screen back to overview of city the player was managing the last time. The screen is described in subsection A.6.6 City View Screen. The X and Y values in brackets represent the position of the city in the world map.
- **Resource Panel** displays information about resources currently stored in the city. The panel also contains the information about the maximum of each resource the city can store and the current and maximum population the city has.
- **Production Panel** shows information about resource production in the city the player is currently viewing.
- **Unit Panel** shows units that are currently present in the city, including reinforcement. Additionally, the panel displays warnings about incoming and outgoing attack and support actions.

Game screens contain **Current Screen Panel** which is dynamic depending on a screen we are currently viewing.

A.6.1 Player Configuration Screen

This screen is accessed when player joins an existing server or starts a new game on his own. Figure A.9 shows the layout and default values of each field. The screen is the only one that does not contain any of the new components of game screens.

There are four configurable fields.

- **Choose Your In-Game Name** field is a name the player would like to use when playing the game. This name becomes a user name which is used to connect back to a server where the player already has progress. When the name is in use, the player will be prompted to enter the password associated with the user name to be let in the game with the progress connected to the user name. If the password entered does not match, the user will receive a message and will be allowed to choose a new name or try another password.
- **Choose The Name of Your First City** is a name with almost no restrictions. Players can change a name of their cities anytime during the game.
- **Choose The Place You Would Like To Spawn At** can be set by choosing from a selection of eight different cardinal directions and a choice to randomly pick one of them. Depending on the choice, a position of the player is picked in the game world.
- **Choose The Nation You Would Like To Play** can be set by choosing from a list of playable nations. If a player is connecting to a server hosted from console, the list of playable nations is downloaded from the server. The choice of nation is permanent.

Once the configuration is ready, hitting the **Confirm** button will confirm the changes. If everything works out properly and the server is not full or the name is not taken, the player will be taken to the next screen described in subsection A.6.2 Cities Overview Screen.

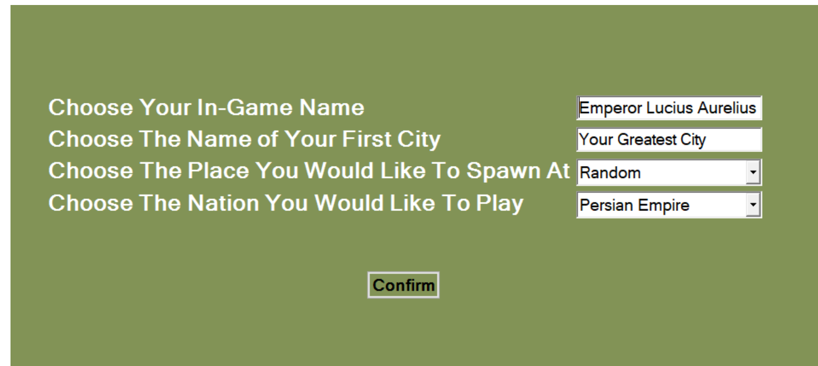


Figure A.9: Player setup screen allows the player to set up their user name, name of their first city, position in which the player would prefer being placed in and which nation they would like to play.

A.6.2 Cities Overview Screen

A player always finds himself at this screen after logging in to the game that looks similar to a screen in figure A.10. A few seconds after entering this screen as a new player, a window to set up a password pops up. The password has almost no restrictions and can even be empty but cannot be changed later.

Because no city is selected at the point of displaying this screen, **Current City Panel**, **Resource Panel**, **Production Panel** and **Unit Panel** are not present at this screen.

Figure A.10: After a player sets up his credentials for the game, this is the first screen that pops up. The aim of this screen is to list all cities owned by the player to be able to access any of them as fast as possible.

A.6.3 World Map Screen

This screen is only accessible when a player has a city selected. The map shows the city owned by a player in the middle of the piece of map unless the city is too close to an edge of the map.

From this screen it is possible to send attack orders, reinforcements, private messages when clicked on a tile containing a city. On tiles with events it is only possible to send out attack and support orders. When the tile is empty, the player can is offered to establish a city on that tile. Attack and support actions

open new screens covered in subsection A.6.8 Send Attack and Support Screen. Private messages are described in subsection A.6.9 Send Private Message Screen.

Hovering over each tile with cursor shows a tool-tip message providing additional information. For empty tiles it is the position of the tile and resources required to establish a city on that position. The cost increases with greater distance. Tiles that contain a city will show information about who owns the city, how many points the city has and what nation the city uses. Event tiles show description about the event, which can be configured in the framework.

To navigate on the map, it is possible to use arrow keys on the keyboard which will move the map view by one tile to the chosen direction.

An example of how the map is displayed can be seen in figure A.11.

