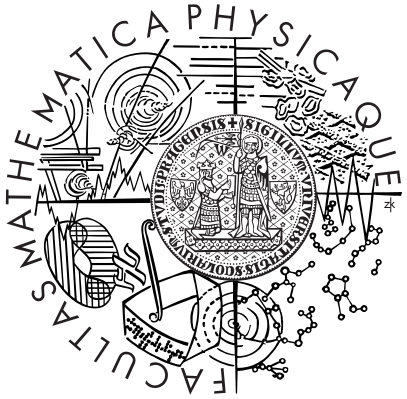
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Filip Ressler

Wildmen: strategická hra v terraformovatelném světě.

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Programování

Praha 2014

Na tomto místě bych rád poděkoval všem lidem, kteří se podíleli na této práci. Především děkuji mému vedoucímu, Mgr. Pavlu Ježkovi Ph.D., za ochotu, trpělivost, připomínky ohledně struktury a za pomoc při psaní této práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 31.7.2014

Název práce: Wildmen: strategická hra v terraformovatelném světě

Autor: Filip Ressler

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Dnešní real-time strategie jsou založeny na konfliktu znepřátelených frakcí, kde proti sobě bojují armády jednotlivých hráčů. Nicméně, tyto strategické hry neobsahují jednotku, která by reprezentovala hráče. Takovou jednotku, která by měla schopnost měnit terén mapy, používat silné útoky na nepřátelské síly a jejíž ztráta by pro hráče znamenala prohru.

V této práci prezentujeme 2.5D real-time strategickou hru používající jazyk C# a .NET Framework, kterou lze hrát po síti s dalšími hráči. Tato hra se zaměřuje na schopnosti kritické jednotky reprezentující hráče, její útoky a schopnosti měnit terén. Tento projekt umožňuje upravovat a přidávat jednotky, budovy, zdroje a schopnosti jednoduše pomocí editace souborů XML, souborů obsahujících zdrojový C# kód a souborů s texturami.

Klíčová slova: Real-time strategie, multiplayer hra, terraformace mapy, modifikovatelná hra

Title: Wildman: A real-time strategy game with world terraforming features

Author: Filip Ressler

Department / Institute: Department of distributed and dependable systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D., Department of distributed and dependable systems

Abstract: Nowadays real-time strategies are based on the conflict of enemy fractions, where the armies of the players fight against each other. However, these strategic games do not have a unit which would represent the player on the battlefield. A unit which would have the power to alter the landscape or use powerful attack against the enemy forces, but a unit which loss would mean defeat for its player.

In this thesis we present a 2.5D real-time strategy using the C# language and .NET Framework, which can be played with other players over the network. This game focuses on the abilities of the critical unit representing player, its attacks and abilities to alter terrain. This project allows simple modification of the game or addition of a new content by editing XML files, files with C# code and image files.

Keywords: Real-time strategy, multiplayer game, map terraformation, modifiable game

Table of Contents

1 Introduction

A real-time strategy – or RTS for short – is a type of video game where the player assumes control over a faction of people and leads them towards victory. It is a game where one can test their tactical abilities by carefully thinking out their tactics and trying to be one step ahead of the enemies. Player must also be ready to make fast decisions in the middle of the battle in order to tip the favor to his or her side. However, if the decision turns out to be a bad one, the player might lose the battle and ultimately the game.

Nowadays, there are many RTS games, for example StarCraft [1] or Age of Empires [2], which are excellent games where one can spend many hours of their free time. However what most of these games have in common is, that the game doesn't really end until all buildings of the enemies are destroyed. Or, to be more precise, until all buildings that can ensure that opponent can come back are destroyed. But what we would like to have is an RTS where the battle is even more intense.

For example, the Age of Empires added a game type, where the objective was to eliminate the kings of all other enemies while keeping them from killing yours. The king was not in any way a powerful figure – it was (combat wise) weak unit that was best kept in the castle surrounded by thick walls. However, if we invert the situation, we might get some interesting results. Let us make our critical unit a superhero, a person of extraordinary power which would make him an excellent unit to be thrown into the fray of battle. This concept was already explored some time ago in the game Populous: The Beginning [3] by adding such critical unit – a shaman that leads its tribe with strong and powerful spells.

We would like to re-examine the same concept, since there are not many recent games with this mechanism. Our goal is to make an RTS game where our central unit can call down a rain of fire on the enemies or summon a big volcano where the vanguard of the enemy army was just a few seconds ago. A unit with spells that can rise and sink land and allow armies to cross lakes and seas. But also a unit whose death would mean that comeback will be really, *really* difficult.

In the rest of the chapter we will summarize the basic concepts behind our game and in order to do that, we first have to clarify what the RTS genre actually is and what are its key components.

1.1 RTS genre

When we look at the name of the genre – real-time strategy, it is clear that the game will be played out continuously. That means the orders we give will be carried out as soon as possible (for example, as soon as the building finishes current construction or as soon as the unit arrives at given position) unlike in turn-based games, where players take turns and the game won't continue until all players gave the orders to game objects under their control – until they finish their turn. That means that whatever we do will not only rely on our overall strategy, but we have to pay attention during the entire game so we will not get unpleasantly surprised.

Now let us move to the second part of the name, strategy. The Wikipedia on topic of strategy says "[a] s*trategy is a high level plan to achieve one or more goals under conditions of uncertainty. Strategy is important because the resources available to achieve these goals are usually limited.*" [4] Which perfectly matches what RTS games are. At the start we are presented with basic building and few units and we are left to make our own plan how to win the game without knowing anything about the opponents' intentions. And our resources might be what we find around the map in form of game object, that needs some form of interaction from our units, or something we will get periodically from the game, but our most important resource is time, which is generally vital for defeating the opponents.

However, such description of a genre not only matches RTS games, but also few others which would not necessarily fall under this category. And those games would be for example simulation games. Let us look at the SimCity [5] series. SimCity games are based on building a thriving city from scratch. At the start you are lent money and you use them to build a town which will gradually grow into a big city. Or not, depending on your abilities and your own goals. But the point is, that the town you are building takes time to grow, so this is a real-time game. Also, you are deciding what to build, where to build and when to build, therefore you are making a plan for your city. And finally, your resources are the money. So, if we used the description we figured out from the name of the RTS genre, we would have to consider SimCity one of the games of the RTS category. However SimCity, by conventional understanding of the video-games genres, would be a city-building simulator. Therefore, we need to specify the genre a bit more.

The problem here is the fact, that while we call these games strategic it's more often than not a tactical game. Chris Taylor, a game designer who worked on RTS games like Total Annihilation or Supreme Commander said "*[..] realizing that although we call this genre "Real-Time Strategy," it should have been called "Real-Time Tactics" with a dash of strategy thrown in.*" [6] What he meant is that the only way how to defeat the enemy is by attrition – to wear down the enemies, before they wear us down – one can't win just by diplomatic means. Players have to build a force with which they will defend themselves from the opponents and eventually, in order to win, these forces will have to engage one another and at that point the one with better tactics (or luck) will come out of the conflict victorious.

If we were to sum up the key components, we would get following list:

(C1) The game plays out in real time.

(C2) The game features resources players have to acquire and utilize.

(C3) The game requires player to build and maintain control of a force that will bring down enemies. This includes dealing with the issue of having diverse army, staying within population limit and being able to use abilities of special units at correct times.

(C4) There is a condition that will declare one of the player or team a winner of the game.

Additionally, RTS games might include following features:

(C5) Player can progress and upgrade his or her forces throughout the game. It might be enabling additional build orders or improving of existing units that occurs

once player builds certain building or spends a resources on a "research." Additionally, another form of a progress is idea of unit getting stronger after each battle.

(C6) Introduction of a concept of a base, where different types of buildings are being constructed and the possibility of expanding the base.

Now we will go through these components (C1) – (C6) individually.

1.1.1 Real-time play

The real time aspect of the game (C1) ensures the players will play simultaneously. That means, as mentioned earlier, all unit orders are carried out immediately. Only way a command can be delayed is by the execution of queued earlier actions and by the timeouts that are there to prevent from continuous stream of the same actions and are simulating the time unit or building needs to prepare for the ordeal.

1.1.2 Resource gathering

Resource management (C2) is another important aspect of an RTS. Rule of thumb is that players should aim for getting as much resources as they can, because the entire economy is based on them. Without resources you will not be able to build or research anything and usually the player that is able to gather more resources will have higher chance of success.

The resources are usually represented as objects on a map which have to be collected by special type of unit and the same unit will then bring the resource back to the gathering point. The resource can be either permanent source which will never deplete or it can have a limit how many times it can be harvested before vanishing. This mechanism will add the need for looking for another resources to keep player's economy running.

For example, let us consider wood as a resource – a lumberjack (the special unit) first has to go to the forest (the resource), then over some time cuts down the tree (collecting the resource) and then brings it back to the sawmill (gathering point). We might consider that a new tree will grow immediately after we cut the old one down (simulating permanent resource) or that once the tree is cut it will not reappear (a resource that will vanish after set amount of collections).

The economy also doesn't have to be restricted only to one resource – there might be as many resources and types of their sources as the developer desires. The problem here is, however, that more resources adds to the complexity of the game. Having too many kinds of resources might lead to confusion of the player, not to mention that they will have to be placed somewhere on the map, which will make it more cluttered.

Additionally, one might consider time as a resource as well. The main reason is that everything you build or research usually have a timer that will start when player gives order and the requested order is available once the timer runs out.

In our game we will consider two resources randomly spread on the map that will be gathered by a base unit and will be depleted after given amount of collections and will force players to seek out new grounds for their bases and will eventually introduce more conflicts.

1.1.3 Unit management

Unit control (C3) is a core mechanism of an RTS. Units won't do much without player's interaction, safe for self defense when they are attacked. Players have to give them orders in order to utilize them and this is where the tactics comes into play.

It's important to keep army diverse in order to ensure the best efficiency. Different types of units have their advantages and disadvantages against other kinds of units. For example, an archer would be perfect against slow warrior, since the archer can pick warrior off before he comes close. However, archer will not be as successful when attacking shield-bearer or fast unit like cavalry. Therefore, if army consists only of one type of unit, it's easy to produce a counter unit, defeating the entire force with ease.

The diversification also goes in hand with the population limit. The goal of population limit concept simulates limited amount of personnel available at a time and exists to keep armies to reasonable size. Otherwise, at some point it might seem easier to build an absurd amount of units which would “roll over” just anything.

Unit management becomes even more difficult when we add one or more abilities to units, that have to be activated manually by player. Also, it makes the potential skill ceiling very high as it becomes more and more difficult to handle all units and their abilities in combat.

Our goal is to have several types of units and with exception of the basic unit they will have bonuses and disadvantages against certain other types of units and attacks. Some types of units will have special abilities that couldn't be controlled by player and shaman – the central unit will be able to use spells controlled and directed by player.

We will have two kinds of population limits. First one will be dynamic limit that will be directly dependent on number of house buildings. These buildings will increase the limit when they are built and decrease it when they are destroyed. If the population limit has not been reached yet (the number of current units is lower than the limit) a house buildings will start producing basic unit until the limit is reached.

The second population limit will be maximum amount of units a player can have and can only be changed when the game is being created. Reason for this threshold is the absence of any limit on the number of house buildings increasing the dynamic limit, which would defeat the original purpose of the population limit.

1.1.4 Victory conditions

As we mentioned earlier, the victory conditions (C4) revolve around military domination over the game battlefield, but while the best way is to destroy all enemies there can still be variance in the conditions to some extend.

The first and most straightforward way would be, as we just said, to eliminate everyone else simply by destroying all of their buildings. That is the most usual victory condition. Another way how to win can be a building race to special building like the wonder construction in Age of Empires 2, where the goal is to build this expensive wonder that takes very long time to construct. And while one might argue for this to be very non-military strategy, we have to consider that usual counter-move is to take the army and simply march on and destroy this very construction.

Next example of victory condition is named King of the hill (which is also featured in Age of Empires 2) and the goal here is to control given areas for specified amount of time. And as the last but not the least example we have the Assassinate the king game mode, where the game is won by killing a special unit of each enemy while keeping your special unit alive.

Our game will be revolving around the idea of a central unit – a shaman. Every player will have one shaman that will be available from the start and if player doesn't have shaman, one will be produced in central building. The goal of the game is eliminate all shamans of the opponents and the means of their recreation.

1.1.5 Progression

The idea of progression (C5) goes in hand with the base building. The motivation is that as the game plays out, the game should progress, effectively delaying the pace of the game. At first there will be basic buildings and weak general purpose units. Later, after certain requirements are met (usually in form of constructed building or a research in the building) another set of build orders is available for player, allowing the construction of better buildings and specialized units.

However, it's important to mention that it doesn't mean that the conflicts are postponed to the later stages of the game. Example might be a surprise early attack which itself introduces a problem if the opponent is either underestimated or overestimated, which would lead to significant economical throwback.

Another form of progression is unit upgrading. Most common ways of doing that are by research and by combat experience. Researching an upgrade in a building usually applies upgrade to all existing and future units of given type. It might be in a form of boosting base attribute like health or attack strength, or in a form of additional abilities. Such upgrades aren't very cheap but they can ensure that early game units can stay relevant in the game for longer time.

Other way of improving unit is via its experience in combat. The more enemies the unit dispatches, the better it gets. Making individual units more powerful will make player treasure these experienced units and add another layer of complexity for unit management.

1.1.6 Base building

The concept of base building (C6) is based on having a central point – a headquarters if you will, from where we will expand and control our mission. It is a place where we concentrate our production and research facilities, and as such needs to have good defense, since these buildings are usually expensive.

Not all RTS games have the base building part, for example Command & Conquer 4: Tiberian Twilight [7] introduced a super-unit, that would serve as a self-powered production facility that requires no resource. While some might argue that the unit is the base itself, it still lacks many aspects of the traditional base like the entire concept of resource gathering and the build order of the buildings.

Base building is usually where the first major deciding takes place. We have to decide which buildings we need the most at the moment and will build them first and which

buildings can be delayed. In following sub-chapter we will describe four main types of buildings available to players with specific abilities that are most common in RTS games.

Building types

- Resource gathering buildings either produce resources or serve as gathering points, where resources needs to be carried to. Both cases make these buildings one of the most important, since if player loses control of them it will throw back his or her economy giving the opponents an edge and if the player loses all of these buildings, they most likely lost the game, since the entire economy will lose the influx of resources and they won't be able to produce anything.
- Production buildings, as the name suggest, is used to create products, in this case new units. These will build your army and it's important to strike the right number of these buildings, because one's economy might have limited put-through. If player builds much more production facilities than the economy can efficiently support, it would mean that player has wasted the resources they have spent on the construction of this building.
- Research buildings are core to the development of your base and units, they allow you to upgrade their abilities, basic attributes or provides you with additional build options. While in short run these buildings are usually expensive, in the long run the bonuses they provide usually decide the battles.
- The defensive structures helps in the defense of the base and are important for holding off enemies until defender's units reach the site and supporting them. Additionally they might provide better sight radius – an area around the structures/units that is revealed. Well placed tower might reveal sneak attack early and rises defender's chance of fending off the attack.

Base expansion

In order to prevent the players from building structures anywhere they want (like in the middle of the enemy base) and to simulate logistics within the base a restriction has to be implemented. One way is to simulate the logistics directly. This can be illustrated in the Settlers [8] game series where a path has to be built between buildings and resources are carried by people following this path. A different and more common way is by setting a restriction allowing new buildings to be built only in given radius around a central building. This approach to the problem will force players to think carefully where to place it and how to organize the buildings they want to build inside their bases and the defenses. Alternatively, the game might limit the buildings only partially and only restrict key buildings like resource gathering structures.

Both these attempts will ensure, that players will have to expand their primary base if they want to be more efficient. That means the player will have to build a special structure – an outpost or a new center, that will serve as a heart of a new secondary base and will allow for constructing new buildings nearby or serve as a new gathering point for resources.

In order to build these expansions player must dedicate substantial amount of resources to afford the new base. And because of the new base's cost it's important to secure it, which means that the player now has two bases to defend from potential attacks bringing many

problems with the unit distribution between these bases and increased need for defensive structures. What this means is it is best to build the expansions near to the primary base.

However, there are few reasons why to consider remote expansion. Depending on the map design, there might be places on the map, that are either in the middle of everyone's sight or are very distant, that offer rich resource fields. Such expansion would then be both high risk and high reward endeavor. Other case would be a distant advanced base near enemy, which if kept undetected will allow for gaining upper hand by the possibility of surprise attack from this position.

For our game we will have a central building that would define the radius of our base where we will be able to place new buildings. The central building will also allow re-spawning of our critical unit and will also serve as a gathering point for resources. In order to keep its exclusivity, we will have an outpost building that will have same function except for the re-spawning part, so that we can expand, but keep the importance of the primary base.

Next base building will be a house, which will increase population limit and where new basic units are produced if the limit is not reached. We will have training buildings that will allow specialization of the units. We will have research building that once built will enable abilities and additional buildings.

1.2 Our goals

In this chapter we summarize what we will try to accomplish in this project.

1.2.1 Game elements

Here we will go over the main elements of the game that are interacting with the players.

(G1) Victory condition

The game will revolve around the central unit – a shaman. Every player can have only one such unit and a central building that will allow for this shaman to respawn after a while. If both the building and the unit are destroyed, player loses the game. The last player in game is victorious.

(G2) Map

Our game will play out on a two dimensional map. The areas on the map will have certain height which will determine whether it's a land or water and depending on the surroundings cliffs or very steep slopes may occur. These obstacles will separate players and will require terraforming abilities that will allow changing heights.

Units will be able to walk across the map on the land and will not be able to step into the water or into steep slopes. Buildings can be placed on a flat land terrain.

(G3) Resources

Our resources will be placed on the map and will be gathered by units. Every resource on the map will have counter how many times it can be collected and once the counter reaches zero, the resource will be depleted.

Unit will have to come to the resource in order to collect it and then will have to return with it to the collection point – a central building or outpost. Also, after resource is collected,

it will have a cooldown before next collection can happen. This will encourage gathering from more sources.

(G4) Buildings

The buildings in this game have to be constructed on flat land terrain. First, a unit will initiate a construction, provided player have sufficient amount of resources, and then player has to order a unit or more to spend time building this structure. The building can not be used until it is constructed.

The game will have following buildings:

- The central building will serve as a resource gathering center and will respawn central unit – a shaman after given delay.
- The house will increase current population limit. It will produce basic units until the limit is reached.
- The stockpile building will have only function - to serve as a resource gathering point, but will be cheaper compared to the central building.
- Training buildings will allow specialization of units. Once a unit will enter the structure it will re-emerge as different type of unit after a delay. This process will cost resources.

(G5) Units

The basic unit is automatically produced in the houses until the population limit is reached. These unit can do most tasks like resource gathering and building other buildings. This unit can be then upgraded in the training buildings for given resource price.

Various types of units will have bonuses against other types of units forcing the players to diversify their armies.

(G6) Shaman unit

The shaman unit behaves as regular unit with exception that it can cast spells. Once killed, it will start re-spawning in the central building. If there is no central building and the shaman is killed, the player loses the game.

(G7) Spells

The spells are effects that may affect building, units and terrain in specific way. They might change certain properties of the objects, damage or heal them, they might rise or lower the terrain. They will have large impact on the game and will be available only to the shaman unit.

Players will be able to “unlock” various spells throughout the game in order to limit their power in order to delay the game progress.

1.2.2 Under the hood

This chapter will summarize the concepts which aren't directly related to the game design.

(G8) Multiplayer

The game is a multiplayer game – more players will be able to concurrently control their units and buildings and interact with other players in the game.