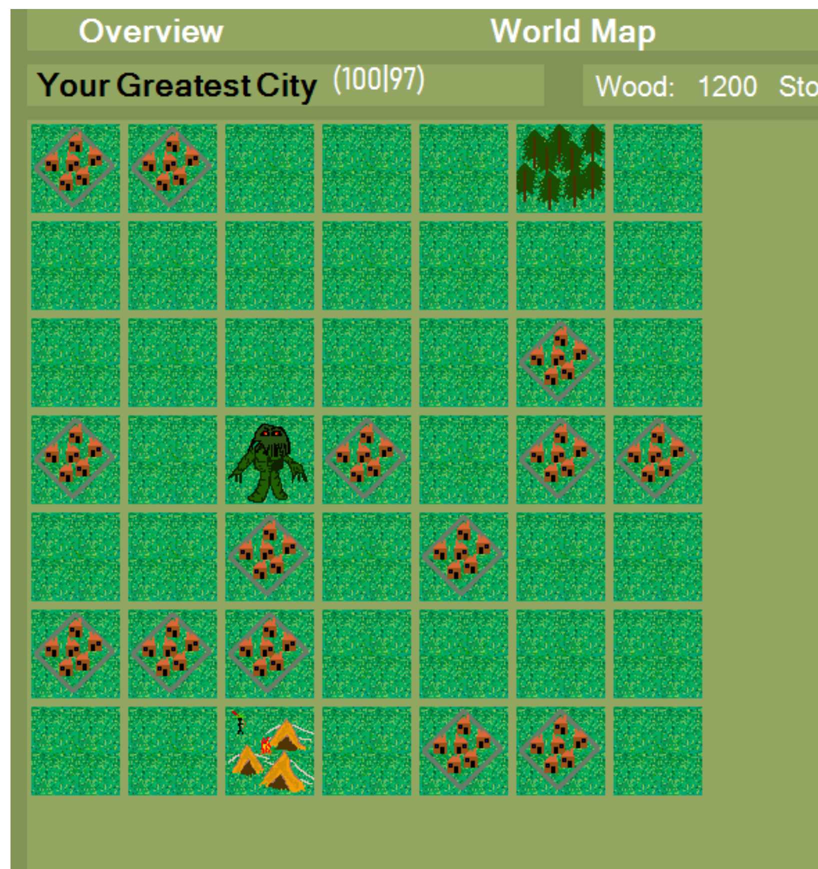


Figure A.11: A world map screen showing surrounding cities and three event tiles nearby.

A.6.4 Action Reports Screen

Action reports are stored in this screen. They contain information about which city got attacked and the position of the city on the world map, see figure A.12 for reference.

Each action report can be selected to show detailed information about the action. In case of defense, all information is shown to the player like in figure A.13. However, if the player performed an attack which resulted in all of their units getting killed, the action report does not contain any information about the status about the enemy defensive units.

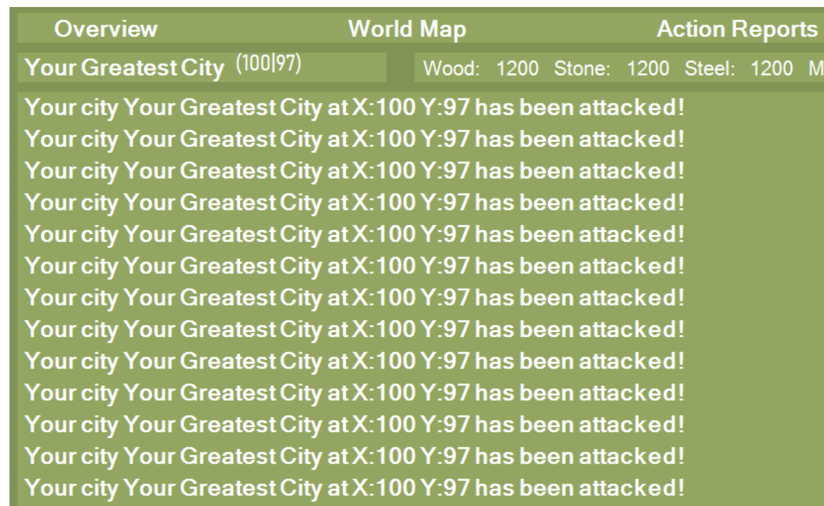


Figure A.12: Action reports screen showing multiple attacks performed on a city owned by selected player. The action report always states which city has been attacked and in case of multiple cities have the same name, world map coordinates are provided to distinguish the cities.

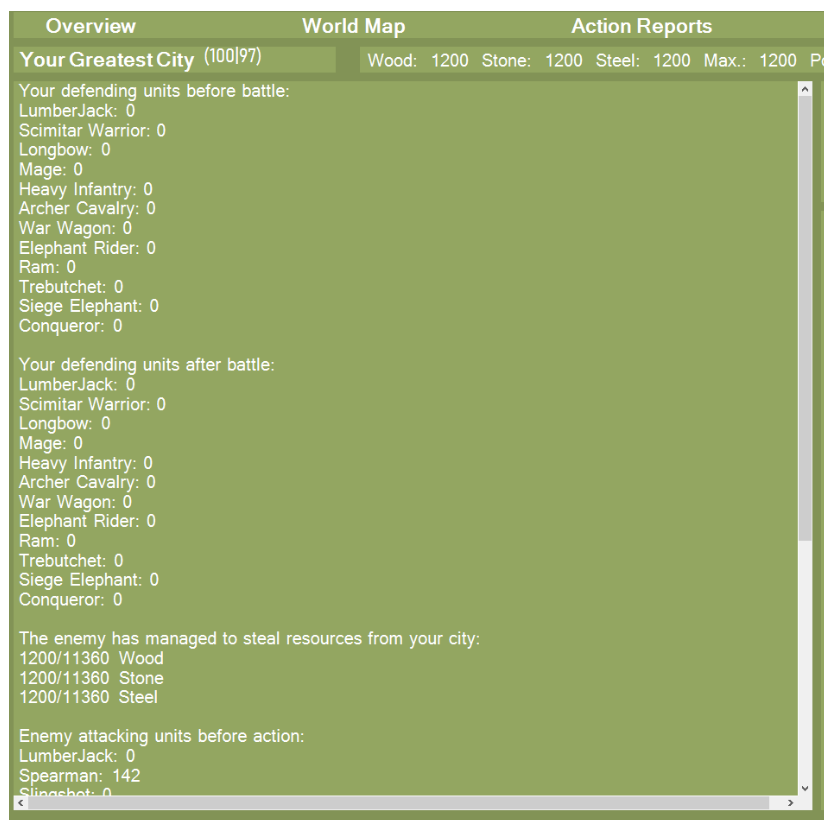


Figure A.13: An example of an opened action report. It displays details about units attacking the city and units protecting the city at the time of attack. It also contains information about stolen units and how many units survived the fight on both sides.

There is a limit of action reports that can be stored at once. So once a player has too many action reports, the oldest reports will start being replaced by new

ones.