(G9) Extendability of the game

The game will be modifiable without the need to recompile the entire game. It will be possible to add, change or remove units, buildings, spell or parts of them through editation of external files.

(G10) Programmed in C#

The game will be programmed in the C# language with use of the Microsoft .NET Framework.

2 Problem analysis

Now, that we established our goals and expectations we have to look over their possible realizations and implementation details. We split the problems into two categories. In the first category there are the gameplay-related problems, where we describe problems concerning the game objects (units, buildings and resources), the map and the spell system. Second category is about the technical issues that are not directly related to the gameplay like network communication, graphics an extendability of the project.

2.1 Gameplay design

In this sub-chapter we will look at the game design details and possible issues that might occur. We will take a look at how the map will be represented and how will be generated, as mentioned in goal (G2). Then, we consider our options for the representation of the spells – goal (G7). After that, we analyze what information we need to store about the map (G2), the spells (G7) and the objects (goals (G3) through (G6)). And at last we will take a look at the victory conditions (G1).

2.1.1 Map representation

We have to decide how we are going to the represent our game-board. What we know from goal (G2) is that our game-board will be a map where the game objects will be placed and interact with each other. The goal specifies that the map shall be two-dimensional and should support elevation. However, if we are to implement our map, we will need more details about the map. Specifically, we will have to decide how the map will be restricted and how the elevation of the map regions will be handled.

Restriction of the map

A map may or may not be restricted and by restriction we mean that the map will have given width, height and depth.

An unrestricted map is a map where the size is virtually boundless. We say virtually, because we will always be subject to the limitation of the data representation or computer storage, but for all rational intents and purposes it is safe to assume that the world does not have a limit. The map is not created completely at the start, but is procedurally generated as the players explore the world. However, this style of map representation is not very suitable for a competitive strategy as the layout of the map does not allow to completely contain the players. It is more suited, as we already hinted, for an exploration or building games where the space restriction limits the creativity. Example of a game with limitless map can be Minecraft [9].

The other way of map representation is a restricted map, which has its size determined at the start of the game and nothing can be built or moved beyond this boundary. It allows to contain the gameplay in one area and ensures the tension and conflicts between the players. The limit on the dimensions allows us to create the map at the start of the game and allows us to use less dynamic data structures for representation of the map. In case of restricted map

we can chose to implement a map-wrapping which allows to simulate a globe by "connecting" the ends on one axis of the mapboard.

Considering our goal (G1) about victory by elimination of opponents meaning that players have to engage one another to win, and the goal (G3) saying the resources are limited, we will pick the second approach which is limited in both available space and number of resources.

Terrain elevation

Our game has to allow terraforming (goals (G6) and (G7)), therefore we have to represent an elevation of the terrain. We have to consider that we will need several elevation levels for the terraforming spells and that we will need to check for an elevation differences for building placement or for determining accessibility mentioned in goal (G2).

- One way of simulating elevation is through impassable game objects that will represent the obstacles that occur in real elevation maps, like for example cliffs or coastal areas. While the areas on both sides are at the same level as the elevation information isn't stored anywhere, the obstacle can make an illusion of elevated terrain on one side. We can find example of this implementation in the game Age of Empires II, where water and land tiles are on the level and the inaccessible cliff only makes an illusion of the elevation.
- A different implementation is to assign the elevation value to every point on the map, which allows for better representation of small height differences on the map. The resulting array of values is called elevation map. An example is on the figure 1 where on the left side the elevation value is represented by shade of gray, on the right side is the rendered terrain. This approach makes the transitions between levels smooth, however, checking the elevation in an area would force us to go through all the points in the concerned place.

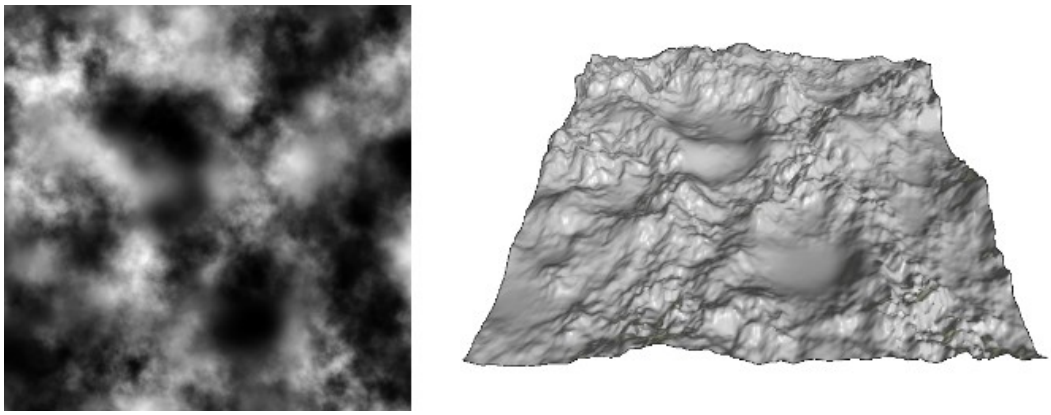


Figure 1: Elevation map on the left side as a 2D bitmap, on the right side as a rendered image. (reprinted from [10] and [11])

- The third way is a simplified version of the second case. The map is split into small areas called tiles which hold information about themselves - in our case it would be mainly their elevation. The tile system also simplifies the way how the map is drawn and how the game objects interacts with it. For example in case of buildings we only have to check the elevation of few tiles. Example of a game with this mechanism is

Chaos Reborn, with a screenshot from early version of the game on the figure 2 where we can see hexagon tiles with various elevations.

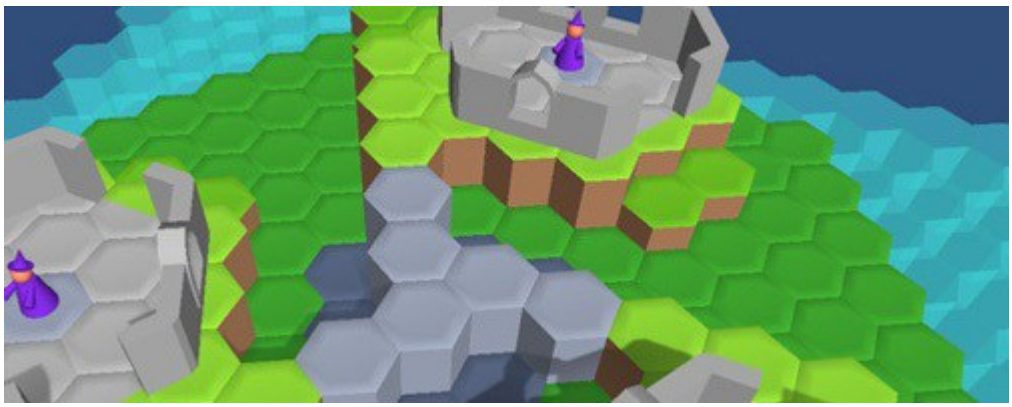


Figure 2: Representation of tiles in alpha version of the game Chaos Reborn. (reprinted from [12])

The first approach is simple, however it isn't ideal for game with more than few elevation levels as the game would get cluttered with these "elevation" objects as a result. The per-point elevation system as described in second case is not ideal as the checking for slope differences and determining whether unit or building can be on this place will require checking of every pixel in area. That is why we will pick the tile-based representation of the map as it will be best for our terraforming oriented game.

2.1.2 Map generation

Now, that we know how we will represent our map we have to figure out how the map data will be generated. In order to fulfill our goal (G2) we need a terrain that we could move around with our units (the map should have flat areas or areas with low elevation differences), we need a sea-level forming inaccessible areas and we also need passages with steep slopes that would not allow units to pass through.

Random terrain generation

The first thought was be to generate a new random number for every tile through a random function that does not accept any parameters. We took values from already generated surroundings of the tile and then used the random number which specified variance of the terrain. The problem with this approach was that the terrain was very unpredictable, generating either terrains that did not have many flat areas, making movement around the map difficult, or terrains that were too flat.

It was clear, that using random number generator with one or less than one parameter was not enough or that it would require a lot of additional parameters and restrictions to keep the map reasonably diverse while not being completely random.

Elevation through Perlin noise generator

The (2D) Perlin noise [10] is a noise generator that takes two parameters representing the position and gives us result directly depending on these numbers. The spatially close positions would have similar values allowing for relatively smooth transitions. An example of a generated texture using a method involving Perlin noise is on the figure 3.

The results of using Perlin noise better represents a map and fulfills our requirements for both flat and steep areas.

Polygonal map elevation

Another approach to the map generation would be to split the map into polygons and then work with these objects.

The algorithm [12] will first generate a polygon map (usually by creating Voronoi polygons from randomly placed points). Then a function is used for shaping the landmass (usually using a noise map, like the Perlin noise mentioned above) and then the elevation of the tiles is determined either from distance to the sea or, again, by using a noise function. An example of a map generated through these steps can be seen on the figure 4.

Such map then can be even more improved by using a rainfall noise simulating rainfall and allowing to define rivers and lakes, and distorting the edges of the map allows to create more realistic feeling of the map.

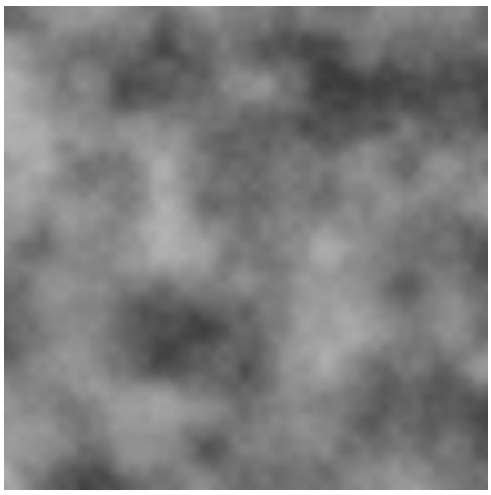


Figure 3: Example of a texture generated using Perlin noise (reprinted from [11])

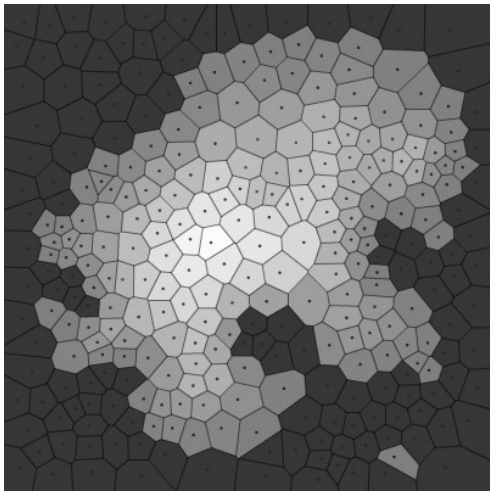


Figure 4: Example of map generated through map elevation (reprinted from [13]

Comparison

On the figure 5 we can see the comparison of the results from the Perlin noise and polygonal map generating techniques after being divided into regular cells. While the polygonal generation seems to have generated more cohesive landmass, we have to remember that the shape was taken from noise function and the elevation inside the generated island was measured by distance from water. Not to mention that using distance from water would be the best approach as this way we would not get any cliffs on the map.

Therefore, if we use the the noise function directly to the individual tiles we will save time by not computing Voronoi diagrams and smoothing functions for the polygons, and will get more unpredictable terrain within the landmasses.

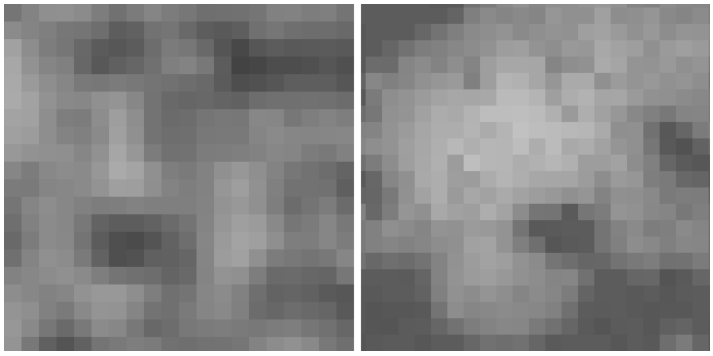


Figure 5: Comparison of height map from Perlin noise (left) and from polygonal generation (right) divided into regular cells.

Picking starting locations

After we generate the map, we have to pick appropriate location for the player to start. The player should start on a land of at least size that would allow to place several buildings around the start point. Additionally, the game should not place the player too close to avoid conflicts right from the start of the game.

In order to solve the first problem, we define zones of the map, where the zone represents all tiles that are accessible from any tile within the zone. Since we need the starting zone to be flat in all directions, we calculate for each tile its distance to the edge of the zone it belongs to. This distance represents a radius in which all tiles do have low differences in elevation and can support a building. There is a constant defining the minimum distance and if zone does not have any tile that would reach this minimum number, it is excluded from the picking process. If there is no zone on the map that would reach the minimum number, the map is regenerated.

When picking starting position for the first player we simply pick the biggest zone and we take the tile with the largest distance to the edge of its zone. This starting tile will then form a new zone and recalculate the zone-edge distances for tiles, which will cause drop of the edge-distance values in tiles surrounding the starting position.

For every other player we have to re-check whether a zone with minimum size is available. Additionally, we skip any candidate-tile for starting position that's too close to any already defined starting position.

2.1.3 Spell system

Another mechanic to explore are the spells (G7) and the process of spell casting. A spell is an effect (a process) that will alter the game or its part in some way. In our case the alteration will affecting only the game objects (units, buildings and resources) and the terrain.

We could generalize spells into two types:

- Immediate spells
- Over-the-time spells

The immediate spells doesn't present much of an issue in terms of implementation. The spell can be a function that will modify the objects and the end result will show once the function has ended. This means that the process or a mid-state of the spell doesn't have to be stored in any way.

The over-the-time spells present more complicated scenario. The spell will modify the game as long as it is active, therefore it has to be revisited every update frame. It also means that at any time the state of the spell have to be serializable if the game is stopped and the spell is still active.

In both cases, the effects will have to contain a code that will describe such process. With respect to our goal (G9), which we will examine later in this chapter, these instructions ca not be hard-coded in the game (directly programmed in the game code and compiled with the game at compile-time). This means that we will need to provide interfaces to the game elements that will allow for editing the properties of the game elements.

2.1.4 Entity data

Every game object in the game will need some information that will represent it, describe its relation to the other objects and/or map, and data that will define this object. Here we will look at the properties the units, buildings, resources, spells and map tiles will require.

Map tiles

The map tiles themselves only need to know their elevation, however, for easier handling, we will include information about it's position in the map array and on the screen. Also, tile will contain information about units currently present on top of it and the resource or building if there are any, which will allow us faster access to the information as we will not have to iterate through all units, buildings or resources to check what is placed on the tile.

Units

The unit has to contain information about it's position on the map. It will be a vector, so that unit isn't restricted to tile, but may move around freely. Additionally, it needs to know it's health, it's owner, whether it's selected or not, what order is it executing right now.

Also, in order to be compliant with our goals (G5) and (G6) it needs to know information related to the type of this unit, which are information about which texture it has, what is its name and speed, whether it can gather resources, construct buildings and/or attack enemies.

Buildings

Again as with unit, we need to store information about the position, but unlike unit, the building will be tied to tiles which the building will exclusively occupy, so we need to store info about this tile instead. We will also need to store information about the health of the building and about the construction progress of the building.

The building, as mentioned in goal (G4), will also need to know what unit it can produce or transform, what other building is required so that construction of this building can even be started, the cost of this building and whether the building can serve as a resource gathering center.

Resources

Resources will, in terms of their relation to map, behave similarly to building – they will occupy tiles. Then, by following the description of goal (G3), we need to store information about amount available in this resource and the timeout before another gathering can occur. And at last but not least, we need to store information about type of this resource.

Spells

Spells, in addition to the code that will determine what will happen at the beginning, during and at the end of the effect have to contain information about the duration of the effect, the speed how often the update is called on this spell, what can be target of the spell (unit, building, resource, tile), which building will unlock this spell and how long will it take for the caster unit to recover before it can do anything again.

Player

The player class needs to store the lists of his or her units and buildings, list of the resources and how much of them is available to the player at the moment and the color of the player.

2.1.5 Victory conditions

As mentioned in the goal (G1), we need to keep track of a shaman for each player. If the shaman is killed, the respawn process should start at the central building. If there is no central building, player have lost. The last player in game is victorious.

2.2 Technical issues

Now, that we established the needs of the game elements, we have to clear the technical problems concerning our game. These problems are the picking of the game engine, which will handle the graphics and the way how the window of the game is updated. We will also have to focus on the details of the goal (G9), the extendability of the game, specifically, the extend of the possible modifications and their implementations. Then we will have to decide the network side of the game and the connectivity between players as required by the goal (G8).

2.2.1 Game engine and graphics

If we would break the code of any game to the most basic elements, we would find that in most games there are two main functions which keep the game alive. One for updating the game and checking the user input and the other for updating the screen. In some older games these two functions might be merged into one, meaning that the game updates will be tied to the screen refresh rate. While this would be of no issue on older computers it is not ideal, because of following two problems. Either the game will update every time and at better computers we would have to limit the screen refreshing speed and not fully use the potential of the computer. Or, in the second case, we would have too many update functions called per second which would make everything in the game happen faster.

Graphical style

We also have to make decision on how complex our graphics will be. While using the 3D graphics would allow for better representation of terrain elevations, it would also mean, that 3D models of units and buildings will be required, which are fairly limiting the

extendability for less skilled users. Considering that the extensibility is among our goal (G9) we decided to go with 2D scene for our game.

However, the 2D scenes allows to have several kinds of projection – the way we look at the map. Now we will consider them and see which will fit our use.

- The top-down perspective, also known as the bird's view perspective, is the case where the camera (in our case the game window) is above the gameboard and we are looking straight down at the game objects. This will entirely leave out the third axis and the terrain would seem flat, which is not ideal for our use. Example of such perspective can be found in the game Dune 2 [14] as pictured on the figure 6.

Figure 6: Dune 2, example of top-down perspective (reprinted from [15])

- The side view, also known as side-scrolling, is very similar to the top-down one with the exception that we are looking at the game from the side of the gameboard. While this will allow us to show the elevation of the terrain, it will not be able to show more than one slice of the terrain at a time. The game Mario [16] uses this technique, as pictured on the figure 7.

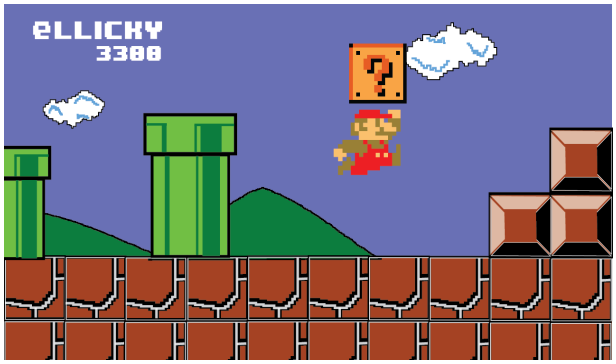


Figure 7: Mario, example of side-scrolling graphics (taken from [17])

- The 2.5D view, also known as pseudo-3D, is a mix of top-down and a side view in a sense, that the camera is rotated at given angle between these two positions. Such projection allows us to see all three axis at the cost of inability to see areas that are behind tall objects. However, if we are careful with how we represent the terrain elevations, we can minimize this flaw to acceptable level. Example of such game is SimCity 2 [5] as pictured on figure 8.

Figure 8: SimCity 2 uses the 2.5D perspective.(reprinted from [18])

We will not be able to use the side-scrolling view, since our map is two-dimensional, so our decision is between the top-down view and pseudo-3D. Since our game will have terraforming (as mentioned in our goal (G7)), we will lean towards perspective that handles the elevations better from these two options, which is the pseudo-3D view.