A.6.5 Player Messages Screen

A player can read private messages received from other players from this screen. Figure A.14 displays the overview of all received messages.

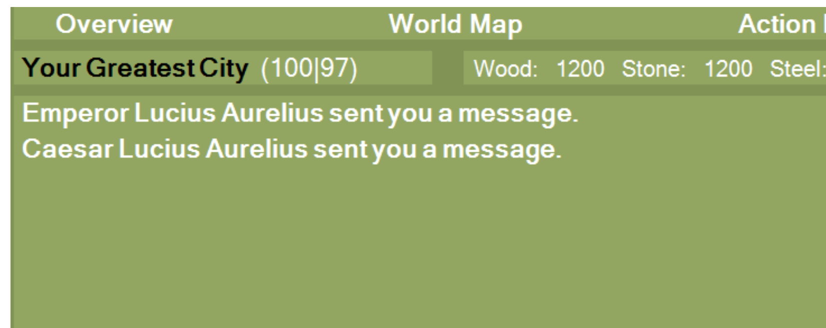


Figure A.14: A screen with list of private messages. Each message can be selected and opened for reading.

Selecting one of the messages in this screen changes the **Current Screen Panel** so that it shows the text of the selected message as shown in figure A.15.

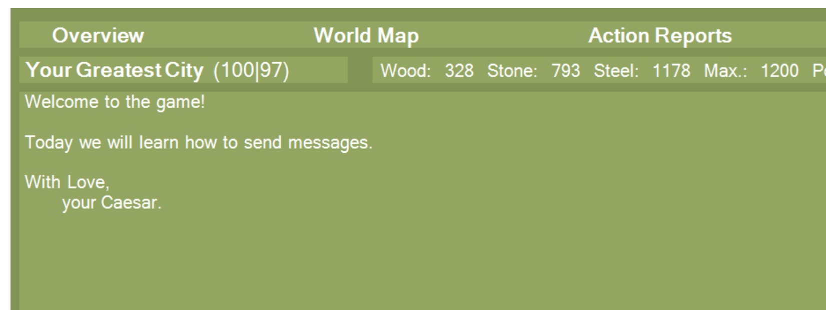


Figure A.15: An opened message from list of messages in figure A.14.

A.6.6 City View Screen

The purpose of this screen is to allow the player to see all buildings available in the city for construction. Each building has its current upgrade level shown next to it. An example of this screen can be seen in figure A.8.

Selecting any of the buildings opens a new screen that shows information about the building and allows performing actions specific for the type of building. Description of each building and their use is covered in subsection A.6.7 Building View Screen. Hovering over a name of a building on this screen causes a tool-tip text to appear, giving a short description of the building.

A.6.7 Building View Screen

This screen has different use and shows different information depending on what purpose the selected building has. We will describe all buildings provided

in the default game, seen in **Current Screen Panel** of figure A.8, depending on the following categories:

- **Building Construction** - Construction Building
- **Resource Storage** - Warehouse
- **Resource Production** - Lumberjack Hut, Quarry, Steel Mine
- **Resource Exchange** - Marketplace
- **Inns** - Tavern
- **Military Troop Recruitment** - Barracks, Stables, Siege Workshop
- **Military Action** - Commanding Center
- **City Protection** - Walls

Now we describe each category and buildings they include for the default game provided by the framework.

Building Construction

Buildings from this category allow construction, upgrading and downgrading of all other buildings in the city. Each building has **Upgrade** and **Downgrade** options as shown in figure A.16.

Construction Building (Level 1)	Wood: 270	Stone: 180	Steel: 120	POP: 2	0h 0m 25s
Upgrade	Downgrade				
Commanding Center (Level 0)	Wood: 100	Stone: 100	Steel: 100	POP: 2	0h 0m 13s
Upgrade	Downgrade				

Figure A.16: Information that appears when a construction building is selected. It shows the resource cost, population requirement of upgrade of each building in the city and amount of time it takes to finish the upgrade. The seconds building cannot be downgraded anymore and because of that, the **Downgrade** option is disabled.

The downgrade option cannot be used if the building has already reached the minimal level it can reach. Downgrading a building takes much lower amount of time than upgrading but grants not resources in return.

When the upgrade option can be clicked, it means that the city has enough resources to be able to start upgrading the structure. When a building is maxed out, the information about resource requirements is replaced with *Maxed Out!* message as seen in figure A.17 and the **Upgrade** option disappears, leaving it with only one option which is **Downgrade**.

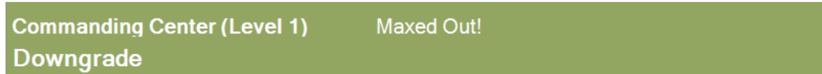


Figure A.17: When a building has reached its maximum upgrade level available, it becomes maxed out.

Resource Storage

Buildings from this category serve the purpose of storing resources in the city. Upgrading buildings from this category increases the maximum capacity of each resource the city can store at one time.

The window itself provides information about the amount of resources the building can store and how much population the building requires at current and next level.

Resource Production

Resources are gained passively from these buildings. In the default game that comes with the framework, every resource producing building only produces one kind of resource.

Lumberjack hut, however, produces one type of resource and also allows training a unit called **LumberJack**.

Buildings that only provide production of a resource show information about the production and population required by the building on previous, current and next level of the building. For **LumberJack**, the window also adds options from figure A.20. How exactly a windows of this combined building looks like can be seen in A.18.

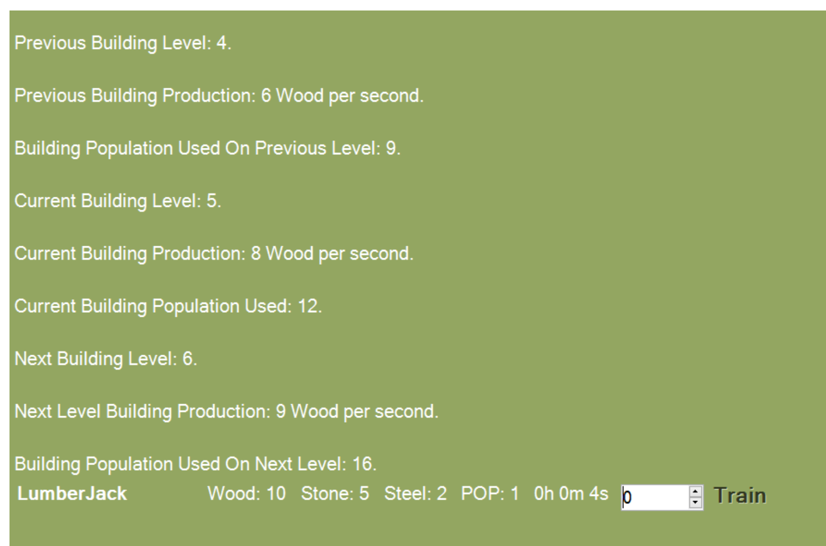


Figure A.18: A window of **LumberJack Hut** which is a combination of two building categories. These categories are **Resource Production** and **Military Troop Recruitment**.

Resource Exchange

Exchanging resources is important once player accumulates way too many resources of certain type and there is no way to spend them. For this purpose, in the default game of this framework, there is a building called **Marketplace** that allows exchanging resources for any other resource. However, the ratio is not one to one, moreover, the ratio is not in favor of the player.

The ratio can be made more fair by upgrading the building, but it also increases the amount required to perform an exchange of resources. How the screen looks like is shown in figure A.19.

Figure A.19: A window that allows exchanging resources. These values and the exchange ratio is specific for a **MarketPlace** on upgrade level 1.

Inns

Inns allow cities to increase the maximum of population of the city. Population is required to build and upgrade more buildings and to train larger armies.

A window showing details about this building category shows the information about population capacity provided by the building and population used by the building itself for previous, current and next level of the building.

Military Troop Recruitment

These types of buildings allow recruiting new units for the currently selected city. In this window, each unit that can be trained from that building contains the following:

- **Unit Name**
- **Tool-tip** when hovering over the name of the unit showing the type of unit, attack power, defensive powers and movement speed. Additionally, this tool-tip provides information about required buildings that need to be built before the unit can be trained.

- **Unit Resource Cost** required to train the unit
- **Population Cost** of the unit
- **Time Required** to train one unit of the kind
- **Numeric Field** containing amount of units of the kind the player is willing to train.
- **Train** option which starts training the amount of units provided by **Numeric Field**.

Figure A.20 shows how the window looks like for **Stables** provided in the default game of the framework.

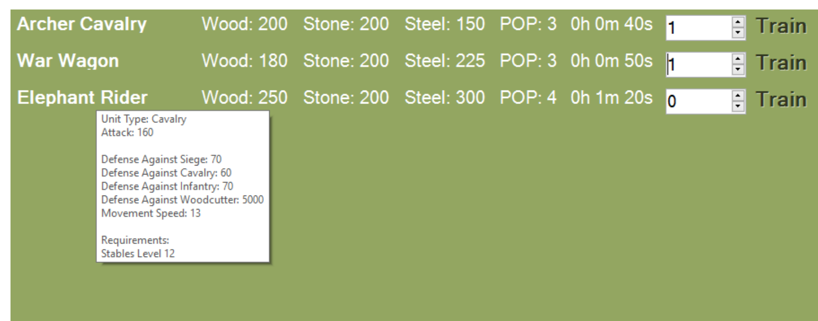


Figure A.20: An example of window from section **Military Troop Recruitment**.

There are units with abilities. Units called **Ram** can attack enemy protection structures to lower the defensive capabilities of enemy city.

Very important unit is **Conqueror**, which can capture enemy city if the unit is present in an attack and survives the fight.

Military Action

These types of buildings offer an option to change the name of the currently selected city. However, the main use of this building is to provide more information about ongoing military actions related to currently selected city.

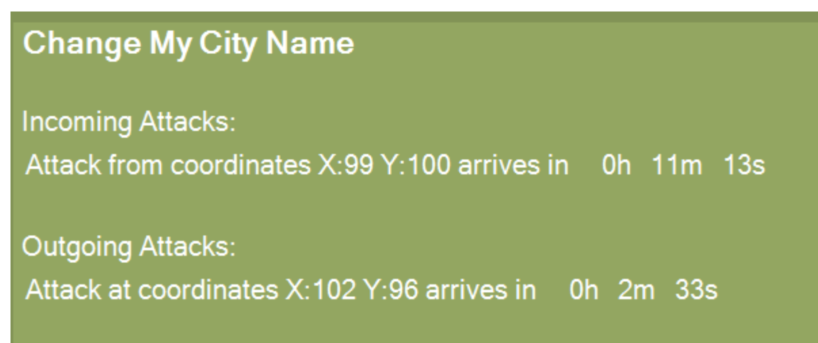


Figure A.21: An example of a building from military action category. It provides details about ongoing military actions that involve the city.

The additional information consists of the source of enemy attack or reinforcements, or where are the troops from currently selected city being sent. The details include time of arrival of the units to their destination. An example is provided in figure A.21.

When hovering over an action where the currently selected city is the initiator of the action, a tool-tip window appears showing more information. Typically it shows the amount of units involved in the action. In case of units that are returning back from a successful raid on enemy city, the window also shows the amount of salvaged resources which they are bringing back from the action.

If the currently selected city has reinforcement in other cities, it is possible to give them an order to return back. The order can only be given from this type of buildings.