Game engine

Now that we clarified which graphic representation we will use for our game we have to consider which game engine to use. The reason for picking a game engine is that there is already quite a few of them and it would be unnecessary to create our own implementation as there might already be a library that would fully suit our needs. Our expectations of the game engine are as follows:

- As our goal (G10) states, we will make this game in C# language, therefore, the game engine also has to be made for C#.
- We need the game engine to handle the main game loop – calling of the update and draw functions. As we already established earlier in this chapter, these are the basic functions game a has to have.

- As we picked the 2D graphics, we would prefer if there is a relatively easy way of drawing two dimensional textures on the screen, so we do not have to rely on low-level drawing functions.
- The goal (G9) about extendability of the game will need to load additional game content that will not be known at the compile-time, therefore, the game engine should support loading new graphical content at the run-time. Preferably, it should be able to load conventional graphical formats like portable network graphics (PNG) or GIF file without any preprocessing from user side in order to make the extendability available even for less skilled users.
- Preferably, the game engine should be up to date and with proper documentation.
- The game engine has to be free of charge.

Unity3D

The Unity3D [19] is a very popular and powerful game engine. It allows multi-platform development ranging from Windows and Linux platforms to Android and iOS, making it very universal tool.

The programming is based on scenes, which contain objects that are automatically rendered and every object can have assigned either Javascript or C# script, that allows modifying the object or the game when specific event arises by overriding methods. As we can see in figure 9 in the IDE user can add additional "components" to the object that will modify its properties, rendering and the behavior towards other objects.

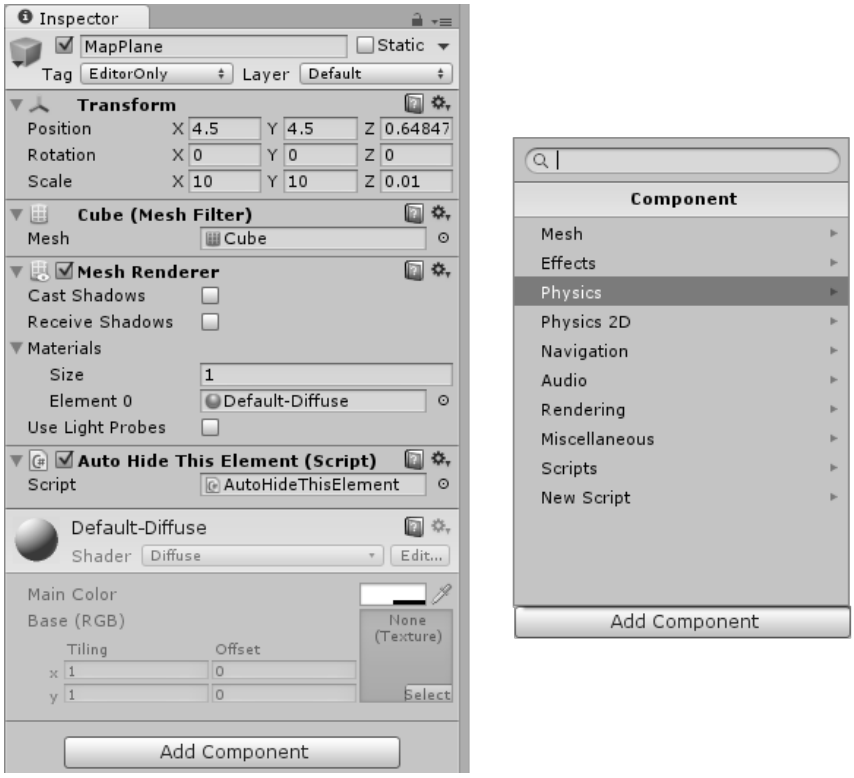


Figure 9: On the left side we see example of object properties in Unity3D. On the right side there is a context menu for adding a new property.

However, the 2D rendering is somewhat limited because - as the name of the engine itself suggests - it's designed primarily for the 3D development and the support and documentation reflect that. And while there are various 3rd party plug-ins for the 2D programming in Unity3D, they are not free of charge.

Microsoft XNA

The Microsoft's XNA Framework [20] was designed for game development for XBox 360, however, applications using XNA are compatible with other Microsoft platforms like Windows and Windows Phone.

The library provides an abstract class `Game`, that has (among others) overridable methods `Update` and `Draw`. The `Update` method is called approximately 60 times per second and in the meantime between the update calls the `Draw` function is called as many times as it can be called.

The resources can be loaded either through the `ContentManager` component, which at compile-time compiles all game resources into unified format and automatically manages them. However, it is still possible to load resources from file stream and manage them manually.

Nevertheless, the XNA libraries have been discontinued and are no longer developed which presents an issue since the last version of XNA isn't compatible with the current version of .NET.

SharpDX

The SharpDX [21] is a C# wrapper for DirectX and therefore supports most versions of Windows. Using DirectX functions directly would be difficult, but the SharpDX provides a toolkit namespace, that simulates the interfaces of XNA, which we will focus on. The toolkit does not bring any new functionality, however, unlike XNA, the SharpDX is up-to-date and does not have issues with the latest .NET libraries.

The documentation is focused mainly on the low level functions with little focus on the Toolkit, however the main reason for this is, again, the similarity with XNA and the most functions are identical in their use and functionality.

OpenTK

The OpenTK [22] is a C# wrapper for OpenGL. However, unlike SharpDX's Toolkit, it does not provide many functions for easier manipulation with textures or drawing. The texture processing heavily depends on the use of the `Bitmap` class instead of the `ContentManager`. The library contains a `GameWindow` abstract class that allows overriding functions `OnUpdateFrame` and `OnRenderFrame` that are called given times per second.

The documentation is not as complete as the one for XNA and using directly the OpenGL documentation would be problematic as the functions are not very similar.

MonoGame

The MonoGame [23] is a library that under the hood is using OpenTK and SharpDX depending on whether the current platform supports DirectX or OpenGL. As such, the library can be used to develop for Windows, Linux, Android and other platforms. The library is part of the Mono project which is an open source implementation of the Microsoft's .NET Framework.

The idea behind is to fully implement the XNA API, so the functions are indistinguishable from the original XNA ones and therefore does not offer any new functionality for 2D development.

Conclusion on graphics libraries

We dismissed the XNA because it is no longer supported. We also dismissed the Unity3D, because it is a 3D engine and at the time of the beginning of this project it did not have implicit free 2D support. And the problem with OpenTK is the lack of simpler way of drawing textures.

Therefore the decision is between SharpDX Toolkit and MonoGame. Considering we will be making this game only for the Windows platform, then they are for our purposes equally useful. And when we look at the functions of those libraries, they are similar enough that migrating from one to another of these libraries should be relatively easy in case of need. Therefore we selected with SharpDX's Toolkit.

2.2.2 Extendability

Another important part of our project is the extendability of the game content. The goal (G9) requires a way how to load and store the data about units, buildings, resources and spells. This way we can separate the code part from the content part of the game, which simplifies the editing of the properties of game elements as we do not have to recompile the entire game. It also allows modification of the game after the development cycle is completed and the users may add or remove game content without knowing the source code of the game itself.

When we talk about extendability of the game, there are three types of content and each will require different handling, storing and loading. These are the types:

(T1) Graphical data, mostly in form of textures that will represent given game entity (unit, building, resource or spell effect) on the screen.

(T2) Definition data, which contain information about game entities, like displayed name, maximum health, etc.

(T3) Script data, that will store the description of a spell from which the game will be able to execute this spell.

As was stated by the goal (G9), all graphical data (T1) should loadable from outside the game (as opposite to embedding resources directly in the executable), preferably from a widely used file format, so that the less experienced users can modify the game as well. Luckily, the Texture2D allows for loading texture externally (that is without the pre-compilling and loading via `ContentManager`) and supports for PNG format, which will be sufficient for our needs.

The definition data (T2) store the basic information about buildings, units and resources which were described in the gameplay-related problems. Using databases directly seem to be excessive for such small amount of data, therefore we will look at the alternative ways of storing information on disk that will be loaded into memory at the start of the game.

There are many file formats for holding data. Examples may be XML, JSON or CSV. Considering our intent is to load the files into the list of objects that will hold these information at the start of the program, we do not need any sophisticated features these

formats may have beyond simple iteration through elements. Our only condition is to have format, that allows us simply edit the values without knowing the loading process of these files, which means that we need a format that would allow us to specify value of given property as opposed to just having a set of values. As we can see on figure 10 on example of a unit, the XML file may take more space than the CSV file, but is more descriptive and allows to immediately see, what properties the unit have. Even more, the XML allows us to skip certain parameters that have to be present in the CSV file.

```
<Units xmlns:xsi="http://www.w3.org/2001/XML
  <Unit>
    <Id>shamanUnit</Id>
    <Name>Shaman</Name>
    <Health>150</Health>
    <UnitGroup>0</UnitGroup>
    <Texture>unit_square</Texture>
    <Shaman>true</Shaman>
    <GivenOnStart>1</GivenOnStart>
    <AttackRange>100</AttackRange>
    <AttackSpeed>30</AttackSpeed>
    <AttackAmount>5</AttackAmount>
  </Unit>
</Units>
```

```
shamanUnit,Shaman,150,0,unit_square,
1,1,100,30,5,0,0,0,0,0,0,0,0
```

Figure 10: On the left we see example of XML file, on the right CSV file.

Also, we would like to use the existing .NET features for this task so we do not have to concern us with the reliability and the documentation of 3rd party libraries. For these reasons we picked XML, as the structure of the file is reasonable for our usage and the `System.Linq.XML` already allows parsing of these files. Additionally, the XML allows us to define schema file, that will tell the user (or the proper XML editor) how the definition of the game object should look like, which would help the modification of these files

The last (but not the least) type of extendable content are the script data (T3). The original idea is that we need is a description of the spell, that would interact with the game objects and the map. And since describing such process in the XML files would be challenging at least, we decided we will use a script to define the behavior of the spell.

As was the case of the previous extendable content, there are many variants of scripting languages that can be embedded in our program to pick from. Examples might be Python or Lua. However, we do not have any important requirements for the language itself, so we might first see if the abilities of the .NET Framework itself does not have any form of scripting language, which would allow us to stay with the current libraries. Also, using C# code would allow us to only expose interfaces which would be implemented in the game objects. This is unlikely to work with other languages as the calling conventions would be different and the exposed functions would require additional handling.

First idea would be to let the users to compile their C# code directly into DLL library with provided interfaces and load them from the game. However, that would defeat one of the purposes of why we are making this game extendable – allowing the users to edit the game, without having to compile the code themselves and without forcing them to acquire a C# compiler. Therefore this is not what we are looking for.

However, the .NET Framework allows us to compile C# code at runtime via the `Microsoft.CSharp.CSharpCodeProvider` class. This allows us to directly compile code from a plain text C# source file, which perfectly suits our needs.

2.2.3 Network communication

Another component of our game to look at is the network communication. As our game is a multiplayer-only game, the networking is one of the key components. Here we discuss some of the issues with the communication.

Communication flow

There are two ways how the communication can be directed:

- Peer to peer communication
- Client server communication

Let us take a look at the peer-to-peer variant first. In this case the clients are sharing data between themselves and are on the same level in a sense, that there is no arbiter who would decide the results of the actions. The positive of this is, that any player can drop from the game and the game can continue without them. However, the problematic part is, that as there is no arbiter that would clearly state what happened and what did not, the peer to peer variant might not be a good choice for us, as there might be ambiguous states of game objects which would lead to some problems.

We might consider a code that would share the states between clients and decide which one shall be used based on consensus (between clients, not players) or by rolling a random numbers, but relying on randomization might not be the best way to go as the lack of determinism might be problematic in case of various delays between users resulting in unfair decisions.

The second approach, is the client-server communication. The server is locally running the game and updating it. The clients send the data about players decisions to the server and the server will apply the data on its game and inform all other clients about actions happening in this server's game. This will ensure, that all players are looking at the same game objects and map without any ambiguity, since all the commands are run at server locally, which prevents these ambiguous events to happen. The downside is that the game requires a server, which would mean, that if the server gets disconnected, the game will be disrupted entirely.

We will pick the client-server communication, as the disparity between client game-states is more problematic than the server getting disconnected.

Data packets

Next thing in network communication to consider is the content of the sent and received packets. The data from the game have to be transformed into linear stream, sent and then on the other side transformed back into game data. This process of transformation is called serialization and deserialization.

Let us first look at the serialization that .NET Framework provides. We may set the object to implement `ISerializable` interface, which will allow to use the serialization methods on the instances of this class. All variables with the `[Serializable()]` attribute will be included in the process of serialization, if the variable is value-based we do not need

to take any extra steps, in case of objects we have to make sure these objects implement this `ISerializable` interface as well. However, this way we would always have to serialize the entire object, which is not always needed, and especially in case of map or player update would transmit too much data while only fraction is needed.

Therefore, we should implemented our own methods, that will serialize content according to our needs. When the serialization is required, our serialization method will determine from the function argument which data are related to the request and will store them in string format along with the context-defining argument and will return the string. Vice-versa, when the deserialization method is called on a string, it will determine from the string the context and then proceeds to restore the information contained within the string.

In case of game objects like units, we need to determine which unit is in question. For this reason every game object shall have a unique ID that is passed every time data are serialized for this object and when the data are being deserialized, this ID allows to determine which object should apply these changes.

3 Programmer documentation

In this chapter we will describe the main parts of the program and the relations between them. We will go through the important classes and will comment on the functions and how they interact with other classes of this game.

We used the Microsoft Visual Studio 2013 to develop and compile this game with the addition of the SharpDX libraries. The solution (as provided in the attachment [A]) contains three projects: the Wildmen game project with the game itself, the WildmenExposedData with the interfaces exposed to the scripts and an installer for the game

3.1 Structure of the solution

First, let us take a look at the structure of the solution, what namespaces is the solution composed of and what are the relations between individual namespaces.

3.1.1 Namespaces

The entire project is divided into 7 namespaces which contain logically similar classes.

Namespace Wildmen

The root namespace for this project contains only the main classes that aren't specific enough to fit any other namespace. One of them is the `Program` class which contains the entry function of the application. Then there is the `GameWindow`, which is a descendant of the SharpDX's `Toolkit.Game` class that handles calling of the `Update` and `Draw` functions. This namespace also contains the `UI` class that contains code for checking user input, handles loading and storing textures, fonts and the functions for drawing them.

Namespace Wildmen.Game

This namespace contains objects that are directly related to the gameplay, which means classes that handle map, tiles, objects that are on the map, class that represent player and finally a class `WildmenGame` that contains the instances of these classes and represent the game as a whole.

Namespace Wildmen.Database

The `Wildmen.Database` namespace contains classes that load and store information about different types of units, buildings, resources and spells.

Namespace Wildmen.Networking

This namespace contains classes related to the network communication.

Namespace Wildmen.Screens

This namespace contains individual game screens representing different application states, and the `ScreenManager` that keeps track of the active screen.

Namespace Wildmen.Controls

This namespace contains user controls that are used in screens for interaction with user.

Namespace WildmenExposedData

This namespace contains enumerables, interfaces and abstract classes for the spell scripts.

3.1.2 Initialization process

On the figure 11 we can see the initialization process of the game. When the game starts (that is when the `GameWindow` class is instantiated and started with the method `Run()`), the method `Initialize` is invoked, which will then call the initialization methods on the instance of `ScreenManager` class and on the singletons of `Db` and `UI` class.

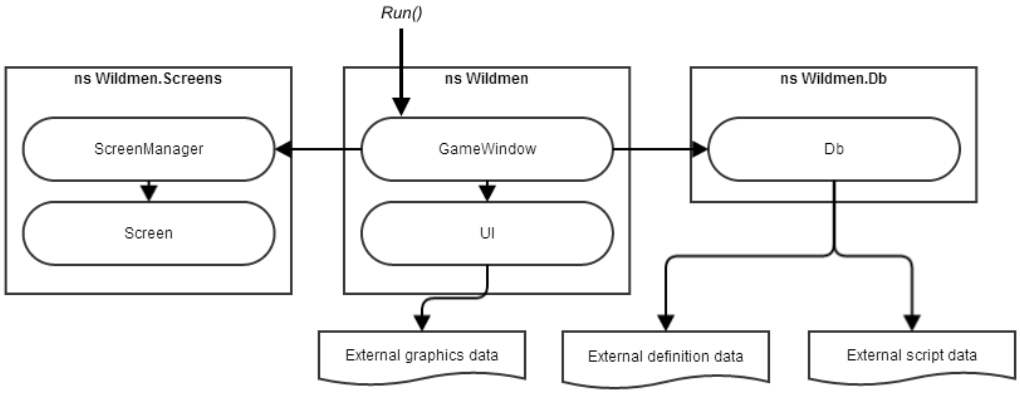


Figure 11: Initialization of the game

The initialization of the `Db` singleton is responsible for scanning the /Data folder for XML files, which contain the definition data (as mentioned in chapter 2.2.2, type T2) and depending on the content of the file calling parsing method of proper class that will load the contents of the XML file into dictionary. The initialization method of `Db` will also look for any CS files, which are C# source files (chapter 2.2.2, T3) and tries to compile and load them.

The `UI` singleton initialization is split into two parts, the first one will initialize variables related to input control (`KeyboardManager`, `MouseManager`) and event handlers for the window. The second part will load the graphical data (chapter 2.2.2, T1) from the /gfx folder to `Dictionary<string, Texture2D>` and will store the fonts from the /gfx/fonts in `Dictionary<string, SpriteFont>`.

And finally, the initialization of the `ScreenManager` will populate the list of available game screens and initialize all of them, which will load all the user controls for given screen.

3.1.3 Main loop

At the core of the game there is the `GameWindow` class in the `Wildmen` namespace that contains the functions `Update` and `Draw`, that are periodically invoked (as mentioned in the chapter 2.2.1). In the figure 12 we see an illustration of which classes have their update (light gray line) and draw (dark gray line) functions called. When `Update` or `Draw` function is called, the `GameWindow` will request corresponding action from the `ScreenManager` which, depending on currently active scene, will update or draw the `Screen` that is responsible for representation of current state of the game. In the case that the game itself (instance of the

`WildmenGame` class) is running (which is the game state illustrated in the figure 12) the next step depends whether or not is the application serving as client and/or server.

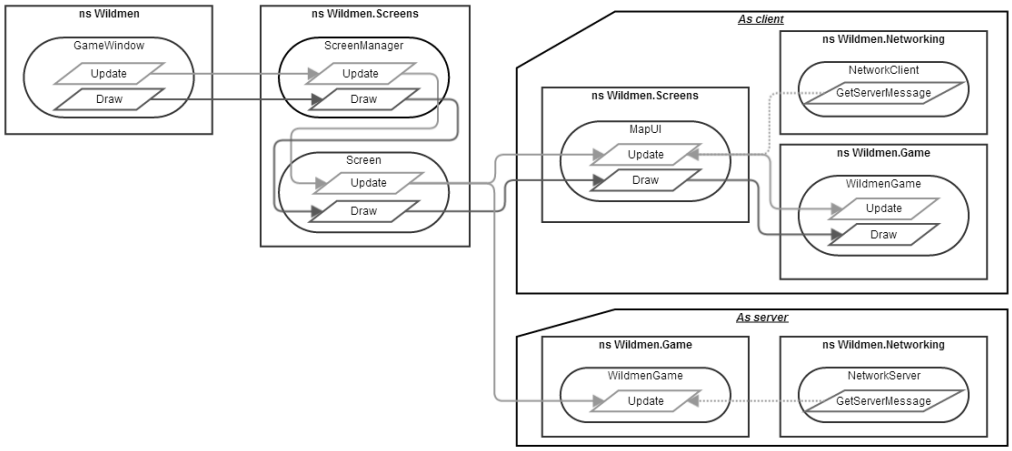


Figure 12: Illustration of calling hierarchy in running game

In case of application serving as a client, there is a `MapUI` class that handles the user's interaction with the game, their keyboard and mouse input and then from this class the `WildmenGame` is updated or drawn. The `WildmenGame` takes and applies the network information about the game from the `NetworkClient` class.