City Protection

Buildings from this category provide additional defense bonus to all troops present in the city at the time of defense against enemy attacks. Because the bonus can become high, there are types of units that can start attacking the protection buildings directly during attack, which will cause the protection buildings to degrade certain amount of levels.

There are no special options for this building category. The only information provided when viewing details of protection buildings is the bonus to defense and population used for the current and next level of the structures.

A.6.8 Send Attack and Support Screen

Attacks are being sent from screen seen in figure A.22. It allows specifying how many units should be sent at target city. The city to attack is specified when choosing an attack option from world map.

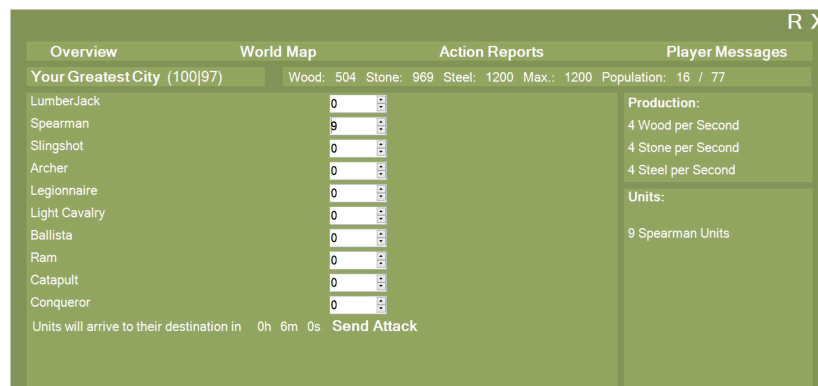


Figure A.22: When a player selects an option to attack a player, this is the screen that opens. It is possible to specify the amount of units and the values will not go above the amount of units the player owns.

All game panels are present when planning the attack from this screen. Because of that it is possible to see how many units there currently are in the city. The units panel also shows incoming attacks for the player to be aware of possible dangers.

Screen for sending support is almost identical with the screen to send an attack.

A.6.9 Send Private Message Screen

This screen is used to write messages to other players. The recipient is chosen by selecting a city owned by that player on the world map. The messages have set limit of characters.

Private messages can be used as reminders or notes when the recipient is the same player as sender of the message.

An interface for sending messages is shown in figure A.23.

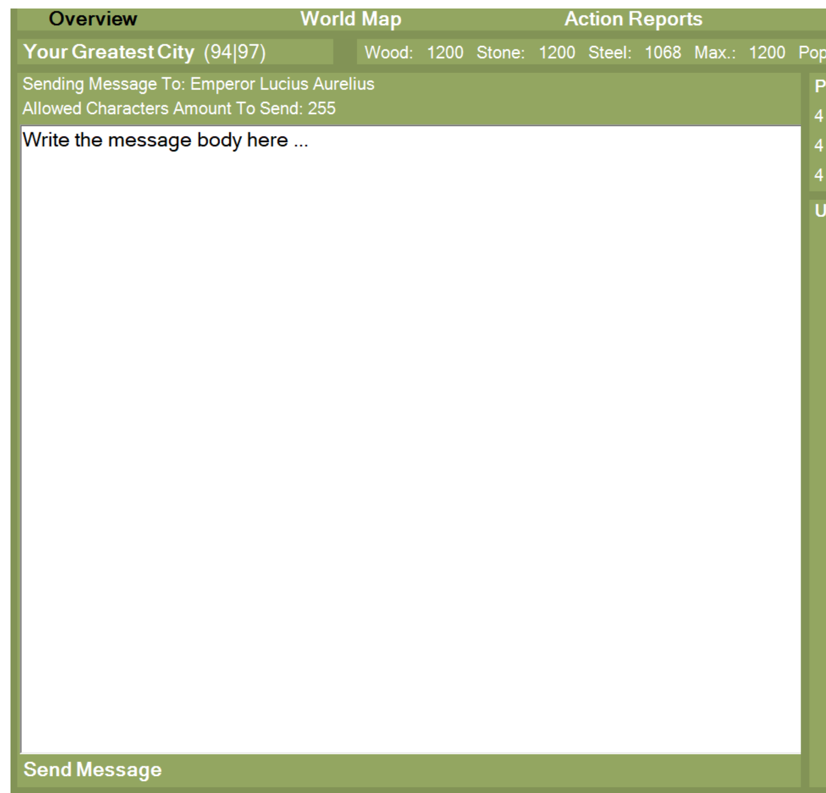


Figure A.23: A screen that shows immediately after selecting an option to send a private message to a player.

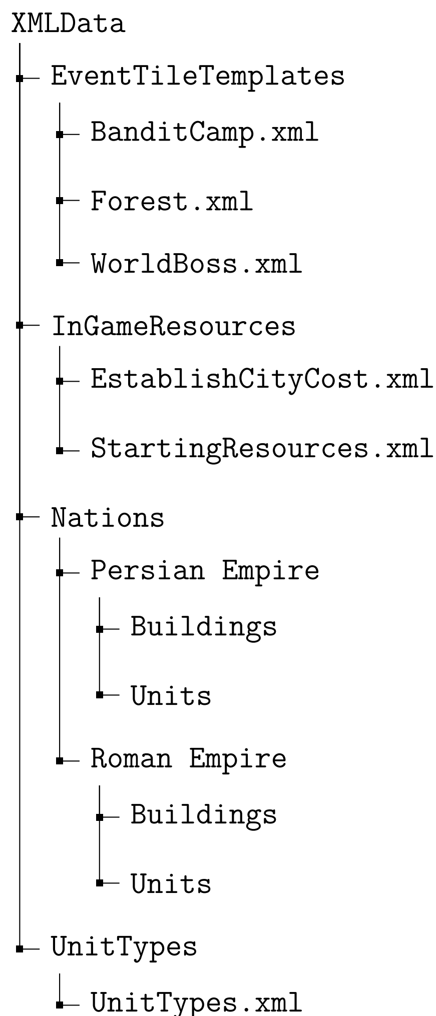
B. Advanced User Documentation

In this chapter, we will describe the elements of the game configurable by modifying XML documents which are generated in section A.3 Generating Configuration Files.