In the second case, when the application is being a server, the game does not require any mouse or keyboard interaction and can be updated directly by calling the `Update` method on the server's instance of the `WildmenGame` class. The network messages from clients are taken by the `GetServerMessages` function from the `NetworkServer` class. As the server does not require direct user interaction and all commands are taken from clients over the network interface, there is no need to draw the current game on the screen and the `Draw` function is not called.

3.1.4 Structure of the WildmenGame class

The `WildmenGame` class holds all the information about the game itself as illustrated on the figure 13, so if the game is being saved it is this class that is processed and serialized. The `WildmenGame` class contains list of all players (`List<Player> Players`), a special player that represents neutral game objects (`Player NaturePlayer`), an instance of `MapBoard` class that holds the information about map (`MapBoard Map`) and then a list of active game effects and spells (`List<GameEffects> Effects`).

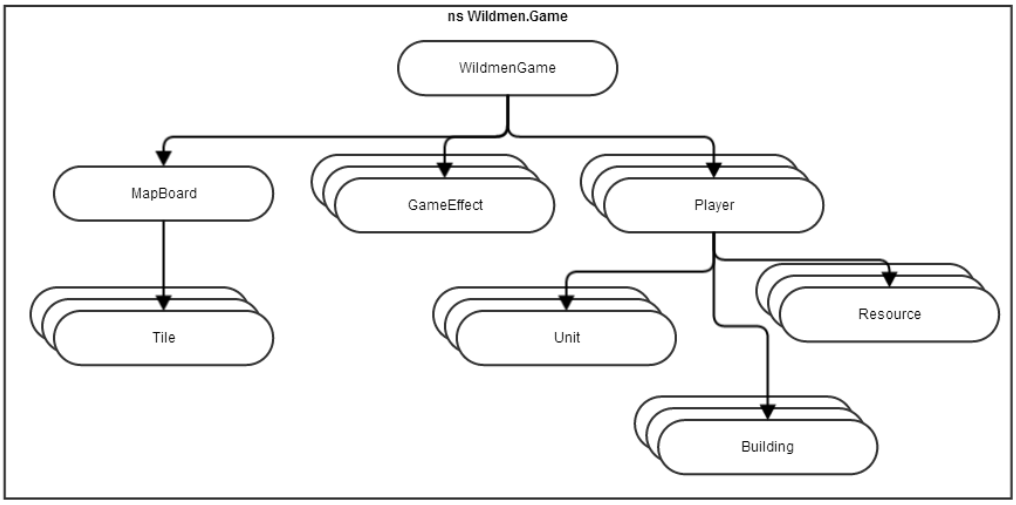


Figure 13: Classes stored in the WildmenGame class

The players are represented by the class Player and the player's units and buildings are stored in `List<Unit> Units` and `List<Building> Buildings` in this class. Lastly, this class also contains list of resources (`List<Resource>`).

The `MapBoard` class holds the data about the map in form of a 2D array of tiles.

3.2 Class WildmenGame

As we mentioned earlier, this class contains all information about the game. The class will call the update and draw functions for all the game objects present in the game, as well as the map and the game effects.

There are two public functions for drawing the game. The first one, `Draw`, will draw the regular content of the map – which are the individual tiles and the game objects. The second one, `DrawOverlay`, will draw additional graphical elements of the game objects and game effects, for example selection effect of the game object or a floating text.

There are two delegates that handle network communication, one is for sending data (`Action<string> SendData`) and the second one is for receiving data (`Func<string[]> ReceiveData`). Both of them are set when the game is being created in the `CreateGameScreen` class along with the flag (`bool IsServer`) which determine whether game should behave like server or client. The difference is that as a client the game objects are relaying the commands they receive from the user to the server and will not execute any action, except for movement. Once server receive these orders from client, it will pass these orders to its own instance of the game and send the players the result of the action (for example gathered resource or appearance of a new building). If the order cannot be completed, the server will only send updated information about the unit. This solves any ambiguous events that might occur, which were mentioned in the chapter 2.2.3. We will talk more about handling network communication in the later chapter dedicated to the networking.

3.2.1 Class MapBoard

The `MapBoard` class contains all information about the map and the constants restricting and defining the map, like what ranges the map can be, what is the maximal elevation of a tile or at which height is the sea level.

Since the map tiles can have various height and are viewed from a 2.5D view, there is not a simple way of converting point on the screen under the cursor to a tile coordinates. For this reason we have the `Tile GetTile(..)` function, that will first find tile that would be under the cursor if the map was had no elevations. Then, by calling the function `Stack<Tile> GetStackTilesBelow`, we check all the tiles that are downwards from the tile that could by the potential tile (we can determine how many tiles we have to check, because we know the maximal elevation of a tile). Now we only have to get first tile that match the position of the cursor by checking the position of the surface of the tile.

The `MapBoard` contains nested class `MapGenerator` that uses `PerlinNoise2D` random number generator (adapted from pseudocode at [32]) to generate terrain (as described in chapter 2.1.2) and is used to assign elevation valued to the Tile classes in the map array. This nested class also determines the starting positions for the players.

Additionally, as the individual tiles are drawn as an column of blocks, in order to improve the performance we are only drawing few blocks of the tile. However, the tile itself does not know how many blocks it should draw, because it is determined by the tiles in front of the tile (on the screen it is downwards from the tile), therefore we have `GetTerrainBarHeight` function in the `MapBoard` class, that will calculate how many blocks have to be drawn to ensure the graphical compactness and avoid tiles “floating” in the space (which would happen if a tile would have high elevation while all tiles around would have low elevation).

3.2.2 Class Player

The players are represented by the `Player` class, which contains the list of units, buildings and resources. It also holds the player's name, color, available resources, population limits and fog of war data.

The fog of war is implemented as an array of bytes of the same size as the map, where the value 0 represents unknown terrain, 1 means explored but not visible tile and 2 means the tile is visible. When update is called, all values 2 are replaced with 1 and for all units and buildings of this player, values in the array in given radius around the position of the object are set to 2 using the midpoint algorithm for determining the edges of the area.

The `UpdateUnitCounts` function goes through all the player's buildings and units and checks how many units the buildings can support and how many units the player actually has or is currently producing. This way we can tell whether in the `UpdateSpawners` function a new unit should be produced (as mentioned in goal (G5)). The building, before it can spawn a unit, has to be constructed and is not already spawning another unit. Additionally, in order to involve the new buildings in unit production, they are given priority when picking where the production will be started. Therefore, the building selection is split into two phases – first pass will only pick buildings that have not spawned given number of units and the second pass then picks first available building for spawning of a new unit.

The function `ShamanAvailable` attempts to find whether shaman is alive or whether it can be revived (as per victory conditions mentioned in chapter 2.1.5). If the shaman is not alive, it will attempt to find a building, that can produce shaman and is constructed. If there is no such building, the function returns false.

3.3 Class GameObject

The `GameObject` class is an abstract class containing common members of the entities on the map and is their ancestor. These entities are units (`Unit` class), buildings (`Building` class) and resources (`Resource` class).

Initialization, the vector-bound objects and tile-bound classes

The `GameObject` class contain the main properties for the entity, like health, position on the map, nearest tile, current elevation of the entity, owner, variables for graphical corrections (`Size` and `AlignmentOffset`). Additionally, there is member `Entry`, which is pointing to the reference for this entity and defines what kind of building or unit this entity will be.

Because we have two types of game objects – ones that are bound to the tile they are occupying (buildings and resources) and ones that are free to move around the map (units), there are two intermediate abstract classes, the `GameObjectTileBound` and `GameObjectVectorBound` abstract classes. These classes only contain declaration of the `Initialize` method and differ in the last parameter where the first one accepts a `Tile` and the second one accepts a two dimensional vector representing position on the map. In addition, we have to provide the `EntryDb` parameter, that defines the subtype of the entity, the `WildmenGame` instance of the game this entity is in and an owner (`Player` instance) for this entity.

Members related to drawing

In the function `InitializeGraphics` we have to assign values to the `Size` and `AlignmentOffset` members as various entities may have different sizes which have to be drawn in different positions with respect to the actual position of the entity. Additionally, we load textures that are not primary to the entity (the texture that is defined in the XML file of the entity).

The `RecalcGameObjectPosition` function will recalculate the position on the map and will find the nearest tile. This is important, since the object position is stored either as a 2D vector on the flat representation of the terrain for easier handling of movement or a map Tile. Therefore, we need to calculate the position at which the entity will be drawn and in case of vector-positioned game objects we need to determine the nearest tile, which is what the `RecalcGameObjectPosition` is for. Since this function does not have to be ran every draw or update frame, there is the `bool RecalculatePosition` variable, that indicates when the recalculation is needed and in which case it is handled by the `Update` function.

There are two drawing functions. The first one, `Draw`, should be drawing the entity itself whereas the `DrawOverlay` should be drawing only the overlay elements for this object, like selection highlighting. The reasoning behind this is that the map is drawn in two phases. The first one is drawing the elements with respect to the Z-level. That means the draw functions have special parameter that specifies Z-level and all the objects are drawn once the

phase is over in order from lowest Z-level value to the highest value. That means that the entities have to select a Z-level (usually taken from the tile they are "on", which is calculated by the `RecalcGameObjectPosition`, as mentioned earlier), so they are drawn in the correct order. However, when drawing an overlay, it is more desirable to draw over all other elements, so that the overlay is visible regardless of the Z-level of the entity. In this case the Z-level is not used, because it would have to be the highest number and drawing two objects in the same Z-level causes graphical artifacts if they are overlapping. Therefore, we decided to split the map drawing into two phases, where the second one is drawn in deferred mode, which performs the drawing when the phase is over in the order the drawing functions were called.

Class members related to orders

The ordering of the entity is done via the `CanDoOrder`, `SetOrder` and `CancelOrder` functions. However, the actual orders are handled differently by each type of the descendants, so these virtual functions only simulates the behavior of an entity that does not accept any order and the code is overridden in the classes that can accept orders.

The `SetOrder` function that checks whether the order can be performed and sets the variables to proper values. Since direct user input is required to give unit an order, this method should be only called at client's game. When setting orders we will modify several variables. The `float OrderRange` specify how close we have to be to perform the order, for example the spells can be cast from range, but melee attacks has to be performed at close quarters. The `Vector2 OrderPosition` represents the vector where the the order should be performed, which is used when we move to given position or cast spell on some tile. The `GameObject OrderTarget` is similar to `OrderPosition` but refers to an another game entity that should be targeted, example attacking another unit. The `int OrderTimeout` ensures that there is a delay between individual actions meaning the game object is not performing order every update frame and for example killing enemy unit only several update frames. And the last is the `EntryDb OrderEntry` that allows us to specify (in case of constructing a new building or casting a spell), which type of building or spell we are about to construct or perform.

3.3.1 Class Unit

The Unit class is a representation of a unit. When they are initialized, the unit type entry is copied from the original entry in the `Db` class, because we allow to change the unit properties, but do not want to affect all units, unlike other game objects which can not have instance specific building types. Another difference between this class and other game entities descending from the `GameObject` class is that units can move freely on the map.

Receiving orders

Let us take a look how the orders are received. First, the `CanDoOrder` reports whether unit can perform the order. When giving out orders the basic checks are made. If those checks are passed, then the variables mentioned above are set. The `State` variable is set to `MovingToOrder`, because the distance checking to the target will be handled in the `Update` function automatically switching the `State` variable when the unit is in range.

Executing orders

If the order is set (the `State` variable has value `MovingToOrder` or `DoOrder`), the `Update` function will execute the order. However, before doing so, it will perform certain

checks as illustrated on the figure 14. If the order is to gather resources, it will check whether the unit should go towards the resource or return to the resource gathering center. The conditions are reaching the carry capacity of the unit, the depletion of the resource and the mismatching types of the carried resource and targeted resource – if any of these condition is matched, the unit will return to the resource center to unload the cargo.

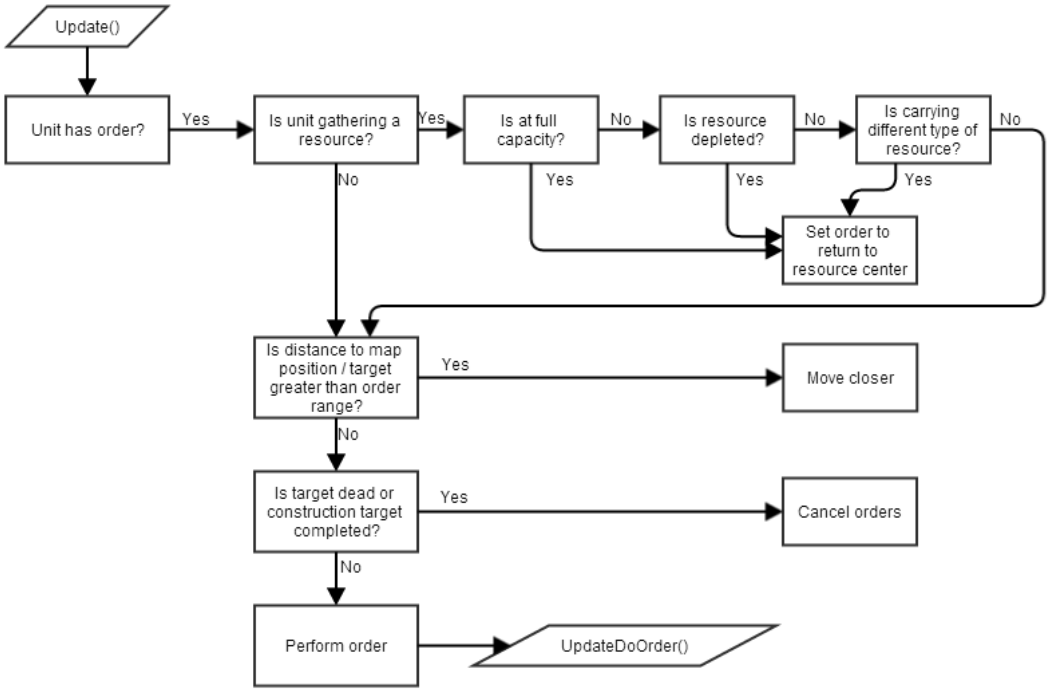


Figure 14: Changing order depending on situation

If the unit is not in range of the set position or target entity, it will change set the `State` to `MoveToOrder`, so that the unit will move towards the location. If the target of the order is dead or in case of build order the target construction is constructed, the order is canceled.

Finally, if none of these conditions are met, the `State` is set to `DoOrder` and the order is performed in the `UpdateDoOrder()` function. Then, regardless of the matched conditions, the `Update` function checks whether the `State` is set to `MoveToOrder`, in which case the `UpdateMoveToOrder()` function is called.

Orders

The `UpdateDoOrder` will perform order set in the `Order` variable with the parameters we mentioned earlier in this chapter. Now, let us look what orders a unit can have and how they are are executed.

- `Idle` order, which is not performing any action and only makes sure the unit is standing in sufficient distance from others
- `Attack` order, which makes unit deal damage to other objects.
- `Construct` order, which makes unit either initiate construction or contribute to building already initiated construction.

- `Spell` order, which makes unit initiate a spell.
- `Gather` order, which makes unit gathering resource and walk between resource and resource center.
- `Train` order, which makes the unit enter building that can train this building.

Idle order

Idle order means, that no action needs to be performed, therefore, this is result of simple move order which has now completed. When ordering the unit to move, we are not checking whether the unit will step on the same place as other unit as there is likely possibility, that the other unit will already be gone by the time we get there. However, it is possible, that when the unit arrives to given place, another unit will wander around to the same place. In that case we would have two or more units on the same place, which would bring an issue that we can not simply tell how many units are in that position, which would be problematic. For this reason, when unit finishes its move order, we check if there isn't any unit in close proximity (`float MIN_IDLE_DISTANCE` constant). If so, we order the unit that just finished moving to move slightly aside (`float DODGE_DISTANCE` constant).

Attack order

When the unit it attacking and is in range, the client will only create a combat animation, nothing else. The actual damage calculation is done by server (for reasons we described in the chapter 2.2.3) which will then report the results to the players.

Now, as for the interaction between units let us have a situation where attacker unit A attacked unit D (as defender). Unit A, as every other combat unit, has a strength attribute (stored in the `UnitDb` instance for the unit type). Moreover, the unit A can also have a bonus or a disadvantage against the D's unit type (as referred to in the goal (G5)), therefore it is required to adjust the damage value the A unit will deal to the D. Once the value is calculated the method `D.ReceiveDamage(int value)` is called, which will subtract the damage from the D's health pool and send the message about received damage or death (if the health pool drops below 0).

Construct order

This order, just like the attack order, will be executed at the server side of the game and clients will only be informed about the results. This order depends on whether the order was set with target building or not. If the target building was set, than it means that the unit will contribute to the construction process of the building (since as mentioned in goal (G4) and chapter 2.1.4, the buildings, in addition to being placed, will also have to be constructed). The progress is handled in similar fashion as the damage is handled when being attacked – the target building has function `ConstructionProgress` that will update the progress and perform additional actions once the building is completed. The amount the unit contributes to the construction is defined in the `UnitDb` entry of the unit.

In the second case, when the `Construct` order is set without the target building, a new building will be placed at given position. However, first it has to be checked whether the building can be placed at this position. We have to check whether there is any tile-bound game entities at this position and whether the terrain is flat enough for a construction. Additionally, the player must meet the conditions mentioned in the building type definition (in the `BuildingDb` class for this building), such as the restriction of one building per player,

requiring another building to be built first and the resource cost of the building. If all these conditions are passed, the building is created and initialized, the resource cost is deduced from player's stockpile (in the `Player` class instance) and the unit is automatically set to start construction of this building.

Spell order

This order will only check whether the spell is unlocked (required building is built, as mentioned in the goal (G7)) and then, depending on the type of spell, will be created an instance of a `GameEffect` class in the server's instance of the game, that will receive either tile or game entity as target, depending on the spell entry (`SpellDb` class).

Gather order

The `Gather` order will perform action depending whether the unit has just returned to the resource gathering center (a building which `BuildingDb` entry has the `ResourceCenter` set on) or whether the unit is by the resource. In the first case the unit will transfer all the resource it is carrying to the owner's resource stockpile and then will head back towards the resource after a short delay. In case the resource source is depleted, the unit will cancel its orders. The second option, when the unit is by the resource, will first check whether the resource has an active cooldown (that prevents multiple units to gather the resource at the same time, as mentioned in the goal (G3)). The amount unit can get from the resource is specified in the `UnitDb` entry of the unit, which is passed as parameter in the function `Take` of the resource instance. The `Take` function then compares the requested amount with the available amount and returns smaller of both number and then deduce this number from the available amount, which will ensure that the unit can not take more than available amount from the resource. Again, this order is performed on the server side.

Train order

This order will first check whether building can train this unit now and is not training other unit, in which case the unit will attempt again after certain cooldown (`int TRAIN_QUEUE_CHECK_TIMEOUT`). Then the player resources are check whether they contain enough resources for the task and if so then the spawning process is started, resources deduced from the player and the unit submitted for training is killed (because the process will spawn new unit). This command is ran at the server's instance of the game.

Movement

As we mentioned earlier, the movement is done when the `State` variable is set to `MovingToOrder` and is carried out at the end of the unit's `Update` function. The movement is simple. As long as the tile in the direction of the movement is on the map, doe nots have elevation difference larger than given threshold (`float MapBoard.CLIFF_THRESHOLD`) and is above the sea level, the unit will go forward. If the path forward is unavailable, the unit will try to go in either orthogonal direction with the same restriction as with the forward path. If none succeeds, the movement is canceled along with the entire order.

3.3.2 Class Building

The `Building` class represents an unmovable structure on the map. This `GameObject` descendant, unlike others, has a construction progress, splitting the buildings into the not-constructed and the constructed ones. Also, buildings can produce units.

The building is placed across multiple tiles, therefore after initialization they all have to be claimed by the building by the function `ClaimTiles`. The reason is that when the building is being drawn, then we have to draw it in certain Z-level (as we mentioned in the chapter 3.3), but if the tiles would draw themselves at their default Z-level, they would cover the building, and if we would adjust the level of the building, it would cover surrounding tiles it should not cover. This is avoided by assigning the building to the tiles, so they can determine how they should be drawn. Additionally, when the building is placed, all the tiles below are flatted into same height, so that the building is not floating on one side or is not borrowed into the ground.

As we mentioned earlier, the building has two stages. The first one is the construction stage. In this state the building is being constructed by units capable of constructing and the building can not perform any task nor can be used to unlock spells or other buildings. Once the building is completed (the construction progress reaches the value in `ConstructionTime` member of the `BuildingDb` instance), the `Completed` function is called and the building can be used.

The second mentioned specific ability of the buildings is the ability to produce units. This is done via the function `StartUnitSpawner` which will create a `SpawnerEffect` (which is descendant of `GameEffect`) which will, after given amount of updates (the `speed` parameter), spawn a new unit of given type. Additionally, we can set a new spawner directly with the function `ResetSpawner`, but this function is only used when the spawner is disposing or when the spawner is created by deserialization. We will talk more about this `SpawnerEffect` class in later chapter.

3.3.3 Class Resource

Finally, the last descendant of the `GameObject` is the `Resource` class. This class is representing a resource and keeps information about amount of the resource this instance has available for gathering and cooldown before the resource can be gathered again. Both these values are defined in the `ResourceDb` class for given resource type and the values are updated when the `Take` function is called.

3.4 Class GameEffect

The `GameEffect` class is the class that represents the spells, the animations and the spawning processes. The spells are events handled directly by the `GameEffect` class and are defined by their description and the script, which are stored in the `EffectDb` class. The animations using this class are the unit death animation (`UnitDeathEffect` class) where unit's color slowly bleach into white, the scrolling text (`ScrollTextEffect` class) and an animation from a texture (`AnimationEffect` class), which used for example when two units fight. And finally, there is the spawning process (`SpawnerEffect` class). All these are classes descending from the `GameEffect` class.

As we mentioned, the `GameEffect` class is just a skeleton for an effect (or a spell) that is extended either by overriding the methods or by the delegates stored in the `EffectDb` entry. Additionally, all spells have to set a tile in the initialization and only the players, that can see this tile (their fog of war value for this tile is set to 2) will be able to observe the graphical animations - the effects, that have set the `Graphical` or `GraphicalOverlay` flags. The

difference between these two is when the draw function will be called – either in first or in second draw pass.

The `Speed` member determines how many global update frames will pass between individual calls of the `GameEffect`'s `Updates`. The `Duration` then specifies how many times the `Update` will be called before the `GameEffect` will be deactivated and disposed. The `UpdateOffset` stores the offset of the global update counter at the time of initialization of this `GameEffect`. The offset is important when determining when the `Update` should be called as the decision when to update the `GameEffect` is decided by using modulo `Speed` on the global update counter, which without offset would lead to triggering the first `Update` early.

Let us take a look first at the spells made from the `EffectDb`. The spells can be called either on a tile or on a game entity (a game object), which will changed which delegate will be called and which objects can be targets of the spells. The `OnSpellStartGameEntity` is called for spells targeting game entities and `OnSpellStartTile` for spells targeting tiles. After that the code is same for both types of spell. Every time the `GameEffect` is updated the `OnSpellStep` is called and on the last `Update` call the `OnSpellFinish` is called before the spell is declared as inactive. We will talk about the scripts in later chapter dedicated to scripting.

The `UnitDeathEffect` is an effect that will take the information from the unit passed in the constructor and will create texture on the screen that will slowly bleach the color from the player's color to the transparent color.

The `ScrollTextEffect`'s constructor accepts text, position, time and vector and will display this text at given position, that will for certain amount of updates move in specified vector.

The `AnimationEffect`, similarly, will take animation, position and time and will display an animation. The animation is loaded through the `TextureTileSet2D` struct, which contains a texture with a series of images and the struct provides information which part of the image should be drawn.

Finally, the `SpawnerEffect` will create a new unit once the Step is called, provided the building still exists and the effect is running at server. While it might seem unnecessary to run this effect at the client's instances of the game, the building provides visual feedback how long it takes for the spawner to spawn a new unit and these data are taken from this `SpawnerEffect`'s data.

3.5 Class MapUI

The `MapUI` is an intermediate step between the `GameScreen`, which is one of the scenes that define individual states of the application (they will be mentioned later in their own chapter), and the `MapBoard` of the `Game`. As such, it provides the interface the user needs to control their forces and overview the situation.

Once the `MapUI` is initialized, a player has to be selected using the `AssignControllingPlayer` function. `MapUI` will then draw map this player (`Player controllingPlayer`) according to this player's fog of war and will allow to control the

game objects owned by this player. If no player is selected, the `MapUI` will draw entire map with full visibility and will not allow the player to control any game objects.

MapUI states

The `MapUI`'s behavior depends on the `InGameState` variable which defines its current mode. When the variable has the value `Default`, it means no game object is selected. The `DragSelecting` value means, that selection is in progress. The `DefaultSelected` indicates that one or more objects have been selected, but no order has been specified. In addition to these types there is an enum value for every type of order. When we want to order the game object to do something, we first select the order we want to give this unit and then apply the order on a target. We need this `InGameState` as the orders do not share the same requirements (attack can be performed only on other object, unit can move only on tiles, for constructing a new building we need to first select the building type).

Selecting order and target

When we select an order, the `InGameState` is changed by the `ChangeIngameState` function, which will change the behavior of the drawing and target selecting functions. Once we select the desired target, the function `ExecuteOrder` will pass the selected order and target to the game object through the `GameObject.SetOrder` function.

The most important changes that depend on selected order are the tile highlighting (`bool mouseoverTileTracking`) that will keep the the track of an tile under the mouse (`Tile mouseoverTile`) and game object highlighting (`bool gameObjectTracking`) which keeps track of the game entity under the cursor (`GameObject mouseoverGameObject`). Both are handled by the `UpdateTracking` function. In addition to these, when spell order or build order is selected, the selection menu is drawn, which allows to select spell or building that should be cast or built.

Additionally, if no command is explicitly selected once the unit is picked, the `SmartCursor` function will try to select the most suitable order for the tile or game object under the cursor. This will happen continuously (`bool autoCursor`), until order is explicitly changed.

Drawing and updating

The `Draw` function is split into two phases that will call the `Draw` and `DrawOverlay` functions of the `WildmenGame` class (which are described in the previous chapter 3.2). In addition to that, after the game's overlay is drawn, the `MapUI` will draw its own overlay consisting of player information (player's name and available resources), selection menu when a build or a spell order-mode is active and the user controls for selecting order.

There are two update functions. First one, `Update`, implements regular updating logic for the `MapUI`. The second one, `BackgroundUpdate`, serves for updating the game and interface when the escape menu is activated, in which case we want the user input to be ignored while having the game updated. Which function is used depends on the `GameScreen` class and on the `ScreenManager` that handle all the screens, which we will describe in later chapter.

3.6 ScreenManager and ControlManager

The screens (`Screen` abstract class) define individual phases of the application and handle updating and the content drawn on the screen. These screens and their transition is managed by the `ScreenManager` class that keeps track of the active screen. These screens make use of the user controls, which are managed by the `ControlManager` class.

Class ScreenManager

The `ScreenManager` contains a dictionary of screens with the screen's name as a key, which allows to switch between screens with just their name. The switching is done by the `SwitchScreen` method, which will first call `OnDeactivate` method on the old screen and then `OnActivate` method on the new screen. It is important to keep in mind, that after calling `SwitchScreen`, the current screen must finish their current `Update` call after having their `OnDeactivate` method called.

The subscreens are screens that are displayed "over" the current screen. The subscreen is activated by calling the `CallSubscreen` method and are closed by the `CloseSubscreen` method. When the subscreen is active, the original underlying screen is calling `BackgroundUpdate` method instead of `Update` method, allowing the screen to implement update logic with exclusion of user input code.

The `screen` abstract class contains along the abstract and virtual methods also fonts and styles for the user controls. All the methods in this class are called from the `ScreenManager` and should not be called manually. The screens should use the `ScreenManager`'s functions for interacting with other screens instead.

Class GameCreateScreen

An instance of this class represents screen where user can create and configure new game or join to existing game.

When a new game is created, an instance of `NetworkServer` is created which will handle its communication. The user then can change the size of the map and unit limit, where these information are being passed to the server. When the user starts the game, the server is told to start the game via `NetworkServer.StartGame` and if the map of given parameters can be created, the server transmits serialized form of the game to the clients, assign users their players and the game is started. The screen is switched to the instance of the `GameScreen` class.

When the application joins as client to a server, an instance of `NetworkClient` is created and connects to given IP. Once the client is connected, all the game-related messages are retrieved from the `NetworkClient` via the `GetServerMessage` method and are parsed in the `HandleClientMessages` function. There the update messages for the game are checked and once the game is transferred and player assigned, the screen is switched to the instance of the `GameScreen`.

We will describe the network communication more in later chapter.

Class GameScreen

Instance of this class represents the screen where the game is played. When the screen is activated, the game, the server, the client and the player parameters are passed from the previous `GameCreateScreen` screen. During activation of the screen, the instance of `MapUI` is

created and controlling player is assigned. Additionally, the `SendData` and `ReceiveData` delegates of the `WildmenGame` are set. If the application runs only dedicated server (the application is only running as server, not as client), the `EscapeScreen` subscreen is invoked when the screen is activated.

The messages from the server are acquired by the `NetworkClient.GetServerMessage` method, which returns game-related messages, they are then further filtered in the `HandleClientMessages` method that applies changes concerning the entire game (end of the game, restart of the entire game) or how the game is controlled (assignment of a player to user). The rest is then stored in `messageQueue`, that is then queried by the `WildmenGame.ReceiveData` delegate.

The update and draw functions calls their `MapUI` counterparts and if the application is running as a server (either as a dedicated or along the client), the update is called on the the server's instance of the game.

3.7 Scripting

The scripting mechanism is used for the game effects (as mentioned in chapter 3.4), where it defines shaman's spells, specifically, the initialization of the spell, the individual steps of the spell and the finalization of the spell. These scripts interacts with the individual game meta-objects (game entities, map tiles, players, game itself) through the exposed interfaces from the second project `WildmenExposedData` of the solution.

The scripts are handled in the `EffectDb` class. First the `LoadSpellScripts` method looks for all cs files in the /Data folder. Then it will try to compile every file using the `Microsoft.CSharp.CSharpCodeProvider`, with the WildmenExposedData.dll as a referenced assembly. If the compilation finishes without any problems, then the code is loaded into dictionary of `SpellEntry` objects using the `System.Reflection`, the XML entries then refer to these SpellEntries through the `SpellCodeId`, which is the `SpellEntry.Id` field.

Namespace WildmenExposedData

The `WildmenExposedData` project contains interfaces for game meta-objects, struct that represents a spell entry (`SpellEntry` struct), interface for retrieving spell entries (`Interface ISpellCodeProvider`) and few helper enumerations. We placed them in separate namespace in order to prevent the scripts from accessing internal information of the classes from the `Wildmen` assembly. The game meta-objects implement these `IScript-` interfaces and if they return any meta-object, it is cast into proper `IScript-` instance.

The individual spell scripts are represented by the `struct SpellEntry`, where we define the `Id` of the spell, the type of the target and the individual delegates with the code. When the target is set to `Tile` the `OnSpellStartTile` is called when the spell is being initialized, when target is `GameEntity` the `OnSpellStartTile` is called instead. When the effect updates the `OnSpellStep` is called and when the last update is called, the `OnSpellFinish` delegate is called.

The scripts can save their temporary data in the `IScriptGameEffect.LocalData`.

3.8 Networking

The `Wildmen.Networking` namespace is handling the network communication and is split into three classes. The Network class is a static class that contains static information about the networking, like port number and is responsible for composing the network messages for which the `INetworkSerializable` interface is used.

The `NetworkServer` class is encapsulating a `TcpListener` and represents the server. When a client is connected a new instance of `ClientConnection` class is created an the network messaging with this client is handled through this class. The `NetworkClient` is the client side of the communication built around the `TcpClient` class.

Initialization of the connection between server and client

Figure 15 illustrates the network messages between client and server when the client connects to the server and in the second part when server starts the game. When client connects, the server informs is about about its id and temporary name through the `MessageType.Hello` message. Then, the client is informed about current server settings (map width, height and unit limit) in the `MessageType.ServerSettings` message and finally in the `MessageType.UserList` the client is informed about already connected users. The client responds with a nick change request to new nick (`MessageType.NickChange`) which will either get approved and the server will respond with `MessageType.NickChange` message or gets denied, in which case the response is `MessageType.Failed`.

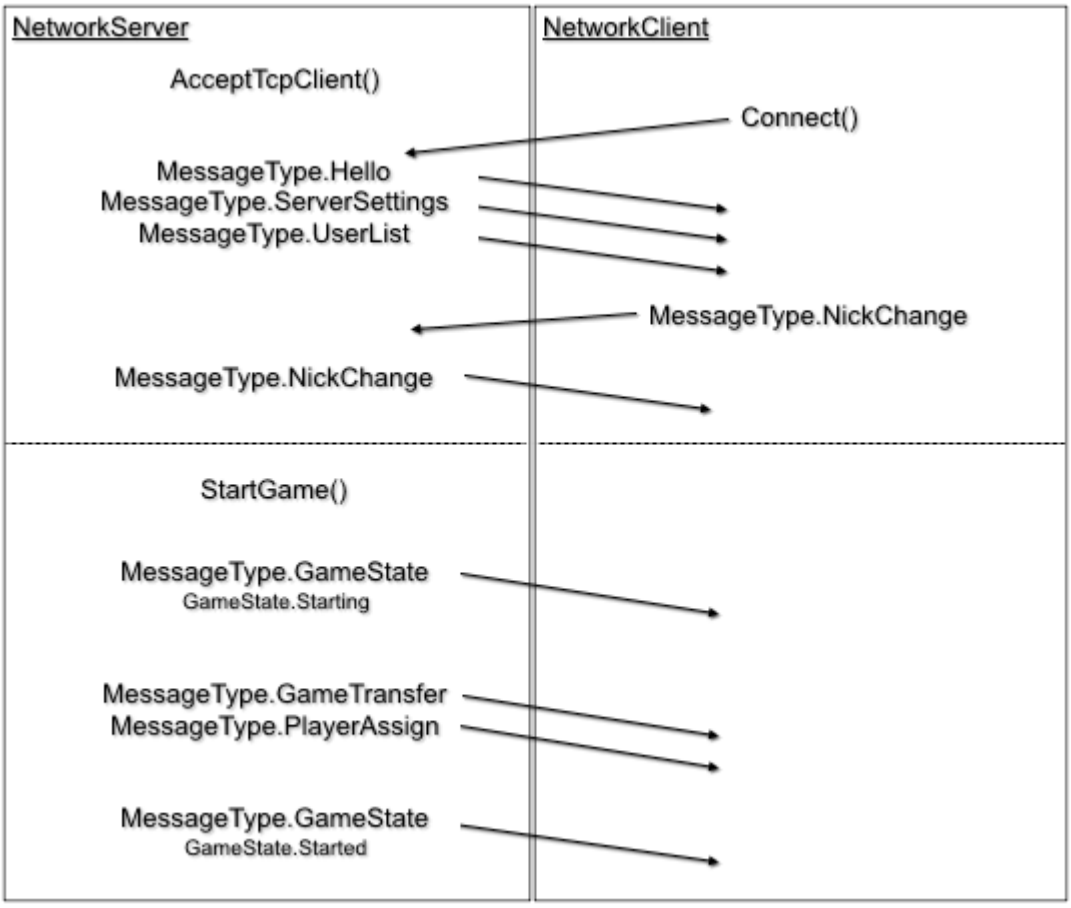


Figure 15: Diagram of network communication when client connects and when the game starts

When the game is starting, first the game state is changed to `GameState.Starting`, which will disable any changes in game settings and then tries to generate a map with current settings. If successful, the serialized game is sent through the `MessageType.GameTransfer` followed by `MessageType.PlayerAssign`, which will assign the individual clients their players. Once all users receive the map, the game state is switched to `GameState.Started`. If the map generation fails, the game state falls back to `GameState.Lobby` and the players are informed that map could not be created with given parameters.

Examples of handling the network communication

On the figure 16 we can see an example of how network message for unit update is made and sent. First, a message is made using the `Network.MakeClientMessage` method, which as a first parameter accepts type of a message, followed by a list of arguments. The `MakeClientMessage` will decide according to the message type what to do with the arguments. In this case, the `GameObjUpdate` message means the first argument is a game object and we want to inform all clients about its current state. Therefore, a `Serialize` method (as declared by the `INetworkSerializable` interface) will transform the unit data into text stream. Once the message is composed, it is passed to the `WildmenGame.SendData` function, which (when the game is in the server mode) pass the message to `NetworkServer` that will redistribute it among the `ClientConnection` classes representing individual clients.

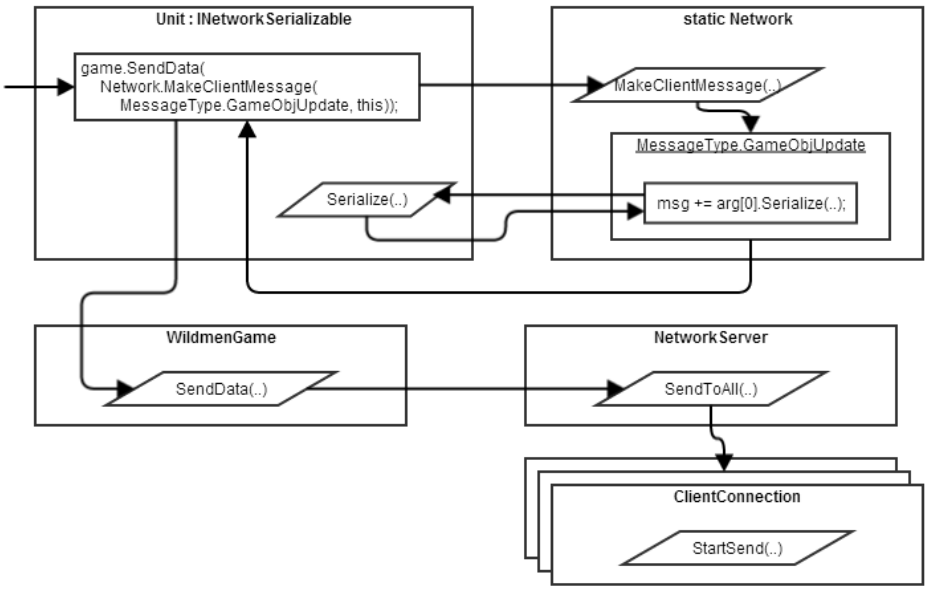


Figure 16: Example of a process for sending unit update (at the server side)

On the figure 17 we can see the process of applying unit update from the network message. First the `NetworkClient` will asynchronously receive message from server. It will check whether it concerns the connection itself (like disconnection or user renaming), if not it will store the message in the `Queue serverMessages`. When the `GameScreen` is updated in the regular update frame, it will check for any new messages in the `NetworkClient`'s `serverMessages` queue. It will check if the message doesn't change the game itself or the way how the game is controlled (restart of the map, assignment of controlled player) and

then pass the message into `Queue messageQueue`. Later, still in the same update frame `WildmenGame.Update` is called, which will invoke the `HandleNetworkMessage` function, which will check the `GameScreen`'s `messageQueue` and apply them. Our example message contains unit update, which has `MessageType.GameObjUpdate` identifier. This message type means, that following number in the message is an id of the concerned game object, which allows us to select correct entity and pass the message to the `Deserialize` method of the entity (as declared by the `INetworkSerializable` interface).