B.1 XML File System

The files are stored in a hierarchy of folders. It is very important not to change the hierarchy unless it is allowed. Which changes to the hierarchy are allowed and how to configure the files will be covered in section B.2 XML Configuration. The root folder of the hierarchy is called **XMLData**.

The hierarchy of folders that is generated by the framework looks like this:



B.2 XML Configuration

We will go through each folder on the first level of the **XMLData** folder hierarchy. XML files store their data in format that can be read by human while being able to. Because of that, any software capable of editing text files is enough to be able to configure and create new games using this framework.

B.2.1 Game Resources

It is possible to configure starting resources of every city from file **StartingResources.xml** located in the **InGameResources** folder. This means that even a city that is established starts out with the amount of resources specified. The resources will also become the only resources that can be used in the game.

```
<?xml version="1.0" encoding="UTF-8"?>
- <ArrayOfResource xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <Resource>
      <Name>Wood</Name>
      <Amount>500</Amount>
    </Resource>
  - <Resource>
      <Name>Stone</Name>
      <Amount>400</Amount>
    </Resource>
  - <Resource>
      <Name>Steel</Name>
      <Amount>200</Amount>
    </Resource>
  </ArrayOfResource>
```

Figure B.1: Contents of **StartingResources.xml** file showing all relevant elements of the XML tree. The contents are identical to contents of file **EstablishCityCost.xml**.

As seen in figure B.1 the root element of the XML tree is called **ArrayOfResources**. It is possible to add arbitrary amount of **Resource** child elements to the root. Each **Resource** child element consists of **Name** and **Amount** child elements, which specify how the resource type will be called and how many units of that resource each new city receives.

Another file in this folder is called **EstablishCityCost.xml** which has exactly the same structure as **StartingResources.xml**. The difference is, that this file is used to provide the base resource cost required to establish a new city on an empty tile. The meaning of the values specifying **base** resource cost is, that the values are not modified by distance modifiers. Because of that, the values in the game will always be higher than the defined ones.

B.2.2 Unit Types

This section covers contents of **UnitTypes** folder, specifically the only file it contains called **UnitTypes.xml**.

In figure B.2 we can see the structure of the XML document. The root of XML tree is called **ArrayOfString** and its child elements can only be called **string**.

Similar to resource types, the number of child elements of the root element can be arbitrary.

```
<?xml version="1.0" encoding="UTF-8"?>
- <ArrayOfString xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <string>Siege</string>
    <string>Cavalry</string>
    <string>Infantry</string>
    <string>Woodcutter</string>
</ArrayOfString>
```

Figure B.2: Contents of **UnitTypes.xml** file showing all relevant elements of the XML tree.

This file provides data about all available types of units that can be used in the game. Each unit that will be defined for nations in subsection B.2.3 Nations is required to use exactly one of the unit types defined in this file.

Assigning a unit type, which is not defined in this file, to a unit, should never be done. The reason for that is, that combat uses the definitions to be able to calculate the power of both sides. If the type of unit cannot be found in definitions, its power will be evaluated to be none and the unit will have no effect, which will cause it to die.

B.2.3 Nations

Nation definitions are found in **Nations** folder. Each sub-folder specifies name of a nation that can be picked and played by players. Each of these sub-folders should contain last set of sub-folders that need to be called **Buildings** and **Units**. These two folders contain only XML files which we can modify, remove and add new ones. However, these files have more complex structure.

Definitions of both units and buildings use resource cost and unit type definitions from previous subsection B.2.1 Game Resources and subsection B.2.2 Unit Types. As mentioned before about unit types, the names of unit types need to strictly follow the definitions. Same applies for the names of costs of resources when specifying building and unit resource costs.

Buildings

The corresponding folder can have an arbitrary amount of XML files. Each file provides definition of one building. The buildings in the game are being shown in alphabetical order based of the file names.

We will list and describe 17 basic attributes that are required to be specified for each type of the building. There are other attributes that have to be specified depending on the choice of building type, which we will also mention in their description. An example of a file with definition of a building that combines two types of building can be seen in figure B.3, which also shows an example of configuration of the common attributes. The list of the 17 basic attributes is as follows:

1. **CurrentLevel** specifies the starting level of this building in new cities.

2. **MaxLevel** sets the maximum level the building can be upgraded to. The minimum is always set to zero.

```
  <?xml version="1.0" encoding="UTF-8"?>
- <Building xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <CurrentLevel>1</CurrentLevel>
    <MaxLevel>35</MaxLevel>
    <BuildingName>Lumberjack Hut</BuildingName>
  - <InitialCost>
    - <Resource>
        <Name>Wood</Name>
        <Amount>30</Amount>
      </Resource>
    - <Resource>
        <Name>Stone</Name>
        <Amount>65</Amount>
      </Resource>
    - <Resource>
        <Name>Steel</Name>
        <Amount>35</Amount>
      </Resource>
    </InitialCost>
    <InitialPopulationCost>1</InitialPopulationCost>
    <InitialUpgradeTimeInSeconds>10</InitialUpgradeTimeInSeconds>
    <PerLevelCostIncrease>0.25</PerLevelCostIncrease>
    <PerLevelBonusIncrease>0.15</PerLevelBonusIncrease>
    <PerLevelUpgradeTimeIncrease>0.35</PerLevelUpgradeTimeIncrease>
    <IsUnitTrainingBuilding>true</IsUnitTrainingBuilding>
  - <TrainableUnitTypes>
      <string>Woodcutter</string>
    </TrainableUnitTypes>
    <IsProductionBuilding>true</IsProductionBuilding>
  - <InitialProduction>
    - <Resource>
        <Name>Wood</Name>
        <Amount>4</Amount>
      </Resource>
    </InitialProduction>
    <IsMarketPlace>false</IsMarketPlace>
    <IsStorageBuilding>false</IsStorageBuilding>
    <InitialCapacity>0</InitialCapacity>
    <IsPopulationBuilding>false</IsPopulationBuilding>
    <InitialPopulation>0</InitialPopulation>
    <IsProtectionBuilding>false</IsProtectionBuilding>
    <IsActionCenter>false</IsActionCenter>
    <IsConstructionBuilding>false</IsConstructionBuilding>
  </Building>
```

Figure B.3: Contents of **f_Lumberjack Hut.xml** file showing elements of the XML tree. The building in this file is a combination of two different building types.