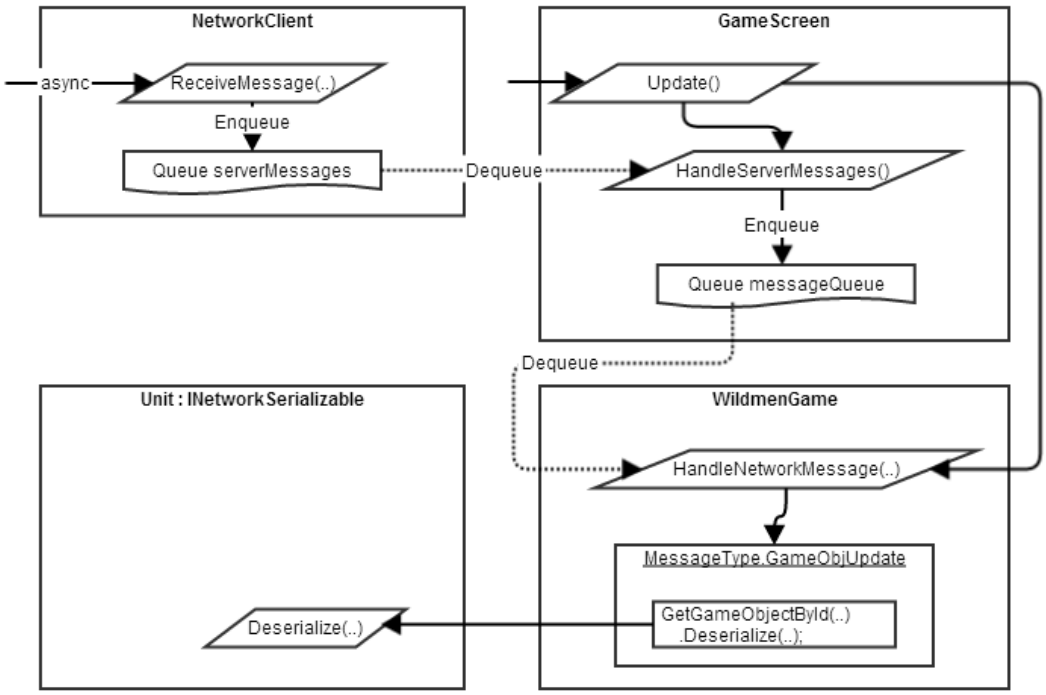


Figure 17: Example of updating unit from the network message (at client side)

Class Network

The static `Network` class, as mentioned above, contains the port number and is responsible for composing messages. Also, this class contains list of user messages where the chat messages between players are stored along with server notices.

There are two main functions in this class – `MakeServerMessage` and `MakeClientMessage`. These functions compose the network message depending on the provided message type, which define how the provided arguments should be handled (as shown on the figure 17. The reason why all network messages are made through this function and are not serialized directly is to separate the serialization and network communication and to unify how the server messages are made, since not all messages contains the `INetworkSerializable` interface instances as their parameter. There are two functions for making messages in order to differentiate between requests for message composition from the client and from the server, because the client can make only fraction of all possible messages.

INetworkSerializable interface

The `INetworkSerializable` interface represents game meta-objects, that can be changed directly by the network messages. The serialization method has a `MessageType` parameter which allows the class to limit the amount of information it will serialize and then store to the provided instance of `StringBuilder`, which is provided in the second parameter. The class should then be able to deserialize these information back from an array of strings using the same `MessageType`.

Class NetworkServer

The server class is built around the `TcpListener` class. When the server is started (via the Start method) it will create a `TcpListener` listening on all IP addresses at the `Network.PORT` port. Then it will asynchronously accept client connections. When client connects the server an instance of the `ClientConnection` class is made for this client.

The instance of `ClientConnection` class is handling the sending and receiving of messages of a single client. Messages are sent using the `ClientConnection.StartSend` method and when a message is received, the callback delegate provided in constructor is used which points to the `NetworkServer.ReceiveMessage` function. If the received message is not game-related (client requests a list of users, nick change, client sends a chat message), the server will respond to it, otherwise the message is en-queued in the `clientMessages` queue where waits until a handling function from the update thread checks the messages.

The reason why the server can not process the game-related messages directly is due to the asynchronicity of the network communication. Since we do not know if the main game thread is updating or drawing, we would risk changing object that is currently being processed, resulting in inconsistent state of the object and therefore of the entire game.

The server is periodically sending `MessageType.Ping` to all its clients expecting same reply. If the server does not receive `MessageType.Ping` back from a client before it sends another wave of pings, the client connection is closed.

As we mentioned earlier in the chapter 2.2.3, the server is running its own instance of the game which is updated from the game thread in the update frame.

The server will only accept 4 clients at a time, if 5th tries to connect, it will be immediately closed.

4 User documentation

In this chapter we will look at the user side of this application, we describe the individual game screens, the visual interface that allows to control the map and give orders to units. We will then describe the individual game objects and spells.

4.1 Installation

The game is installed starting the setup.exe as provided in the attachment [C]. The installer will also check for prerequisite and if it is missing, it will download and install it. The prerequisite is Microsoft .NET Framework 4.5 (x86 and x64). The firewall exception might be needed for the TCP/IP port 31240.

4.2 Individual game screens

The first screen the users will see when they start the application is be the main screen. The only purpose of this screen is to show simple interface that helps the users understand what they are seeing. This screen only contains two buttons, one “Exit” will exit the game, the other “Multiplayer” will take the user to the game creation screen.

The game creation screen (as pictured on the figure 18) allows to start a server or connect to one on given IP. If the application is running as server, it allows to set the parameters of the game and to start the game. And finally, if the server is running or the application is connected to a server, it is possible to chat with other players.

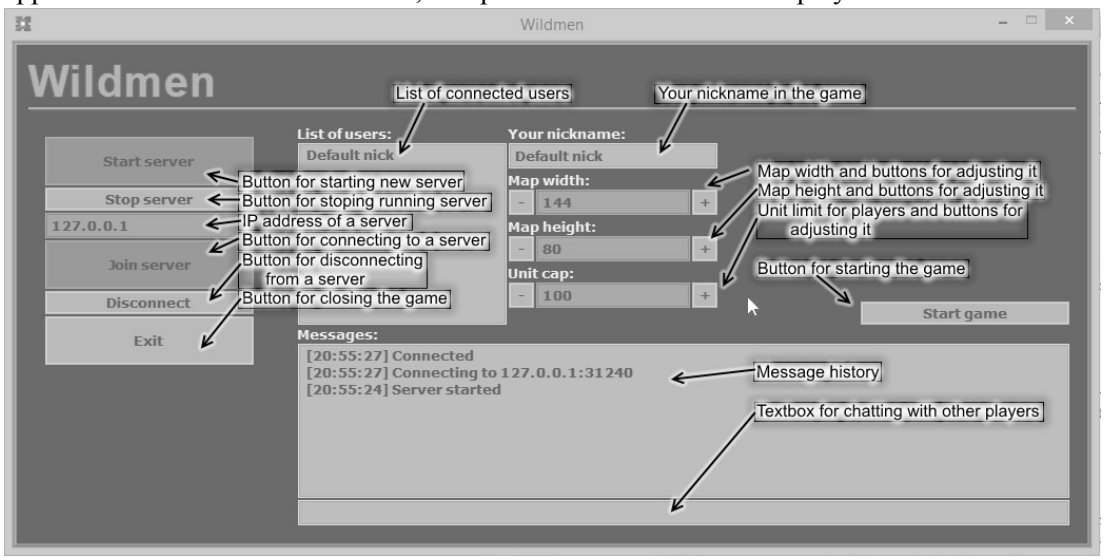


Figure 18: Screen for game creation with description of individual controls

4.2.1 MapUI

The figure 19 illustrates example of the game. On the left side there are the available commands for the selected objects, on the right side we can see the information about the player and below we can see a menu (visible when picking either a spell or a building). Below the overlay we see the map with units, buildings and resources.

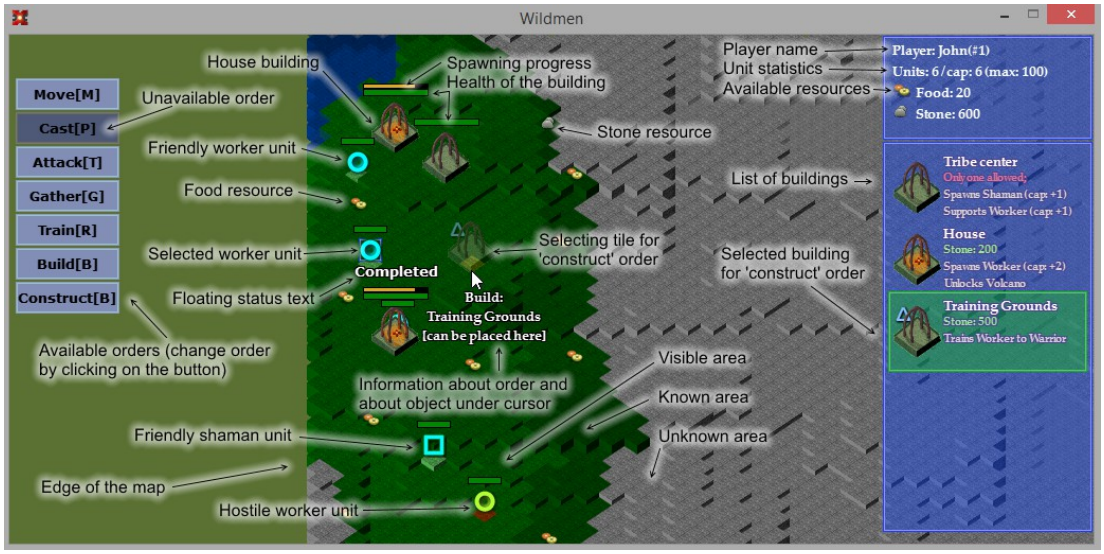


Figure 19: Interface of the game

Camera movement

The current view of the map can be moved using the WSAD keys, where W moves the camera upwards, S downwards, A leftwards and D rightwards. Additionally, pressing Q will center the screen on the shaman's position, pressing H will center on the first building that can respawn shaman and pressing C will center on the first selected object.

Revealed areas

Every unit and building reveals an area around it. As long as the area is revealed, we see any enemy building or unit in this area. Once the unit moves away, the area becomes known, which means we can see the terrain and the resources on it, but we can not see any enemy unit or building on the known tiles. The gray tiles are unknown area, and these are tiles we have not yet visited and we only know its relief.

Selecting object

In order to select a game object, we can either left click on it or drag the mouse over them, which will select multiple objects. However, if drag-selection contains at least one unit, all other types of objects (that means buildings and resources) will be excluded from the selection. If we left click on an empty space or drag over empty space, no object will be selected.

Giving orders

When a game object is selected, it can be given order. If we do not specify the order explicitly, the game will try to guess correct order from the objects under the cursor. We can specify order either by clicking on a button on the left side of the screen or by pressing proper key (the keys are mentioned in the labels of the buttons in square brackets). Only units can receive and execute orders.

The unit can receive following orders and only under certain circumstances:

- Move order, which will move the unit to target tile. A unit can move only if its speed is greater than zero.
- Attack order, which will make the unit attack target game object. A unit can attack only if its attack damage is greater than zero.
- Construction or build order, which will make the unit either initiate a new building construction of selected type at the target tile or contribute to already started target construction. A unit can build only if its construction contribution amount is greater than zero.
- Train order, which will make the unit to enter target building that can train this type of units.
- Gather order, which will make the unit gather target resource and walk between the resource and nearest resource center. A unit can gather resources only if its carry capacity is greater than zero.
- Cast or spell order, which will make the unit to cast spell of selected type at the target tile or game object. A unit can cast spells only if it is a shaman unit.

Move order

When ordering unit to move we right click at a tile on the map. The unit will go straightforward towards the selected tile, but if it encounters steep slope or water, it will stop.

Attack order

The attack order can be performed on any other game object. The attacking unit will first move to the attack range distance from the target and then will start damaging the target. For the movement to the target same restrictions applies as for the move order.

Train order

The training order can only be performed on a building that can train the selected unit. This process will consume given amount of resources and during the training process the unit will be unavailable.

Construction or build order

The construction order can either be performed on an existing building or an empty tile. In first case the unit will move towards the building and then contribute to the construction progress if the building is not constructed.

When the construction order is performed on empty tile, the unit will attempt to start a new construction of the selected building (represented by the semi-transparent ghost of the building as illustrated on the left side of the figure 20) on this tile. The building type is selected in the menu on the right side of the window by hovering over the entry with cursor or by using scroll wheel or by using the up and down arrow keys. The menu entries contain information about the building along with the resource cost and other requirements. If the player has enough resources and the requirements of selected entry are met, the entry is highlighted by green rectangle otherwise the rectangle is red.

Additionally, the tile has to fulfill certain conditions, otherwise it can not support new building, which causes the ghost of the building to be shaded to red (figure 20, right side). The tile and the three tiles above it (tiles to the top-left, top and top-right) must have low difference between their elevations, must be devoid of any other building or resource and must be visible to the owner of the building. Only then the new building can be built on the new position, the unit will move towards the target tile, will create the new building and automatically continues with construction order on this building. When the building is started, the resource cost is deduced the from the player's available resources.

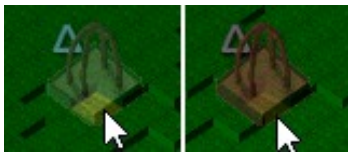


Figure 20: Building placement

Gather order

The gathering order can be performed on a non-depleted resource. The depleted resource is half saturated than non-depleted resource as illustrated on the figure 21. When unit is ordered to gather a resource, the unit will start moving between the resource and nearest resource center (Tribe center for example) until the resource is depleted. If the unit is carrying another type of resource, the unit will first return the resource to the resource center and then will start gathering the new resource.

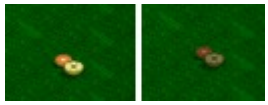


Figure 21: Resource and depleted resource

Spell or cast order

The spell can be selected from the menu on the right side of the screen and can be picked either by hovering over the entry with cursor, or using the scroll wheel or by using the up and down arrow keys. The spell is either targeting map tile or a game object depending on the spell properties. The shaman will first move towards the target and when in range will cast the spell. After spell is cast, the shaman is unable of any action for brief period of time.

4.3 Game entities and spells

In this sub-chapter we will describe individual game objects and their properties. We have total of 9 unit, 9 building types, 2 resources and 8 spells.

4.3.1 Units

The units have their color tinted to the color of the player, allowing their identification. On the figure 22 on the left we can see the worker unit texture from the file and on the right we see green player's worker unit.

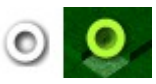


Figure 22: Image from texture file and in-game

Shaman unit

The shaman unit is slower than other units, but has more health than others. It has long-range attacks and can cast spells. If player loses this unit, it will start re-spawning in the tribe center building. The shaman does not fare well against workers and shield-bearers.

Worker unit

The worker unit is the only unit that can gather resources and build other buildings. They are the most basic unit and have fairly low health and attack. They are spawned in the house building.

Warrior unit

The warrior unit is the basic combat unit. It has high health and average short-range damage. This unit is trained from worker in the training grounds building.

Archer unit

The archer unit is a unit with excellent range, exceeded only by shaman, and they are faster than all foot units. It has attack bonus against workers, warriors and spearmen, but does not fare well against shamans and shield-bearers. This unit is obtained by training worker in archery range building.

Spearman unit

The spearman unit is a ranged unit with good amount of health. Its range is not as good as archer's, but in addition of bonus versus workers, warriors and archers, this unit is very good against any mounted unit. This unit does not fare well against shamans and shield-bearers. This unit is trained at weapon-master's camp building from warrior.

Shield-bearer unit

The shield-bearer unit is a slow melee unit with no attack modifiers, however most units do not deal much damage to it, making them good for taking out ranged unit. This unit is trained at armor-master's camp building from warrior.

Mounted scout unit

The mounted scout is the fastest unit in the game, however is is also the weakest unit (right after workers) and have disadvantage when fighting shamans or shield-bearers. This makes them good for exploring the terrain and picking out the enemy workers. This unit is trained at stables building from worker.

Mounted warrior unit

The mounted warrior is a short range mounted unit. This unit has strong attacks and has advantage against archers. This unit is trained at riding trainer's camp building from spearman unit.

Mounted archer unit

The mounted archer is a long range mounted unit. This unit has strong long-range attacks, but they are very slow. Also, this unit has low health and disadvantage when fighting shamans or shield-bearers. This unit is trained at riding trainer's camp building from archer unit.

4.3.2 Buildings

In the left column there is texture of the building when it is not yet constructed and in the middle column there is texture when the building is built. Enemy buildings are identified by faint red glow.


Stockpile building

This building serves as resource center and is given to the player at the start of the game. This building has low health and is slow to build.


Tribe center building

This building serves as resource center, can be built only once and spawns the shaman when it is killed. This building is expensive, takes very long to build and has a lot of health.


House building

The house building serves to spawn new workers and increases soft population limit. This building is cheap, fast to build, but has very low health.


Training grounds building

This building serves to train warriors from workers. This building requires the player to have at least one constructed house building. This building is fairly cheap, is quick to build and has medium health.


Archery range building

This building serves to train archers from workers. This building requires the player to have at least one constructed training grounds building.


Weapons-master's camp building

This building serves to train spearmen from warriors. This building requires the player to have at least one constructed training grounds building. This building has fair amount of health, is expensive and takes medium time to build.


Armor-master's camp building

This building serves to train shield-bearers from warriors. This building requires the player to have at least one constructed training grounds building. This building has fair amount of health, is expensive and takes medium time to build.


Stables building

This building serves to train scouts from workers. This building requires the player to have constructed tribe center building.


Riding-trainer's camp

This building serves to train mounted warriors from spearmen and mounted archers from archers. This building requires the player to have at least one constructed stables building. This building has medium amount of health, is expensive and takes medium time to build.

4.3.3 Resources

Food resource

This resource is fairly common on the map. Food is fast to gather, but does not last long (is depleted quick).

Stone resource

This resource is less common on the map. Stone is slow to gather, but has very large amount of how many times it can be gathered

4.3.4 Spells

There are three types of spells, first type targets allies, second type targets enemies and the last type are neutral spells that either do not do damage or do not discriminate when dealing damage.

Blessing of the water

This spell will increase the movement speed of friendly units in area, but will reduce their attack damage. The effect will slowly diminish until the values returns to their original state.