3. **BuildingName** is the name of the building shown in game.

4. **InitialCost** can contain any amount of **Resource** child elements up to the amount of resources defined in **EstablishCityCost.xml**. The value is affected by **PerLevelCostIncrease** multiplier.

5. **InitialPopulationCost** is the base value used to calculate population requirement of each upgrade.

6. **InitialUpgradeTimeInSeconds** is the base value used to calculate time in seconds required for building upgrades and downgrades.

7. **PerLevelCostIncrease** is multiplier of **InitialPopulationCost** that increases with each upgrade.

8. **PerLevelBonusIncrease** is multiplier of building bonuses which increases

with every upgrade. Bonuses provided by a building depend on the type of building.

9. **PerLevelUpgradeTimeIncrease** is multiplier which affects the value of **InitialUpgradeTimeInSeconds** which increases with every upgrade of the building.

10. **IsUnitTrainingBuilding** specifies whether the building allows training of units. Setting value of this attribute to *true* enables an element called **TrainableUnitTypes** which can contain arbitrary amount of child elements of **string** specifying names of unit types the building can train. The **PerLevelBonusIncrease** multiplier increases the speed at which the units can be trained.

11. **IsProductionBuilding** specifies whether the building is capable of producing resources. Setting value of this attribute to *true* enables element called **InitialProduction** which can contain arbitrary amount of child elements of **Resource** with names matching names from **EstablishCityCost.xml**. The **PerLevelBonusIncrease** multiplier increases the production per second.

12. **IsMarketPlace** allows the building to exchange resources. The **PerLevelBonusIncrease** multiplier changes the exchange ratio in the favor of player. This type of building can always exchange any type of resource for any other resource type.

13. **IsStorageBuilding** specifies whether the building can store resources. Setting value of this attribute to *true* enables element called **InitialCapacity** which can have arbitrary amount of child elements of **Resource** with names matching names from **EstablishCityCost.xml**. The listed resources will be the resources that can be stored in the building. The amount of stored resources is affected by value of **PerLevelBonusIncrease** multiplier.

14. **IsPopulationBuilding** enables the building to increase the population limit. Setting value of this attribute to *true* enables element called **InitialPopulation** which specifies the base amount of population it provides. The amount is affected by value of **PerLevelBonusIncrease** multiplier

15. **IsProtectionBuilding** makes the building increase the defensive capabilities of units defending the city. The bonus is equal to the value of **PerLevelBonusIncrease**.

16. **IsActionCenter** specifies, if the building allows changing name of the city or view details of ongoing actions in which the city is involved, including calling back reinforcements from other cities. Every city needs to have exactly one building of this type.

17. **IsConstructionBuilding** gives the player options to upgrade and downgrade buildings in the city. The value of **PerLevelBonusIncrease** multiplier increases the speed at which buildings can be upgraded and downgraded.

Units

The corresponding folder can have an arbitrary amount of XML files. Each file provides definition of one kind of unit. The units in the game are being shown in alphabetical order based of the file names.

```
<?xml version="1.0" encoding="UTF-8"?>
- <TrainableUnit xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <UnitName>Spearman</UnitName>
    <UnitType>Infantry</UnitType>
    <UnitCount>0</UnitCount>
    <QueuedCount>0</QueuedCount>
  - <Cost>
    - <Resource>
        <Name>Wood</Name>
        <Amount>60</Amount>
      </Resource>
    - <Resource>
        <Name>Stone</Name>
        <Amount>25</Amount>
      </Resource>
    - <Resource>
        <Name>Steel</Name>
        <Amount>10</Amount>
      </Resource>
    </Cost>
    <PopulationCost>1</PopulationCost>
    <TrainingTimeInSeconds>8</TrainingTimeInSeconds>
    <Attack>12</Attack>
  - <Defenses>
    - <TypeWithPower>
        <TypeName>Siege</TypeName>
        <TypePower>5</TypePower>
      </TypeWithPower>
    - <TypeWithPower>
        <TypeName>Cavalry</TypeName>
        <TypePower>50</TypePower>
      </TypeWithPower>
    - <TypeWithPower>
        <TypeName>Infantry</TypeName>
        <TypePower>8</TypePower>
      </TypeWithPower>
    - <TypeWithPower>
        <TypeName>Woodcutter</TypeName>
        <TypePower>5000</TypePower>
      </TypeWithPower>
    </Defenses>
    <MovementSpeed>10</MovementSpeed>
    <SalvagePerEachResource>80</SalvagePerEachResource>
    <DestroysDefenseBuildings>false</DestroysDefenseBuildings>
    <CapturesEnemyCities>false</CapturesEnemyCities>
  - <BuildingRequirements>
    - <Requirement>
        <BuildingName>Barracks</BuildingName>
        <RequiredLevel>1</RequiredLevel>
      </Requirement>
    </BuildingRequirements>
  </TrainableUnit>
```

Figure B.4: Contents of **b_Spearman.xml** file showing elements of the XML tree.

Unlike buildings, attributes of units are all the same for each type of unit. Specifying the type of unit only affects the way they interact with other units during combat. An example of unit definition can be seen in figure B.4. There is a total of 12 attributes that have to be configured for each kind of unit and

a **UnitCount** attribute which is intended for debug purposes only and should always be set to zero.

- **UnitName** is the name of the unit that will be used and shown in game.
- **UnitType** is the type of the unit. The value has to match exactly one value from the unit type definitions in file **UnitTypes.xml**.
- **Cost** can contain any amount of **Resource** child elements up to the amount of resources defined in **EstablishCityCost.xml**. Defines the amount of resources required for every unit of this kind.
- **PopulationCost** specifies the amount of population required by every unit of this kind.
- **TrainingTimeInSeconds** is the base time required to train one unit of this kind. The time decreases with higher value of **PerLevelBonusIncrease** of building that trains the unit.
- **Attack** is the attack power of the unit.
- **Defenses** is an element which can contain an arbitrary amount of child elements of **TypeWithPower** which specifies the amount of defensive power against unit type of given name. When a type of unit is not defined in the child elements, that unit type becomes invisible for this unit in combat and they will not interact with each other. This allows creating unit types similar to spies in *Travian*.
- **MovementSpeed** is the speed at which the unit moves between cities. Higher speed leads to lower time required to attack or support a targeted city.
- **SalvagePerEachResource** specifies the amount of resources of each type this unit can steal from enemy cities upon victory. This value is specified for one unit of this kind.
- **DestroysDefenseBuildings** grants the unit the ability to target and damage enemy protection buildings during attack.
- **CapturesEnemyCities** grants the unit the ability to capture entire enemy city if at least one unit of this kind survives the attack.
- **BuildingRequirements** is an element which can contain an arbitrary amount of child elements of **Requirement** which specifies name of a building and only building with a required level of that building. A unit that has such requirements cannot be trained unless all defined requirements are met.

B.2.4 Event Tiles

A folder called **EventTileTemplates** contains definitions of various events that can be generated and added to the game world. They show as a tile, typically with a different picture than empty tiles or tiles with city.

The events can be observed by players. Depending on the description of the event, the player can decide whether he wants to take advantage of it. Events usually provide additional resources for players who attack or completely annihilate the tile. Once the event is annihilated by defeating the last unit present on the tile, the tile becomes empty and can be used by anybody to establish a city on that position.

It is possible to add arbitrary amount of files containing information about events to the **EventTileTemplates** folder. Each file combines attributes from subsection B.2.3 Nations and new attributes that are specific for events. There are 14 attributes that need to be configured, other attributes should always be kept on their default values. Because the definition of event templates is very complex, the definition of each attribute will be mentioned without many details about the structure of the XML documents. Using the examples provided by the framework, it is possible to see the structure of these definitions.

1. **Owner** is the name of the tile owner. Can be a completely made up name and is only intended as visual component of the tile.
2. **TileName** is the name of the tile itself. Just like name of the owner of this tile, the purpose of this attribute is intended to be shown to players.
3. **CityNation** is the name of the nation of this tile. Another attribute to be shown to players with no practical use.
4. **Description** is a text that shows to player. This is another visual component of the tile, which can be used to provide additional information.
5. **ImageName** a name of the image that should be used to show the event tile on the world map.
6. **Buildings** can contain arbitrary amount of child elements of **Building** definitions of completely new and unique buildings. It is mandatory to define a building with **IsActionCenter** property and other buildings that have an effect are those that produce resources.
7. **OwnedUnits** can contain arbitrary amount of child elements of **TrainableUnit** definitions of unique units specific for this event template. The definition of units and buildings is identical to the structure seen in subsection B.2.3 Nations.
8. **AvailableResourcesMinMax** contains an element of **Min** and **Max**. Both of these elements need to have exactly as many child elements of **int** as the number of elements in **Resources** attribute which is similar to definition of resources owned by cities. The values represent ranges of resources that are stored in the event tiles once they are generated.

9. **AnnihilationMinMax** has the same definition as **AvailableResourcesMinMax**. These values are used to generate the values for **AnnihilationReward**. The definition of annihilation rewards is the same as of **Resources**. The rewards are given away to player who defeats the last unit of the generated event tile.

10. **PerKillMinMax** has similar use as **AnnihilationMinMax**, however, the rewards are given away for every killed unit present in the event tile.

11. **Regeneration** contains definition of the regeneration period. The regeneration period specifies how often the generated event tiles of this kind regenerate and how many units of each kind should appear back every time a regeneration process starts.

12. **AppearanceChance** specifies the chance of the event tile appearing whenever new event tiles should be added to the game world.

13. **InitialSpawn** specifies the amount of each kind of unit immediately after the event is added to the game world.

14. **ProbabilityOfPresence** adds another random factor to the event tiles. When a tile is generated, depending on the values of this attribute, event tiles generated by the same template can contain different kinds of units..

B.3 Image Configuration

In subsection B.2.4 Event Tiles we have seen that it is possible to add customized images to the event templates. This can be done, because the server sends pictures to each client that connects to the server. Images that are being used and sent are stored in folder called **Images** in the root folder of the framework. The default game provided by the framework contains five images shown in figure B.5.

Figure B.5: Preview of all images included in the default game provided by the framework.

It is possible to modify, add and remove images from this folder depending on whether they are used in the modified game or not. When the game is configured properly, the client will be able to process and show the new images in the game window.

C. File Attachments

In the attachments of this thesis, we included an archive containing following data.

1. **Developer Documentation.chm** contains documentation of the code.
2. **Source Code** contains the source code of this project. Also contains the implementation of Simulator and Game Generator projects.
3. **Experiment Results** contains the information about all 66 tested game configurations from all 100 runs for each game. Also contains the summarized information of the three picked game configurations to test our hypothesis in the thesis.
4. **Generated Games** contains configurations of 66 tested games used in experiments.