Blessing of the fire

This spell will increase the attack damage of the friendly units in the area, but will deal damage to them. The attack speed bonus will fade slowly over time.

Blessing of the earth

This spell will increase health of friendly units, but will slow their attack speed and movement speed.

Heal

This spell will fully heal all friendly the targets in the area.

Curse

This spell will inflict damage on the enemies and additional damage over time.

Disruption

This spell will shake the ground around target tile and damage anything that is in range. Once the spell's effect wears off, it kills any non-shaman unit that remained in the area.

Rejuvenation

This spell will replenish a resource.

Volcano

This spell will raise the volcano, dealing heavy damage to any unit in the area where the volcano is rising.

5 Advanced user documentation

In this chapter we will focus on the extendability of the game an on adding new content to the game. We will describe individual fields in the XML data and provide an example of an extension by adding new unit, building, resource and spell.

5.1 XML data

The XML data define the properties of game objects or spells. They are stored in the /Data folder and should conform to the Game.xsd schema file. The data definitions might be in multiple files since all XML files from the /Data folder are loaded and their type is determined by the root element of the file. We will now describe individual fields, for the type (integer, float, string) of the field please refer to the schema file.

5.1.1 Unit entry

The unit entry consists of following fields:

- `Id`, which is an identifier of this unit entry.
- `Name`, which is the displayed name for this unit.
- `Health`, which defines the maximum health of this unit.
- `UnitGroup`, which defines the group of this unit. The buildings extending soft unit limits are extending unit capacity for the unit group of the unit type the building spawns. Therefore, the Tribe center building is increasing the shaman's unit group capacity to one, the House building is increasing the worker's unit group capacity by two. And since all other units (except for shaman) are sharing the unit group with the worker, no additional workers are spawned when the unit is trained to another unit.
- `Texture`, which defines the name of the texture from the /Gfx folder without the “tex_” prefix.
- `Shaman`, which defines the unit as shaman. When player does not have any unit with this flag and does not have any building that spawns unit that would have this flag, the player is defeated. Units with this flag can cast spells.
- `Speed`, which defines unit's speed. Absence of this field will disable move order capability of the unit.
- `PopCount`, which defines the “population cost” of this unit. If a unit has for example `PopCount` set to two, it will be counted as two units for the purposes of unit limits.
- `GivenOnStart`, which defines how many units of this type will be given to player at the start of the game.
- `AttackRange`, which defines the range at which the unit can attack.

- `AttackSpeed`, which defines how frequent the attacks will be.
- `AttackAmount`, which defines how powerful the attacks will be. Unit without this field is not able to perform attack order.
- `ConstructRange`, which defines range at which the unit can construct a building.
- `ConstructSpeed`, which defines how frequent the contribution to the construction will be.
- `ConstructAmount`, which defines how much the unit will contribute to the construction. Unit without this field can not perform construct or build order.
- `GatherRange`, which defines range at which this unit can gather resources.
- `GatherSpeed`, which defines how fast this unit can gather resources.
- `GatherAmount`, which defines how much resource can unit gather per tick.
- `GatherCapacity`, which defines how much resource can unit carry. Unit without this field can not gather resources.
- `Modifiers`, which defines the attack modifiers against certain types of units and contains a list of Modifier with fields:
  - `UnitId`, which identifies target unit type (Unit entry's `Id`).
  - `Mod`, which modifies the power of the attack. Value of 1 means no change.

5.1.2 Building entry

The building entry consists of following fields:

- `Id`, which is an identifier of this building entry.
- `Name`, which is the displayed name for this building.
- `Tier`, which only defines where in the menu will the entry be shown. Lower values are located earlier in the menu.
- `Health`, which defines the maximum health of this building.
- `ConstructionTime`, which defines the amount of construction required before the building is constructed.
- `Texture`, which defines the name of the texture from the /Gfx folder without the “tex_” prefix.
- `TextureConstruction`, which defines the name of the texture from the /Gfx folder without the “tex_” prefix. This texture is used when the construction of the building isn't finished.
- `ResourceCenter`, which defines building, where units can unload gathered resources.
- `BuiltOnStart`, which defines whether players will start with this building.
- `OnlyOneAllowed`, which defines whether only one building of this type can be built.

- `UnlockedBy`, which defines what building is required in order to be able to build this building (Building entry's `Id`)
- `Costs`, which defines the cost of this building, its fields are mentioned later
- `Trains`, which defines what units can this building train, contains a list of `Train` with fields:
  - `From`, which defines what unit can be trained (Unit entry's `Id`).
  - `To`, which defines what unit will emerge (Unit entry's `Id`).
  - `Speed`, which defines how long will the training take.
  - `Costs`, which defines the cost of the training, its fields are mentioned later.
- `Spawns`, which defines what units can this building spawn, contains a list of `Spawn` with fields:
  - `Entry`, which defines the unit type (Unit entry's `Id`).
  - `Speed`, which defines how long will the spawning take. If speed is set to -1, this building will not spawn this unit.
  - `Capacity`, which defines how much will this building add to the soft limit for the entry's unit class.

Common sub-entries

- `Costs` field contains a list of `Cost` with fields:
  - `Resource`, which defines the resource type (Resource entry's `Id`)
  - `Amount`, which defines amount of the resource required.

5.1.3 Resource entry

The resource entry consists of following fields:

- `Id`, which is an identifier of this resource entry.
- `Name`, which is the displayed name for this resource.
- `Texture`, which defines the name of the texture from the /Gfx folder without the “tex_” prefix.
- `StartWith`, which defines how much of this resource will every player have at the start of the game.
- `Amount`, which defines the amount of the resource in single instance.
- `OccurChange`, which defines the rarity of this resource on the map.
- `Cooldown`, which defines how often can be gathered from this resource.

5.1.4 Effect entry

The effects only define the metadata for the spell, the code itself is in the script file, as described in the next sub-chapter. The effect entry consists of following fields:

- `Id`, which is an identifier of this effect entry.
- `Name`, which is the displayed name for this effect.

- `Tier`, which only defines where in the menu will the entry be shown. Lower values are located earlier in the menu.
- `Texture`, which defines the name of the texture from the /Gfx folder without the "tex_" prefix. This texture is displayed in the menu.
- `SpellCodeId`, which is the id of the script tied with this entry.
- `Duration`, which is the duration of the effect (number of update calls).
- `Speed`, which defines how often the update is called.
- `CastRange`, which defines the range shaman has to be from the target.
- `AutoRepeat`, which defines, whether the casting is automatically canceled after first cast.
- `Cooldown`, which defines the period of time shaman can not perform any action.
- `UnlockedBy`, which defines what building is required in order to be able to cast this spell (Building entry's `Id`)

5.2 Script data

The script data define the behavior of the spells. In order to make a script file one needs an editor for C# files. Plain notepad would suffice, but it will not highlight the syntax, which makes the programming more difficult. The scripts are stored in the C# source files (cs file extension) in the /Data folder. When the game is initialized, all the C# source files from this folder are compiled.

5.2.1 SpellEntry

The basic element of a script file is a class that implements the `ISpellCodeProvider`. This interface implements one method – `List<SpellEntry> GetEntries()`, that returns the list of `SpellEntry` structs, that defines the spell. This `SpellEntry` struct then contains following fields:

- `Id`, which is an id of the spell used by the spells XML data to know which code are the XML metadata for.
- `Target`, which specifies the target type for the spell – it can be either `TargetType.Tile` which targets tile or `TargetType.GameEntity`, which targets game entity (unit, building or resource). Depending on `Target`, there are two delegates that are called when the spell is cast:
  - `OnSpellStartTile`, which accepts `IScriptGame` parameter representing the game, `IScriptGameEffect` parameter representing the spell itself, `IScriptUnit` parameter representing the caster (shaman) and `IScriptTile` representing the tile this spell was cast upon.
  - `OnSpellStartGameEntity`, where first three parameters are same like in the previous case, but the last parameter is `IScriptGameEntity` representing the target unit, building or resource this spell was cast upon.

- `OnSpellStep`, which specifies single update of the spell and has `IScriptGame` parameter representing the game and `IScriptGameEffect` representing the spell itself.
- `OnSpellFinish`, which specifies action that happens on the last update and has same parameters as the `OnSpellStep` delegate.

This example will create an empty spell code.

```
using WildmenExposedData;
namespace MyNamespace {
  public class MyClass : ISpellCodeProvider {
    public List<SpellEntry> GetEntries() {
      SpellEntry exampleSpell = new SpellEntry();
      exampleSpell.Id = "exampleSpell";
      exampleSpell.Target = SpellEntry.TargetType.Tile;

      // Spell code here

      List<SpellEntry> entries = new List<SpellEntry>();
      entries.Add(exampleSpell);
      return entries;
    }
  }
}
```

The most important facts on the example are:

- Our class is public, non-static and implements `ISpellCodeProvider`
- Our class contains public function `GetEntries()` that returns `List<SpellEntries>`
- The `GetEntries()` code does not return null

As you might have noticed, it does not matter how we name our namespace or class. For easier referencing, it is recommended to add "`using WildmenExposedData;`" as shown on line one.

5.2.2 Interfaces

We can find all the interfaces of the game the script can use in the documentation, but let us look at the most important interface functions.

IScriptGame

This interface represents the game. It provides (among others) these main methods:

- `IScriptTile GetTile(int x, int y)`
  will retrieve tile on given x, y position in the map.
- `IScriptUnit CreateUnit(IScriptDbEntry entry, Vector2 position, IScriptPlayer owner)`
  will create unit of given type on given position that will be assigned to target owner.

- `IScriptBuilding CreateBuilding(IScriptDbEntry entry, IScriptTile tile, IScriptPlayer owner)`
  will create building of given type on given tile that will be assigned to target owner.
- `IScriptResource CreateResource(IScriptDbEntry entry, IScriptTile tile)`
  will create resource of given type on given tile.
- `IScriptUnitDbEntry GetUnitDbEntry(string id)`
  will return a unit type for given id.
- `IScriptDbEntry GetBuildingDbEntry(string id)`
  will return a building type for given id.
- `IScriptDbEntry GetResourceDbEntry(string id)`
  will return a resource type for given id.

As we mentioned earlier, we are given instance implementing this interface in the delegates from `SpellEntry` we described. Let us improve our example spell from previous sub-chapter and add code near the "`// Spell code here`" part. Let us

And now we add a code that will spawn a new resource on the tile.

```
exampleSpell.OnSpellStartTile = (game, gameEffect, caster, target) => {
    var foodResource = game.GetResourceDbEntry("resFood");
    var tile = game.GetTile(5, 5);
    game.CreateResource(foodResource, tile);
```

The first line will make a delegate function for the initialization of the spell – the first parameter game is the `IScriptGame` instance. The second line will get `DbEntry` of the resource with "resFood" id, the third line gets the tile with coordinates [5, 5]. And the fourth line will create the resource on the. Now, let us also add a unit on this position.

```
    var workerUnit = game.GetUnitDbEntry("untWorker");
    var player = caster.GetOwner();
    var position = tile.GetPosition();
    game.CreateUnit(workerUnit, position, player);
}
```

The first line will get `DbEntry` of the worker unit. The second line will get the owner player of the shaman that cast this spell. We will look at this command later in next sub-chapter Since unit is placed on vector position unlike resource, we have to first get the position from the tile. Then we can create the unit on the position.

If we save this code now into the /Data directory along with XML code with metadata for this spell, this spell once cast will create a food resource and a worker unit on the tile [5, 5].

IScriptTile

This interface represents a map tile. It provides (among others) these main methods:

- `int GetElevation()`
  will return the elevation of the tile

- `void SetElevation(int newHeight)`
  sets a new elevation for the tile
- `Vector2 GetPosition()`
  gets the vector position of the tile, this is useful when we need spawn a unit on a tile, since the unit does not accept tile, but vector
- `List<IScriptUnit> GetUnits()`
  returns list of units on this tile
- `IScriptBuilding GetBuilding()`
  returns the building that is occupying this tile (or null if there is none)
- `IScriptResource GetResource()`
  returns the resource that is occupying this tile (or null if there is none)

IScriptPlayer

This interface represents a player. It provides (among others) these main methods:

- `IScriptUnit GetShaman()`
  returns (first) shaman unit of the player
- `List<IScriptUnit> GetUnits()`
  returns list of player's units
- `List<IScriptBuilding> GetBuildings()`
  returns list of player's buildings
- `int GetResourceAmount(IScriptDbEntry resourceType)`
  returns how many of target resource player has.
- `void SetResourceAmount(IScriptDbEntry resourceType, int newAmount)`
  sets the amount of the given resource player has.

As we might notice, we already used player when we were creating new unit in the last example. Let us improve it even further and modify the last lines of the delegate `OnSpellStartTile`.

```
    game.CreateUnit(workerUnit, position, player);
    // IScriptPlayer example starts here
    foodAmt = player.GetResourceAmount(foodResource);
    player.SetResourceAmount(foodResource, foodAmt + 100);
}
```

We have added the second, third and fourth line. The third line detects amount of food does the player have right now and the fourth will set the amount increased by 100.

IScriptGameEntity

This interface, as we might have noticed earlier in this chapter, is used when the shaman casts spell on a game entity. We have to first determine what entity the spell was cast upon and then re-cast the target into the proper `IScript-` interface (`IScriptUnit`, `IScriptBuilding` or `IScriptResource`). It provides this property:

- `GameEntityType GetEntityType`
  will return the enum type specifying the type of the entity

IScriptResource

This interface represents a resource placed on the map. This interface is descendant of `IScriptGameEntity`. It provides (among others) these main methods:

- `IScriptTile GetNearestTile()`
  will return the nearest `IScriptTile` (the tile the resource is on).
- `int GetAmount()`
  will return the amount of charges remaining in this resource.
- `void SetAmount(int newAmount)`
  will set new amount of charges for the resource.

Let us create a new spell. This spell will increase the amount of charges of a resource.

```
SpellEntry exampleSpell2 = new SpellEntry();
exampleSpell2.Id = "exampleSpell2";
exampleSpell2.Target = SpellEntry.TargetType.GameEntity;
exampleSpell2.OnSpellStartGameEntity = (game, gE, caster, target) => {
  if (target.GetEntityType != GameEntityType.Resource) return;
  var resource = (IScriptResource)target;
  int resAmt = resource.GetAmount();
  resource.SetAmount(resAmt+100);
}
```

The first four lines are similar to the last example, but instead of targeting tile, we target a game entity. The fifth line checks whether the target is resource or not, seventh line will get the current amount of charges from the resource and the eighth line will set new amount.

Do not forget to make the XML entry for this new spell and add the spell to the list of spells.

```
entries.Add(exampleSpell2);
```

IScriptBuilding

This interface represents a building placed on the map. This interface is descendant of `IScriptGameEntity`. It provides (among others) these main methods:

- `IScriptTile GetNearestTile()`
  will return the nearest `IScriptTile` (the tile the building is based on).
- `int GetHealth()`
  returns the current amount of health of the building.
- `void SetHealth(int newHealth)`
  sets the amount of health of the building.
- `void Kill(bool noMessage = false);`
  destroys the building.
- `int GetConstructionProgress()`
  gets the current progress of the construction.

- `void SetConstructionProgress(int newProgress)`
  sets the construction progress.
- `bool IsConstructed();`
  returns whether the building is complete.
- `IScriptPlayer GetOwner();`
  returns the owner of the building.

Let us make a new script. We will take the last example "`exampleSpell2`" and just change the content of the `OnSpellStartGameEntity` delegate.

```
if (target.GetEntityType != GameEntityType.Building) return;
var building = (IScriptBuilding)target;
if (building.IsConstructed()) {
  int currentHp = building.GetHealth();
  building.SetHealth(currentHp / 2);
}
else { building.Kill(); }
```

Just like in the last example, the first line checks whether we have targeted a building. If so, we check (line three) whether the building is constructed or not. If it is constructed, we damage it for half of its current health, if it is not, we will destroy the structure (line seven).

IScriptUnit

This interface represents a building placed on the map. This interface is descendant of `IScriptGameEntity`. It provides (among others) these main methods:

- `IScriptTile GetNearestTile()`
  will return the nearest `IScriptTile` to this unit.
- `int GetHealth()`
  returns the current amount of health of the unit.
- `void SetHealth(int newHealth)`
  sets the amount of health of the unit.
- `void Kill(bool noMessage = false);`
  kills the unit, the `noMessage` specifies whether a scroll-up message should be generated.
- `Vector2 GetPosition()`
  gets the current position.
- `void SetPosition(Vector2 newPosition)`
  sets the position.
- `bool IsConstructed()`
  returns whether the building is complete.
- `IScriptPlayer GetOwner()`
  returns the owner of the unit.
- `IScriptUnitDbEntry GetDbEntry()`
  returns the clone of the `IScriptDbEntry` of this unit. It's important to note that

this will clone the entry and if the entry is modified, it needs to be re-applied to unit.

- `void SetDbEntry(IScriptUnitDbEntry newEntry)`
  clones the `IScriptDbEntry` and applies it to this unit, if any value is invalid then nearest valid value is picked instead (1 for Health and 0 for all other editable properties).

Let us make another script. We will take the same example as the last time, the "`exampleSpell2`" and just change the content of the `OnSpellStartGameEntity` delegate.

```
if (target.GetEntityType != GameEntityType.Unit) return;
var unit = (IScriptUnit)target;
var unitDbEntry = unit.GetDbEntry();
unitDbEntry.Speed = 1.5f;
int maxHp = unitDbEntry.Health;
unit.SetHealth(maxHp);
unit.SetDbEntry(unitDbEntry);
```

Now, this example first checks whether the type of the target is a unit on line one. On line three we retrieve the `IScriptDbEntry` from the unit. On line four we set a new speed for this entry, on line five and six we heal the unit to its full health and finally, on line seven we set the `IScriptDbEntry` back to the unit. So, this code will heal and speed up (or slow down) a unit (when the spell is cast on a unit, of course).

IScriptGameEffect

The `IScriptGameEffect` represents the spell itself. It is passed to the delegates and contains following properties:

- `int Time`
  which will return the number of the current step since the start of the spell
- `int Duration`
  which will return the total number of steps this spell will have
- `List<object> LocalData`
  which allows the script to store data between the steps, initialization and finalization. These data have to be either game objects, tiles or data that can be converted to and from a string.

5.3 Example of new content addition

We will now show on an example how to add new content to the game. We will add a new resource Awesomnium, which will allow the construction of the Spectral barracks building which will allow the training the Warrior unit to the a Spectre unit – a low health unit with lethal short-ranged melee attack. Then we will add a spell that will all non-Spectre friendly units in range, boost the health of all Spectre units in range and kill them after the spell wears off.

Textures

First we prepare five textures that will represent new resource, unit, building, building in construction and spell. We will name our textures (on the figure 23 from left to right)

tex_unit_spectre.png, tex_resAwe.png, tex_structureSpecBarracks.png, tex_structureSpecBarracksProgress.png and tex_spectral_boost.png and we will put them in the /Gfx folder. We included our example textures in attachment [D].

Now, we will create following five files in the /Data folder.

Resource

SpecResource.xml with following XML content will make a resource entry with aweRes id named Awesomnium,

```
<?xml version="1.0" encoding="utf-8" ?>
<Resources xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Game.xsd">
  <Resource>
    <Id>resAwe</Id>
    <Name>Awesomnium</Name>.
```

it will use the resAwe texture (tex_resAwe.png file),

```
    <Texture>resAwe</Texture>
    <StartWith>0</StartWith>
```

every instance of the resource will contain 300 of awesomnium,

```
    <Amount>300</Amount>
```

it will have 0.4% change of spawning on a tile,

```
    <OccurChance>0.004</OccurChance>
```

and it unit will be able to gather the resource every 120 update frames (which is circa 2 seconds).

```
    <Cooldown>120</Cooldown>
  </Resource>
</Resources>
```

Unit

SpecUnit.xml with following XML content will make a unit with spectreUnit id named Spectre.

```
<?xml version="1.0" encoding="utf-8" ?>
<Units xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Game.xsd">
  <Unit>
    <Id>untSpectre</Id>
    <Name>Spectre</Name>
```

It will have 20 health points,

```
    <Health>20</Health>
```

will belong to unit group 1 (same unit group as Worker, Warrior, etc),

```
    <UnitGroup>1</UnitGroup>
```

unit will use unit_spectre texture (tex_unit_spectre.png file)

```
    <Texture>unit_spectre</Texture>
```

and will move 1.3x faster than normal speed.

```
    <Speed>1.3</Speed>
```

The unit will have attack range of 10 pixels

```
    <AttackRange>10</AttackRange>
```

and will deal 60 damage every 10 update frames.

```
    <AttackSpeed>10</AttackSpeed>
    <AttackAmount>60</AttackAmount>
  </Unit>
</Units>
```

Building

SpecBuilding.xml with following XML content will make a building with specBarracksBuilding id named Spectral Barracks.

```
<?xml version="1.0" encoding="utf-8" ?>
<Buildings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Game.xsd">
  <Building>
    <Id>bldSpectralBarracks</Id>
    <Name>Spectral Barracks</Name>
    <Tier>2</Tier>
```

The buildings will have 450 health points

```
    <Health>450</Health>
```

and will require 300 construction points before it will be constructed.

```
    <ConstructionTime>300</ConstructionTime>
```

It will use structureSpecBarracksProgress texture for non-finished version of the building and structureSpecBarracks texture for constructed version of the building.

```
    <Texture>structureSpecBarracks</Texture>
    <TextureConstruction>structureSpecBarracksProgress</TextureConstruction>
```

The building is unlocked by constructing training grounds

```
    <UnlockedBy>bldTrainGrounds</UnlockedBy>
```

and will cost 500 of awesomnium to create this building

```
    <Costs>
      <Cost>
        <Resource>resAwe</Resource>
        <Amount>500</Amount>
      </Cost>
    </Costs>
```

This building can train warrior unit into spectre unit in 250 update frames costing 20 of the awesomnium resource.

```
    <Trains>
      <Train>
        <From>untWarrior</From>
        <To>untSpectre</To>
```

```
        <Speed>250</Speed>
        <Costs>
          <Cost>
            <Resource>resAwe</Resource>
            <Amount>20</Amount>
          </Cost>
        </Costs>
      </Train>
    </Trains>
  </Building>
</Buildings>
```

Spell

SpecEffect.xml with following XML content will make a spell with spectralSurgeSpell id named Spectral surge.

```
<?xml version="1.0" encoding="utf-8" ?>
<Effects xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Game.xsd">
  <Effect>
    <Id>splSpectralSurge</Id>
    <Name>Spectral surge</Name>
    <Tier>1</Tier>
```

Spell will use spectral_boost texture in the selection menu

```
    <Texture>spectral_boost</Texture>
```

and will use code with spectralSurgeCode id.

```
    <SpellCodeId>spectralSurgeCode</SpellCodeId>
```

The spell will update 10 times and will wait 60 frames between updates.

```
    <Duration>10</Duration>
    <Speed>60</Speed>
```

Shaman will have to be at most 20 pixels away from the target,

```
    <CastRange>20</CastRange>
```

will not automatically repeat the spell

```
    <AutoRepeat>false</AutoRepeat>
```

and will have to wait for 600 updates after casting the spell before it can do anything.

```
    <Cooldown>600</Cooldown>
```

The spell is unlocked by building spectral barracks.

```
    <UnlockedBy>bldSpectralBarracks</UnlockedBy>
  </Effect>
</Effects>
```

And finally a SpecSpellScript.cs with following script content:

```
using System.Collections.Generic;
```

```
using WildmenExposedData;

namespace WildmenSpellLibrary
{
  public class Main : ISpellCodeProvider
  {
    private SpellEntry SpectralSurgeEntry()
    {
```

We create a new SpellEntry,

```
      SpellEntry entry = new SpellEntry();
```

set the id and target type,

```
      entry.Id = "spectralSurgeCode";
      entry.Target = SpellEntry.TargetType.Tile;
```

then we set the initializing function, which will first select all units on the tile and tiles surrounding this tile,

```
      entry.OnSpellStartTile = (game, gameEffect, caster, target) =>
      {
        List<IScriptUnit> unitsInRange = new List<IScriptUnit>();
        unitsInRange.AddRange(target.GetUnits());

        foreach (var d in Misc.OmniDirection)
        {
          IScriptTile surrtile = target.GetSurroundingTile(d);
          if (surrtile != null)
          {
            unitsInRange.AddRange(surrtile.GetUnits());
          }
        }
```

and then will filter out spectre units which will get the boost (and will be stored in the `LocalData` list) and all other non-shaman units of the owner will be killed.

```
        foreach (var unit in unitsInRange)
        {
          int currentHp = unit.GetHealth();
          if (currentHp == 0) continue;

          if (unit.GetDbEntry().GetId == "untSpectre")
          {
            unit.SetHealth(currentHp + 40);
            gameEffect.LocalData.Add(unit);
          }
          else if (!unit.IsShaman && unit.GetOwner() == caster.GetOwner())
          {
            unit.Kill();
          }
```

```
      }
    };
```

We will skip update function as we do not have any use for it and will define finalizing function which will kill all the boosted spectre units.

```
    entry.OnSpellStep = null;
    entry.OnSpellFinish = (game, gameEffect) =>
      {
        foreach (IScriptUnit unit in gameEffect.LocalData)
          if (unit.GetHealth() != 0)
            unit.Kill();
      };

    return entry;
  }
```

Then we have to expose the list of the spells we defined in this script to the game through the ISpellCodeProvider interface.

```
  public List<SpellEntry> GetEntries()
  {
    List<SpellEntry> entries = new List<SpellEntry>();
    entries.Add(SpectralSurgeEntry());
    return entries;
  }
 }
}
```

Now we only make sure all the players have our new files and we can play with these new game objects and spell (figure 24).

Figure 23: New textures for the example content

Figure 24: Spectre unit with spectral barracks and an awesomnium resource.

6 Comparison

In this chapter we will compare our game with the selection of other real-time strategy games that are similar to our game.

Populous 3: The Beginning

The Populous 3: The Beginning [3] is the game we were inspired by the most and where some of the core features of our game are from. The Populous game is situated in the world represented by spherical 3D model where the shaman is the central unit which can cast powerful spells and alter the landscape. The spells and new buildings are gained by worshiping gods in the altars placed throughout the map. The regular units are gained through the house buildings, which produces simple units that can be then trained in special buildings. The Populous 3 game is however older game (released in 1998) and newer operating systems and the newer graphics cards have problems with starting this game.

The most apparent difference between our game and Populous 3 is the difference in map representation. The Populous game has 3D map, whereas our game is played out on a 2D map. With the map representation is also associated the portrayal of game objects where we can use simple textures allowing for simpler addition of new graphical content to the game.

The Populous game, unlike our game, does not have the fog of war concept that conceals areas of the map when user does not have any unit nearby, which adds an element of surprise to the game. And as we hinted, our game supports extending the game content directly by editing or adding text files containing XML or script data, which can not be done directly in the Populous 3 game.

Our game, however, does not contain any form of single player campaign or AI player, unlike the Populous 3 game.

Age of Empires 2

The Age of Empires 2 [2] is another real-time strategy game and, just like in our game, its game map is a two-dimensional map with isometric perspective. However, the game does not represent very well various degrees of differences between terrain elevation.

This game offers various victory conditions, the most similar to ours is the “Protect the King” victory type, where players have to defend their central unit – king. However, unlike our game, this unit can not perform any offensive actions and is completely defenseless.

While the Age of Empires 2 allows modifications of the game, it requires additional software to do so, unlike our game that allows to change the game data directly. Also, unlike our game, the units in the Age of Empires game does not have to be trained from basic units but are produced directly.

This game also provides a single-player campaign and custom games against AI players, which our game does not offer.

Warcraft 3

The Warcraft 3 [25] game is a 3D real-time strategy game. The 3D map, just like in the Populous game, allows the terrain to form ramps, slopes or cliffs, however with the fog of

war mechanism. The unit production system is same as in the Age of Empires 2, where units can be directly trained from the building.

The Warcraft 3 game allows the player to build a hero units, that will gain experience which will allow the unit to gain powerful abilities, however, unlike in our game, when this unit is killed (and the player does not have any means of it re-creation), the player is not defeated. The exception is the single-player, where player might have to protect a hero from the AI players. This brings us to the topic of single-player, which the Warcraft 3 game offer both in form of campaign and in form of custom skirmish against computer, neither of which our game game provides.

The game content can be modified, but again, 3rd party software is needed in order to change the data and use of 3D models makes the modding even less accessible to the inexperienced users.

Command and Conquer 4: Tiberium Twilight

The Command and Conquer 4: Tiberium Twilight [7] is basically a Warcraft 3 style game, but differs from other real-time strategies with the concept of a base as a moving unit, which can produce units “on the go”. This game, however, entirely dropped the resource gathering side of the genre and units will only take time to be built, not resources. Additionally, there is no base building and the decision what units the player has access to is decided outside the game through a leveling system, where player gets points for playing games. When certain level is reached a new set of units is unlocked for the player and the base may build them.

This base-unit is certainly a powerful and critical unit for the player, filling same position as the shaman in our game. The number of units player can build is also reduced and set fairly low, having similar functionality to our unit soft-limit.

Other than that, the Tiberium Twilight shares many similarities with the Warcraft 3 game (as we mentioned earlier). The map is three dimensional, with fog of war. The game alongside multiplayer also offers singleplayer and campaign. The modding options for this game are basically nonexistent.

The Settlers 7: Paths to a Kingdom

The Settlers 7: Paths to a Kingdom [8] offers a different take on real-time strategy games. In this game player is building a city and has to take care of the entire production chain, starting with fishing, ending with weapon forging. Player has a number of available civilians and is assigning them to buildings or conscripting them to military (if the player has sufficient amount of resources).

The game is played out on a 3D map with no fog of war. The player is constructing buildings, which will gather or produce resources (if the building is occupied by a civilian). When civilians are recruited into military, they can attack other player's buildings. Player wins either by eliminating all opponents or by accumulating victory points (by non-military means). There is no critical unit.

Again, this game offers single-player games and can not be extended. Unlike our game.

Other similar products

Other notable real-time strategies are more-less conceptually and mechanically similar to those we mentioned. The StarCraft [26] and Command and Conquer: Red Alert 2 [27] are similar to the Age of Empires 2 game with its isometric graphics. The StarCraft 2 [1], Red Alert 3 [28] or Emperor: Battle for Dune [29] games are then similar to the Warcraft 3 only placed in different universe and different time. And the Anno 2070 [30] game is similar to the Settlers 7 game. While these games are not exactly similar and are running on different game engines, the basic concepts (like direct production of units or the lack of critical unit) are same.

Summary

Now, that we have compared our game to similar products we can summarize our observations.

Our game have the two-dimensional graphics, which puts us at a disadvantage as the 3D games have better graphical fidelity and allows better orientation on the map. However, the use of simpler (2D) graphics allows us to have easier extendability of our game allowing less skilled users to add and modify the content of the game, which is not what we usually see in the RTS games.

We can see that the concept of gradually training our units from the very basic ones to the specialized ones is not very usual among the games as they are rather built directly by proper building. The concept of a powerful critical unit is also uncommon and appears just in the Populous 3 game (where we adopted the idea from), Command and Conquer 4: Tiberium Twilight and occasional occurrences in single-player campaigns.

A significant difference between our game and the similar games is the lack of single-player play in our game. As we noticed, most of the games have both multi-player and single-player in form of either custom skirmishes or scripted campaigns.

7 Conclusion

In this chapter we will conclude this thesis. We will assess the result and we will compare it with the goals we established in the chapter 1.2.

The game is played out on two-dimensional map divided into tiles, which have given elevation. We look at this map from 2D isometric perspective, which allows us to see the elevations of individual tiles. The player has won the game if all other players have lost their shaman unit and none of their building can re-produce it. In order to achieve victory players utilize these three types of game objects: units, buildings and resources.

The units can receive orders which interacts with other game objects on the map and the map itself. The units can move around map, with exception of moving into tile with steep elevation difference or below sea level. There is several types of units and each has different properties. We have provided 9 unit types - a shaman unit, a basic worker unit and 7 combat units. The shaman unit can cast spells, which modifies properties of the players, map, map tiles or game objects. Once a spell is cast, the shaman is unable to receive orders for brief moment. The worker units gather resources and builds buildings. The combat units each have different attack ranges and bonuses against other types which forces player to diversify the composition of their army.

The buildings serve as progression system where buildings can unlock spells and other buildings and can train specialized units from more basic ones. Just like units, there can be multiple types of buildings which define their properties. The player can build total of 9 types of buildings. Some buildings extend the population limit, which determines how many units player can have. Some buildings may start producing a basic unit when the population limit is not reached, depending on their type. The buildings require resources in order to be built. When the building is placed down it is in construction mode, where it can not be used until building finishes the construction.

The resource game objects are gathered by units and then used for either constructing new buildings or training new units. They are randomly placed around the map at the beginning of the game and they have limit on how many times they can be gathered.

The game content is stored in the XML files, which define the properties of individual types of game objects. The graphical data are stored in the PNG files, allowing to add or modify the graphical representation of game objects. The spells are defined by the code in the C# source files and these files are automatically compiled when the game starts.

The game features multiplayer game mode, where players connects to the instance of the game running in server mode, which can accept up to 4 clients. and the game is played in real-time, meaning that the players are ordering their forces concurrently with other players.

We have made this game with the use of the C# language, Microsoft's .NET Framework and SharpDX libraries.

When we compare our game with the goals mentioned in the chapter 1.2 we can conclude that we have fulfilled all of them.

8 Future work

While our game matches the goals we set up, we have noticed during the development certain areas of the game that could be improved further.

- As we compared our game with other games, majority of similar RTS games contain some form of single-player, either in form of a campaign with a story or just as a game where instead of human players we play against computer. Therefore, adding an AI player would certainly bring the game closer to the modern games.
- The units in this game are walking straight-forward to their mission and when they reach an obstacle they can not pass, they will stop. It would improve the quality of the gameplay if a path-finding algorithm is implemented.
- While the map generation is sufficient for our use, it could certainly be further improved. Adding a map editor that would allow to create own maps could help ensuring fair starting points for all involved players.
- This game allows to create new network connections only when the game is being set up. Once the game is started it is not possible for another player to join and spectate the game, or replace an inactive player that has quit the game. Allowing the players to reconnect if their connection fails would improve the user experience from the game.
- The units can only perform limited amount of orders. It is common in other games for unit to be able to perform special ability of the unit, which adds additional value to the unit and requires the player to concentrate on controlling of these units as their abilities have to be manually activated.
- The user interface of the game could be improved allowing for better orientation of the player by adding information about selected units or adding a minimap – a miniaturized view of the game, which would allow to react on situations that happen outside the current view.
- The orientation could also be improved if units were forbidden from crossing and moving into one another when performing orders. The lack of this mechanism makes larger combat situation difficult.

9 Attachments

Attachment A: Source code

The source code for the Wildmen solution can be found in the /WildmenSource directory. The folder contains the Wildmen project with the game source code, WildmenExposedData project with source code of exposed data to the script files and WildmenSetup project which compiles the installer of the game using the VisualStudioInstaller plug-in [24].

Attachment B: Documentation

The documentation is located in the /Documentation directory and is made from the source code using the Sandcastle Help File Builder [24] project.

There are two files, one WildmenDocumentation.chm contains documentation of all members and methods from the Wildmen and WildmenExposedData projects. The WildmenExposedDataDocumentation.chm contains documentation of public members and methods from the WildmenExposedData project.

Attachment C: Installer

The setup can be found in the /Setup directory and the game is installed by running the setup.exe program.

Attachment D: Example extension data

Example extension data are provided in the /ExtensionExample directory.

10 References

1. StarCraft 2: Wings of Liberty game. Official website: http://us.battle.net/sc2/en/
2. Age of Empires II: The Age of Kings game. Official website: http://www.ageofempires.com/AoE2.aspx
3. Populous 3: the Beginning game. Wikipedia article: http://en.wikipedia.org/wiki/Populous:_The_Beginning
4. GameSpy: Supreme Commander Interview article. Online document available at http://pc.gamespy.com/pc/supreme-commander/631678p1.html
5. SimCity 2000 game. Wikipedia article: http://en.wikipedia.org/wiki/SimCity_2000
6. Wikipedia article about strategy. Online document available at: http://en.wikipedia.org/wiki/Strategy
7. Command & Conquer 4 Tiberian Twilight game. Official website: http://www.ea.com/command-and-conquer-4
8. The Settlers 7: Paths to a Kingdom game. Official website: http://thesettlers.uk.ubi.com/the-settlers-7/
9. Minecraft game. Official website: https://minecraft.net/
10. Wikipedia article about Perlin noise. Online document available at: http://en.wikipedia.org/wiki/Perlin_noise
11. Rendered height-map image. Online source: http://en.wikipedia.org/wiki/File:Heightmap_rendered.png
12. Polygonal Map Generation for Games article. Online document available at: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/
13. Polygon elevation map image. Online source: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/elevations.png
14. Dune 2 game. Wikipedia article: http://en.wikipedia.org/wiki/Dune_II
15. Dune 2 screenshot image. Online source: http://www.myabandonware.com/media/captures/D/dune-ii-the-building-of-a-dynasty/dune-ii-the-building-of-a-dynasty_14.jpg
16. Mario game. Wikipedia article: http://en.wikipedia.org/wiki/Super_Mario_Bros.
17. Screenshot from Mario game. Online source: http://www.ellick-lee.com/wp-content/uploads/2013/01/8Bit-eLlicky-Mario.png

18. SimCity 2000 screenshot image. Online source: http://www.freegameempire.com/Img/Cache/Games/SimCity-2000/Screenshot-2.png
19. Unity3D game engine. Official website: http://unity3d.com/
20. Microsoft XNA framework. Official website: http://msdn.microsoft.com/en-us/centrum-xna.aspx
21. SharpDX framework. Official website: http://sharpdx.org/
22. OpenTK, OpenGL wrapper for C#. Official website: http://www.opentk.com/
23. MonoGame libraries. Official website: http://www.monogame.net/
24. Visual Studio Installer Projects plug-in. For Visual Studio Website: http://visualstudiogallery.msdn.microsoft.com/9abe329c-9bba-44a1-be59-0fbf6151054d
25. Warcraft III: Reign of Chaos game. Official website: http://us.blizzard.com/en-us/games/war3/
26. Starcraft game. Official website: http://us.blizzard.com/en-us/games/sc/
27. Command & Conquer: Red Alert 2 game. Wikipedia article: http://en.wikipedia.org/wiki/Command_%26_Conquer:_Red_Alert_2
28. Command & Conquer: Red Alert 3 game. Official website: http://www.ea.com/command-and-conquer-red-alert-3
29. Emperor: Battle for Dune game. Wikipedia article: http://en.wikipedia.org/wiki/Emperor:_Battle_for_Dune
30. Anno 2070 game. Official website: http://anno-game.ubi.com/anno-2070/en-GB/home/
31. Sandcastle Help File Builder project. Website: https://shfb.codeplex.com/
32. Algorithm for generating 2d perlin noise. Website: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm