Charles University in Prague

Faculty of Mathematics and Physics

# BACHELOR THESIS

Adam Hanka

# Engine for Real-time Strategy (RTS) Games

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.

Study program: Computer Science (B1801)

Specialization: Programming

Prague 2013

I would like to thank my supervisor, Mgr. Pavel Ježek, PhD, for the patient guidance, advice and useful suggestions he has provided me throughout the work on my Bachelor thesis. I have been lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

I must also express my gratitude to my family for their continued support and encouragement during my Bachelor studies and especially during the time I spent with this thesis.

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

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

In…....... date............ signature

**Název práce:** Engine for Real-time Strategy (RTS) Games

**Autor:** Adam Hanka (adam.hanka@gmail.com)

**Katedra / Ústav:** Katedra distribuovaných a spolehlivých systémů

**Vedoucí bakalářské práce:** Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů (jezek@d3s.mff.cuni.cz)

**Abstrakt:** Realtimové strategie (RTS) jsou velmi populární žánr na poli počítačových her. Bohužel, komerční RTS hry jsou uzavřené a nerozšiřitelné, což znemožňuje jejich důkladné poznání na úrovni zdrojového kódu.

V této práci prezentujeme velmi ilustrativní 2D RTS hru založenou na .NET Frameworku s objektově orientovaným návrhem, která je plně rozšiřitelná a publikována jako open-source. Hra podporuje jak single-player, tak i multiplayer mód s možností hry proti počítačovým hráčům, kteří jsou vedeni umělou inteligencí. Systém umělé inteligence je rozšiřitelný pomocí pluginů.

Projekt obsahuje budovy a jednotky s jejich vlastní vnitřní umělou inteligencí, která jim umožňuje chovat se (částečně) nezávisle. Nové jednotky a budovy i vnitřní inteligence pro ně mohou být přidávány jako kompilované DLL binární soubory.

V tomto projektu dále prezentujeme propracovaný koncept managementu surovin, který umožňuje snadnou rozšiřitelnost sběru, transportu a přeměny surovin.

**Klíčová slova:** RTS hra, realtimová strategie, umělá inteligence pro RTS hry, multiplayer RTS hry

**Title:** Engine for Real-time Strategy (RTS) Games

**Author:** Adam Hanka (adam.hanka@gmail.com)

**Department / Institute:** Department of Distributed and Dependable Systems

**Supervisor of the bachelor thesis:** Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems (jezek@d3s.mff.cuni.cz)

**Abstract:** Real-time strategy (RTS) is a very popular genre of computer games. However, commercial RTS games are closed and not extendable, which prevents the community from investigating RTS games on the source-code level and from tailoring them to their needs.

In the thesis, we present an illustrative, extendable open-source 2D RTS computer game for the .NET framework with an object-oriented architecture. It supports both single-player and multiplayer sessions with the possibility to play against computer players run by artificial intelligence. The system of AI is extendable with plug-ins.

The project contains entities (buildings and units) with their own artificial intelligence, which enables them to behave as (partially) autonomous agents. New entities and artificial intelligence for them can be added easily through a programmer-friendly interface as compiled DLL files.

The project also comprises a developed concept of resource management providing for easy design-extendability of resource gathering, transport and transformation.

**Keywords:** RTS game, game engine, multiplayer RTS games, artificial intelligence for RTS

# Contents

# 1 Introduction

A real time strategy (RTS) is a game making an impression of a continuous time progress. It is a sub-genre of strategy computer games, which has grown to a large extent over the recent years and has become very popular among many computer games players.

Commercial RTS software is closed and not extendable. This prevents the community from investigating RTS games on the source-code level and from tailoring them to their needs. We, by contrast, intend to develop an open source RTS game which would be very illustrative and accessible for the community so that anybody familiar with programming can make use of it.

Although there are many open source RTS games (visit the article List of open-source video games in Wikipedia [1]), they are mostly based on C++ (in the article List of open-source video games, all of them are). However, there is also a large number of programmers familiar with other languages and we would like to make our open source RTS game accessible to them as well. In the past, C++ used to be the development language number one for computer games, but we believe that nowadays, an RTS computer game can also be created using the Microsoft .NET Framework [2] and its language C#.

*For all the reasons, the goal of this thesis is to design and develop an extendable open source RTS computer game. For the development, we will use the .NET Framework.*

Let us first of all investigate how are RTS games generally understood, so that we know what a typical RTS games has to comprise.

## 1.1 Definitions of RTS games

Despite to the popularity of the RTS genre, not many complex definitions of RTS games can be found. Dan Adams [3] describes RTS as "*a military strategy game in which the primary mode of play is in a real-time setting*". The Urban Dictionary [4] says: *"RTS refers to a strategy-based computer game, generally involving management of an army or civilization."* So RTS is a military strategy. The participants (gamers) take the role of a supreme commander in a battlefield. According to the article Real-time strategy in Wikipedia [5], they "*position and maneuver units and structures under their control to secure areas of the map and/or destroy their opponents' assets. In a typical RTS, it is possible to create additional units and structures during the course of a game. This is generally limited by a requirement to expend accumulated resources.*" The article also mentions 4 main components of a typical RTS game:

(c1) resource management
(c2) base building
(c3) war tactics (commanding units in battles)
(c4) technology development

The above definitions exclude city-building games such as SimCity [6] or

Pharaoh [7] due to the absence of military elements, although they process in real time. Similarly, there is a large class of games concentrating on complexity in war tactics and combat while neglecting other components (typically resource management and base building). This genre is called *real-time tactics (RTT)* [8] and it is not considered to be a part of the RTS genre. This category contains for example the Total War series [9] (where resource management is turn-based) or the Warhammer 40,000 [10].

Further, computer games combining elements from both RTS and *turn-based strategy (TBS)* [11] can be found. In such games, either time can be stopped in battles to plan the tactics, or the game progresses in steps between battles held in real-time. Apart from the Total War series, The Lord of the Rings: The Battle for Middle-earth II [12] is one such game.

The genre of RTS games is also distinguished from *god games* [13] (for example Black & White [14]), where the players influence their units (worshipers) only indirectly. Also, players are given supernatural power (e.g., controlling the weather), which is not an element of RTS games.

Now that we know what is an RTS game and what it is not, let us investigate components (c1) – (c4) mentioned in the definitions.

### 1.1.1 Resource management

Resource management symbolizes the player's economy. No matter how many types of resources are present in a particular game, they always stand for a currency; their main purpose is to pay for new structures, units, and for technology development. Each unit, structure and technology has its cost, which is the amount of resources the player expends for constructing/developing it. This concept can be expressed with the following formula:

*Supposed we have resources $R_1$, $R_2$,..., $R_N$, then the cost C of a unit, structure or technology can be expressed as:*

$$C=\sum_{i=1}^{N} a_i \cdot R_i$$

*Where $a_i$ is the amount of resource $R_i$ required for acquiring the unit, structure or technology.*

The article "*Real-time strategy*" [5] gives the following definition of how resources are obtained: "*Resources are garnered by controlling special points on the map and/or possessing certain types of units and structures devoted to this purpose.*" Each game presents different types of resources, gathering them and using them. Examples range from very simple to complex resource-management systems. For example, Dune II [15] features only one type: the spice. A special unit (spice collector) goes to area where spice is present and collects it. On the other hand, in Stronghold [16] for instance, wheat (grown in farms) is ground into flour in mills; the flour is brought to bakery where bread is produced. Extra rations of food can then be given to the population to increase their happiness and raise the taxes income, which, in turn, allows hiring more military units (notice that also abstract resource such as happiness may be present in an RTS game). The chain of resources production can be summarized with the following (recursive) definition:

*Resource $R_{N+1}$ is produced by a unit U or structure S from resources $R_1, R_2, \ldots, R_N$, where $R_i$ is produced from other resources or collected by a unit or obtained directly from controlling an area in the game:*

$$[\![U \,||\, S]\!] \,\&\, \sum_{i=1}^{N} a_i \cdot R_i \rightarrow R_{N+1}$$

*In the formula, $a_i$ is the amount of resource $R_i$ required for acquiring the resource $R_{N+1}$.*

### 1.1.2 Base building

Base building is a general concept in which players build structures in order to (1) secure their position in certain areas, (2) gather, produce, and store resources, and (3) produce military units. Usually, the players concentrate their structures into one area to enhance the interaction between them (e.g. in case of a chain of resources), and to simplify their protection (be it by other structures or by units).

In many games, e.g., Age of Empires II (AoE) [17] and Stronghold, it is essential to erect defensive structures (castles, towers, walls, moats, etc.) around the base to defend it against opponent's attacks. On the other hand, in some games, defensive structures are limited to a small number of buildings (Original War [18]) or they are not present at all (Warcraft [19]).

Games with a complex resource management require many structures in the manufacturing chain. In Stronghold, bread production begins with a wheat farm and continues with mill and bakery before the bread lands in the granary. If the game features simpler resource management, then also less structures related to it are present: in Dune II, it is only the spice refinery.

In most RTS games, military units are produced in buildings (an exception are soldiers in Original War, which are given to the gamer in the beginning of the session). Either all units are created in one building (barracks in Stronghold) or different units are produced in different buildings (in AoE, where swordsmen and pikemen are created in barracks, archers and crossbowmen in archery range, scouts in stables, etc.).

### 1.1.3 War tactics

Military tactics is *"the science and art of organizing a military force, the techniques for using weapons or military units in combination for engaging and defeating an enemy in battle,"* as defined in the article "*Military tactics*" in Wikipedia [20]. This definition expresses the main idea of war tactics in RTS games. It requires the player to combine military units of different types to develop tactics which would be successful against the opponent's units and structures.

The player can not stop the time progress to plan the battle, to investigate the battlefield or to issue commands to units. Some games (e.g. Original War) even penalize players for reloading the game during battle. RTS games are usually fast-paced so that if the player is wasting time, opponents are likely to gain an advantage.

Each unit and structure has a certain amount of health. In a typical RTS game, interactions between different types of units and structures are defined as the damage (the loss of health) caused by one to another.

*We define set of all units and structures SU := {a ; a is Unit or a is Structure}, function damage: SU x SU → **N**, where damage($K_1$, $K_2$) is the amount of health unit $K_2$ looses if it is attacked by $K_1$, and function health: SU → **N**, which expresses the health a unit or a structure currently has.*

*After $K_2$ is attacked by $K_1$: health($K_2$) := health($K_2$) – damage($K_1$, $K_2$)*

*In order to simplify the formula, we presume that such $K_1$, $K_2$ may exist that damage($K_1$, $K_2$) = 0. (In a typical RTS game, there would be many such pairs, for example a wall attacking a soldier). The formula does not presume that the relation "$K_1$ isAttackedBy $K_2$" is symmetric [21].*

It is possible that a unit will be variously effective against different types of units (and the above formula allows such an option). For instance, in AoE, pikemen do more damage to mounted knights than to infantry (i.e. pikemen are very effective against mounted knights). Such information is important in planning the combat tactics.

In some RTS games, the properties of the battlefield play an important role; some areas on the battlefield favor or disadvantage the units and structures occupying them against units or structures occupying other areas. This includes for example an increased archers' range when they shoot from a higher point, the fact that in a forest, soldiers are less likely to be hit by an arrow (both in Stronghold) or a lower speed of vehicles while they go uphill (Original War). Not all RTS games consider the properties of the battlefield important (Warcraft and Dune II).

### 1.1.4 Developing new technologies

The player has usually the possibility to perform research for new technologies. After a new technology is developed, new structures, units or technologies are unlocked or current units and/or structures may be improved/upgraded.

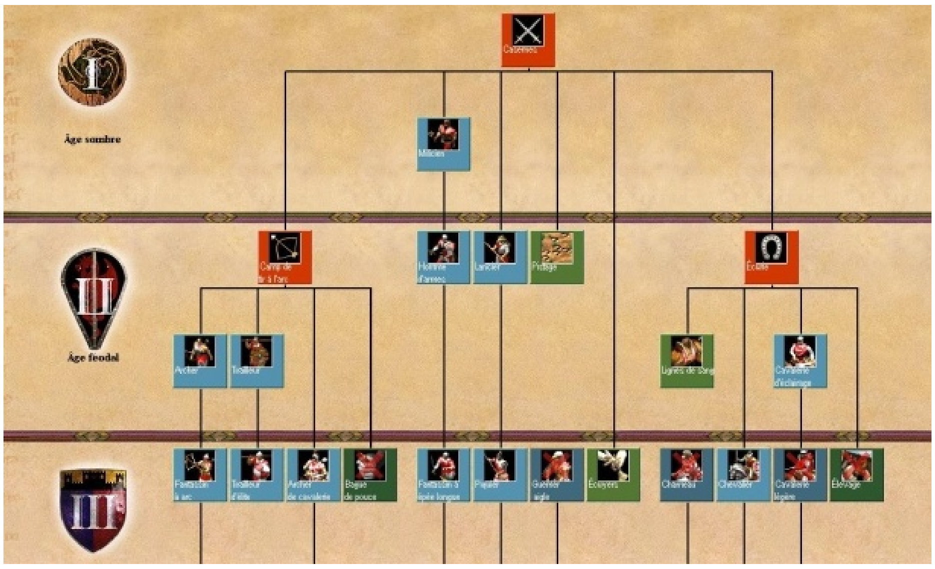


*Figure 1: A part of a technology tree from Age of Empires. Orange squares represent buildings, blue squares units and green squares technologies. Source: http://www.wsgf.org*

Upgrades are ordered into *technology trees*. According to the article Technology tree in Wikipedia [22], it is "*a hierarchical visual representation of the possible sequences of upgrades a player can take, by means of research. The diagram is tree-shaped.*" We can see an example of a technology tree from AoE in Figure 1. Orange squares represent buildings, blue squares units and green squares technologies.

In RTS games, taking one sequence of research does not exclude other sequences from also being researched. It is very usual that the player needs specific buildings and units to perform research (for example in Original War, research is done by scientists in laboratory; in AoE, several buildings such as university, blacksmith, castle, etc. are responsible for research).

Unlike the former 3 components (resources management, base building, war tactics), in-game technology development run by the player is not present in all RTS games; for instance Stronghold, Warcraft and Dune II do not contain it.

### 1.1.5 Our definition of RTS

Now that we have investigated components (c1) – (c4) of RTS games named by the article Real-time strategy, we should refine our definition of the RTS genre so as we understand it. From our experience with playing RTS games, components (c1) – (c3) are reasonably balanced in the most successful RTS games (this is, for instance, the case in AoE and Stronghold) so that the player spends some time with managing each of them. It is important to realize that the more time the player has to spend with one of the components, the less time they have left for others components. So, here is our definition:

*RTS is a fast-paced military strategy simulation making an impression of a continuous time progress. The participant takes the role of a supreme commander controlling units and structures to secure areas in the map or destroy their opponent's units and structures. New units and structures can usually be constructed during the course of the game. The game contains 3 main components, which are (c1) resource gathering and management, (c2) base building, and (c3) war tactics, where non of them is neglected in favor of other two. A 4th component, the development of new technologies (c4), may or may not be present.*

We must not forget the most important aspect about computer games in general: they are intended to entertain players!

## 1.2 Requirements

We have come up with a definition of RTS games as we understand them, and we feel that it is more specific than the definitions we were able to find in literature. We have also covered the 4 main components of RTS games. However, these components are still very general and we are yet to discuss how to implement them within our project. Furthermore, there is still a large number of important requirements on the game which have not yet been explained thoroughly enough. We can divide them into 2 main groups: (1) conceptual requirements, and (2) technical requirements.

(1) Conceptual requirements include decisions about the nature of the RTS game. According to our definition, the player controls units and structures (we will call them *game entities* in our project) and fights against their opponents. The game plays in a battlefield or in a world; we have yet to discuss the game map which represents it. We have to include resource management, elements of war tactics, and base building into our game. We believe that those three components should be in balance so that neither of them will be superior to another. We also declared that it is our intention to make the game illustrative and well comprehensible so that it is accessible to the community. For this reason, we want to keep the three components simple but with further possibility to be extended. For the same reason, we will give up the in-game technology development.

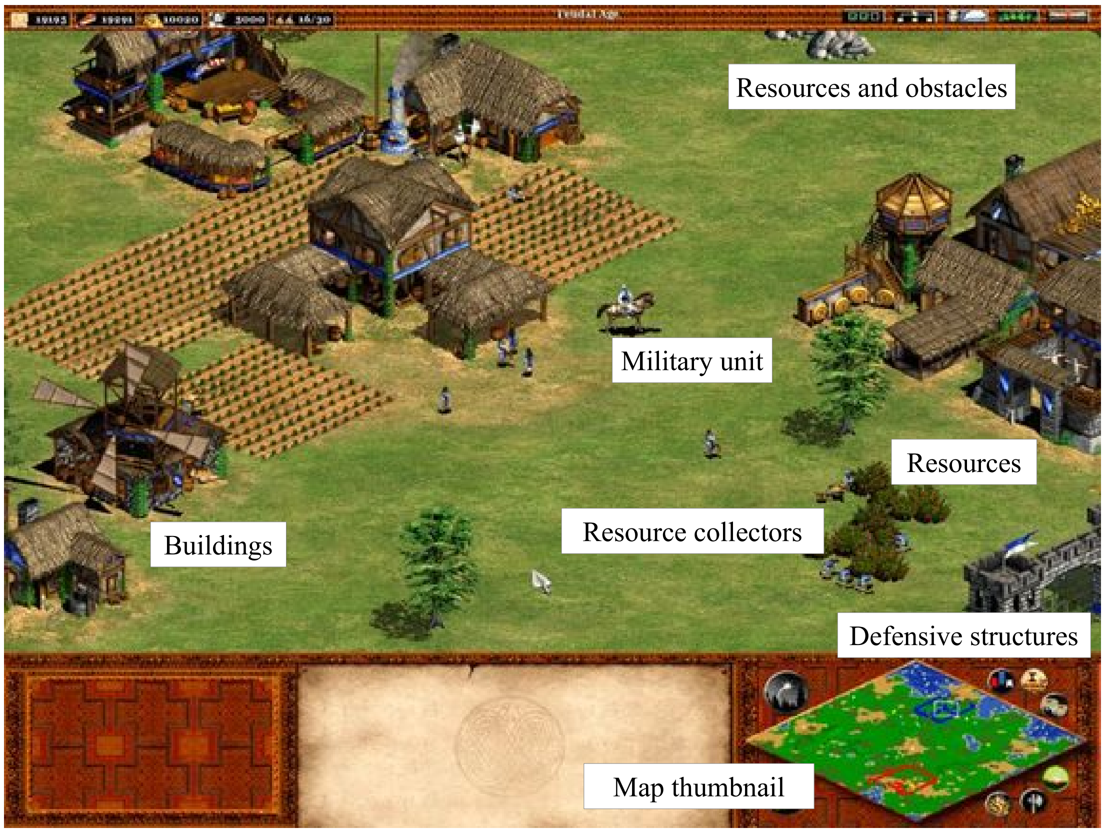


*Figure 2: A screenshot from Age of Empires II.*
*Source: http://en.wikipedia.org/wiki/File:Age_ii_feudal_age_celts.jpg*

(2) Concerning the technical requirements of the project, we have to discuss the extendability of the project (what parts of it should be extensible). Furthermore, the graphics of the game plays an important role, as it influences the experience players obtain from playing the game.

We will illustrate the specific parts using two screenshots from 2D RTS computer games. Figure 2 shows a screenshot from AoE, Figure 3 from Dune II.

### 1.2.1 Players and their opponents

According to the definition of RTS games, the participant plays against opponents. There are 2 possible types of opponents: (1) other human players and (2) computer players. If more human players than one participate in the game, it is called a *multiplayer* mode. In some game genres (such as car racing, for instance), human

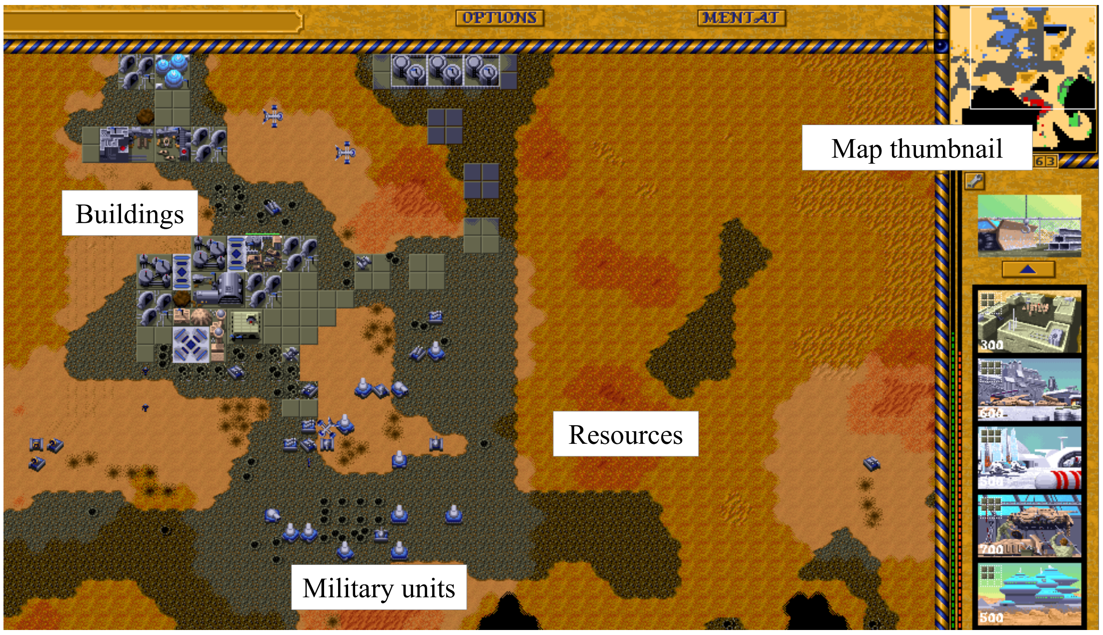


*Figure 3: A screenshot from the game Dune II.*
*Source: http://www.dosgamers.com/uploads/images/original/dune2c.gif*

players can share one computer screen (it is called *split screen*, the screen is divided into two parts). However, in RTS games, players very often try to keep some information (for example position of units, the amount of resources, etc.) hidden from each other for tactical reasons. Therefore, it is necessary to allow each player to have their own computer and connect them via network. The game session is then called *remote game session*. If there is only one human player competing against computer players, the mode is called *single player mode*.

Nowadays, the socializing element is getting more and more important (and entertaining) in RTS games so that our game certainly has to support remote game sessions. As we realize that network communication in general can be prone to mistakes which are difficult to debug, we will include a possibility of testing it on correctness of all information exchanged between communicating games.

Unlike humans, computer players do not have a brain, therefore they need to receive one from us – we will have to give them *artificial intelligence (AI)*. In many games, there are more different AIs available, each of them concentrating on a specific strategy. It simulates real human players, as different human players may choose different approach to different issues in the game. Furthermore, it is more entertaining for the gamers to play against computer players which do not all follow the same pattern of behavior (e.g. some AIs may prefer attacking more often with less units whereas other AIs might like to attack in longer intervals, but with more units at once). Therefore, we want to include the possibility of computer players following different strategies, and we would also like to enable the community to extend the game by inserting additional AI strategies.

Before the game begins, players should be able to set the number of computer players in the game and the strategies the computer players will follow.

### 1.2.2 Resource management in our game

In section 1.1.1, we showed that no matter the concept of resource management, the main point is always in using resources as a currency for obtaining structures, units and technologies. We also showed a chain of resource processing (i.e. resources are either collected or created from other resources). Examples of resource management are shown in both Figure 2 and Figure 3. Let us now have a look at options we have encountered across various RTS games:

- Resources appear during the course of the game and they disappear after being exhausted (collected). For example in Stronghold, trees (provide wood) grow naturally; they are cut down by a woodcutter. In Original war, crates literally appear (they come from the future via time machine) in the game in certain areas and they are collected by workers.
- Resources are given from the beginning and their number may decrease while they are being harvested (collected). For instance, in AoE, stone is present in the map from the beginning and the amount of it lowers while it is being mined by workers. However, its amount does not increase during the course of the game.
- Resources are produced in a structure (sometimes, the structure has to be placed to a certain area), with or without input of any other resource. For instance in Stronghold, wheat grows in farms without any input, but the farm needs to be placed on fertile soil. On the other hand, to produce bread in a bakery, flour is needed (it is an input resource).

One of the main goals is the extendability of our project. For this reason, we want to support the community in creating the resource management according to their needs and we do not want to be restrictive. However, the basic implementation should not contain too many resource types and resource-processing structures for the sake of the illustrative nature of the game. The community has to be able to extend the game by new resource types and game entities related to resource management:

*There has to be a few resource types in our game with the support for both growing and non-growing resources. Furthermore, the game has to contain structures processing the resources brought to them and transforming them into other resources. Players will control special units: resource collectors will collect resources and bring them to the processing structures, and resource transporters will transport resources between structures.*

### 1.2.3 Game entities in our project

Game entities play an important role in all the three main components (c1) – (c3) (apart from resources management, it is war tactics and base building). Examples of various game entities (resource collectors, military units, defensive and other structures) can be seen in Figure 2 and Figure 3.

In various RTS games, there are 3 possibilities where military units are created:

- All types are created in one building (in Stronghold, it is in barracks).
- Different types are created in different buildings (in AoE, soldiers are created in barracks, scouts in stables, archers in archery range, etc.).
- In Original War, the soldiers are not created at all, they are given to the player in the beginning of the game. However, this approach is merely an exception.
- There are also 3 options how structures are created:
- In other structures: for example in Dune II, all structures are created in Construction Yard.
- By a special type of units (in AoE and Original War, there are workers/builders who build structures).
- Without any special structure or unit (in Stronghold). The players choose which structure they want to build from a list in the game and place it to a chosen area.

In some RTS games, there is a special building which can not be created – it is given to the player in the beginning of the game. For example in Dune II, it is the Construction Yard.

The goal of our project states that we want to keep both the war tactics, i.e. the management of military units, and the base building, which is building of structures, easy to comprehend. For this reason, only a few military units and structures may be available in the basic implementation, and we do not want to define complex interactions between them. On the other hand, we want to enable the community to extend the program by additional game entities. According to all the facts listed, the system of game entities has to be as follows:

*In our game, we want to allow different types of units to be created in different structures. Structures should be created in other structures. There may also be structures which will be given to the player in the beginning of the game and which will not be constructed in any other structure (we do not forbid this option). We have to support future extensibility of the game by new game entities.*

### 1.2.4 Game map

The game map represents the world where the game is played. Generally, the map might be of any shape, but in RTS games, it is typically a rectangle. In some games, the map is subdivided into layers; some of them might be inaccessible for some game entities (in AoE, there are two layers – water and ground, and e.g., a ship cannot step on ground or an archer can not walk on water). For the sake of the easy comprehensibility of our game, it will not contain different layers. There will only be one layer in the game map which will be accessible for all units.

In many RTS games (AoE, Stronghold, Original War), there are various terrain profiles such as hills, mountains or lowlands. On the contrary, in Warcraft and Dune II, no terrain profile is to be found. Among other things, terrain profile influences the war tactics. For the simplicity of our game, it is our choice not to use terrain profile, which will also make the war tactics (and units management) easier.

On the other hand, the game has to contain obstacles, i.e. areas of the map which are inaccessible to units and where structures can not be built. We see examples of obstacles in Figure 3. Both Figure 2 and Figure 3 show us also a map thumbnail, which improves human player's orientation in the map. In order to make the game more interesting, we want to enable the community to tailor it to their preferences by adding new maps.

### 1.2.5 Extendability

There are 2 major concepts of extendability: (1) design extendability, and (2) run-time extendability.

(1) Design extendability means that the system is designed for future growth. We intend to use patterns from object-oriented programming to make the game extendable by design. After design extensions have been added, the program typically needs to be recompiled. Since the game is an open-source project which aims to be as illustrative as possible, a well maintainable and extendable design is very important.

(2) Run-time extendability does not need recompilation of the program; new extensions are included at run time. We already declared our intention to allow the community to extend the game at run time by the following:

- Strategies for computer players
- Game entities
- Game maps

To support the community and to make the task of creating new extensions easier, we would like to prepare sample extensions on which we will demonstrate how to make such an extension.

### 1.2.6 Graphics

Authors of RTS games choose from 2 basic possibilities: (1) 2D graphics and (2) 3D graphics. For the sake of simplicity, we will use 2D, because for 3D, we would need 3D models of each game entity, which would also apply for each of its extensions. This would in turn limit the extensibility of our program as making a 3D model is much more demanding than making a 2D picture, so that anybody who would like to extend the game by adding a new game entity would have to be skilled in 3D modeling, too.

Considering 2D graphics, 2 approaches to its projection can be chosen: (1) true 2D projection (called *bird's-eye view* or *top-down projection)* and (2) a simulated 3D projection.

(1) In bird's-eye view, the virtual camera is placed above the game plan, and it shows the game as a true 2D scene. In Dune II (Figure 3) this projection is used.

(2) Simulated 3D projection (used in AoE shown in Figure 2) is sometimes called 2.5D. The article 2.5D in Wikipedia [23] describes it as graphical projection *"using some form of parallel projection, wherein the point of view is from a fixed perspective, but also reveals multiple facets of an object."* In parallel projections, lines which are parallel in reality are also parallel in the projection. In Figure 2, axonometric projection [24] is used.

For this projection, more pictures for one game entity have to be provided (because the game entities may be seen from more camera positions than just one), which would again limit the extensibility of our project. As we want to keep the game easy to comprehend and illustrative, we will use the 2D top-down graphical projection, where for each unit, only one facet is revealed (the upper one) so that only one picture is sufficient.

## 1.3 Project goals

Now that we have had a look at all the important parts of our game, let us summarize the goals of our project.

***The main goal of this thesis is to design and develop an extendable open source, bird's-eye-projected 2D RTS computer game, which would meet our definition of RTS. For the development, we will use the .NET Framework.***

1. Conceptual goals:
    - 1.2. The game has to be illustrative and easy to comprehend, so that the community will be able to tailor it easily to their own needs.
    - 1.3. The game has to support multiplayer mode (human players connected via network), and single player mode (one human player competing against the computer). Players have to be able to set the number of computer players in the game (for both single player mode and multiplayer mode).
    - 1.4. The game has to contain more than one strategy for computer players. Players should be able to choose the strategies for AI opponents.
    - 1.5. Resource management, elements of war tactics, and base building have to be in balance so that neither of them is superior to another. It is our intention to make the game illustrative and well comprehensible so that it is accessible to the community. For this reason, we have to keep the three components simple but with further possibility to be extended.
    - 1.6. The concept of resource management in the game has to be as follows:
        - There has to be a few resource types in our game with the support for both growing and non-growing resources.
        - The game has to contain structures which will process the resources brought to them and transform them into different resources.
        - Players will control special units: resource collectors will collect resources and bring them to the processing structures, and resource transporters will transport resources between structures.
    - 1.7. The concept of game entities in the game has to be as follows:
        - The basic implementation will contain only a few game entities, but the system of game entities will be extendable at run time so that the community will be able to extend it by additional game entities easily.
        - We allow different types of units to be created in different structures.
        - Structures will be created in other structures.
        - There may (but does not have to) be structures which will be given to the player in the beginning of the game and which will not be constructed in any other structure.
        - We do not require complex interactions between game entities.

1.8. The game map will be rectangular and it will comprise one layer accessible to all units. For the simplicity of our game, we will not use terrain profile (the terrain will be a plane).

1.9. The game map has to contain obstacles, i.e. areas of the map which are inaccessible to units and where structures can not be situated.

1.10. The game has to contain a map thumbnail, which improves human player's orientation in the map.

2. Technical goals:

   2.2. The design of the game has to allow simple extensibility of the game by the community.

   2.3. There has to be support for following run-time extensions in the game:
   - Strategies for computer players
   - Game entities
   - Maps

   2.4. In order to support the community, we have to prepare sample extensions and explain, how to make new extensions.

   2.5. Network communication in general can be prone to mistakes which are difficult to debug; for this reason, the game has to contain a possibility of testing the correctness of information exchanged between communicating games.

# 2 Problem analysis

In chapter 1, we described all the important components the game has to contain and we set the goals of our project. Let us now analyze different possibilities of implementing the game components.

## 2.1 Continuous time progress

In our definition (see section 1.1.5), RTS game makes an impression of a continuous time progress. However, based on properties of graphics cards, only separate pictures can be displayed on screen. We have to display a reasonably high number of pictures (for example 25) per second so that the human eye perceives the pictures as a movie and not as separate pictures. This implies a subdivision of the game time progress into rounds; in each round, the state of the game has to be updated and drawn on the screen. In RTS computer games, these 2 activities are very often represented by 2 subroutines: (1) Draw and (2) Update.

- **Draw** displays the current state of the game. It does not know how many times the state has changed since the last time it was displayed. Draw will be able to display the game even without any updates.
- **Update** is responsible for processing one step in the game logic. It will only care for computing the next state of the game.

In other words, Draw receives data and renders graphical output, while Update provides the data.

## 2.2 Graphics libraries

According to the main goal of our project, we are developing a 2D game. We do not need to use vector graphics as we do not expect the textures used in the game to be resized; bitmaps are sufficient. Therefore, we need a graphics library for loading, creating, and displaying 2D bitmaps.

In RTS games, game entities belonging to one player are denoted by his or her color. Figure 4 (A) and (B) shows us the same game entity with colors of two different players. To achieve this, we use two different textures for each game entity, shown in Figure 4 (C) and (D), where (C) is used as the base texture and (D) as a mask, which is covered with the color of the player. Here, magenta color represents transparent color.

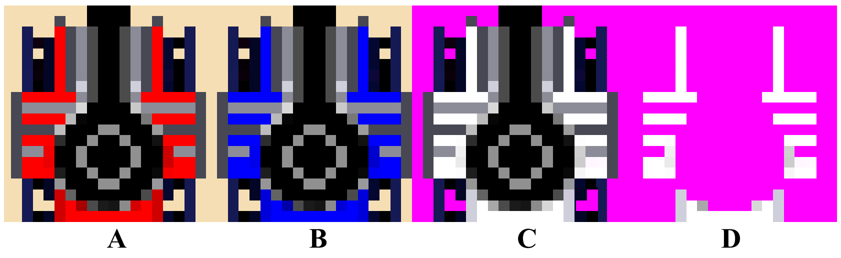


*Figure 4: Composition of a tank. From left to right: Two tanks, (A) first belongs to the red player, the other (B) to the blue player. (C): Image defining the body of the tanks. (D): Image defining the color mask. Note that the magenta color represents transparent color.*

In the game, it will be important to use some transparent colors, either *truly transparent* (using the alpha channel in RGBA [25]) or a *default transparent* color (like magenta color in Figure 4). Both of them are properties of the class `System.Drawing.Bitmap` [26], which is a part of the .NET Framework.

We also need easy rotation of the bitmaps. We will have many moving entities in our game and we want them rotated in the moving direction. For this, the bitmap will have to be rotated with respect to the direction vector of the moving entity. Also, proper documentation of the graphics library plays an important role for us: it makes our development easier and provides valuable support for the community.

Let's summarize our requirements:

- We only need 2D bitmap-based graphics, no vector graphics and no 3D graphics is required (and will not be discussed).
- Create, load, and display bitmaps.
- Rotate bitmaps.
- Proper documentation is important.

We will briefly describe the most popular graphics libraries available for .NET and learn if they meet our requirements; then, we will decide which of them to use. We choose from the following graphics libraries:

- Microsoft XNA [27]
- The Open Toolkit Library (OpenTK) [28]
- SlimDX [29]
- SharpDX [30]

### 2.2.1 Microsoft XNA

The Microsoft XNA Framework, built on top of DirectX [31], enables the game to run under various platforms such as Microsoft Windows, Windows Phone 7 and Xbox 360. Applications created using XNA are portable to various platforms (Linux, Mac OS, iOS, Android) using the open source project MonoGame [32], which is based on OpenGL [33].

XNA provides a class called `Texture2D`, which wraps 2D textures. It also provides a simple way of rotating textures just at the moment they are being drawn by specifying the angle or rotation to the axis. Apart from being able to load pictures of various formats from stream using the `Texture2D.FromStream` method, XNA provides a component called `ContentManager`. It loads images and other content (e.g. fonts, sounds, etc.) from files produced by the design time content pipeline. However, the content has to be inserted into the solution before compilation.

Furthermore, XNA offers a class `Game`, providing methods `Update` and `Draw`, which do the jobs we described as Update and Draw. Another important point of XNA is the large number of examples available on the internet in different forums and blog articles; also the documentation homepage serves as a valuable source of information.

### 2.2.2 The Open Toolkit Library (OpenTK)

The OpenTK is a library allowing .NET and Mono programs to use the OpenGL graphics library. Handling textures mainly relies on the `Bitmap` class to load data. Although the approach is not as straightforward as loading and storing textures in XNA, it is still simple and after adding a few functions, very convenient. Rotating a texture is slightly more complicated than within XNA; in OpenTK, the OpenGL rotation based on rotation matrices has to be used.

The OpenTK also provides a `Game` class with 2 basic methods: `OnUpdateFrame`, and `OnRenderFrame`, dealing with rendering frames (Draw) and updating frames (Update).

At the time we were considering graphics libraries, the documentation at the homepage of the OpenTK project was weaker than the one of XNA. Certainly, one option is using the OpenGL documentation, but the methods very often have slightly different interfaces. On the other hand, OpenTK provides many examples and tutorials with the installation packet; users can take them and play with them just after installing OpenTK.

### 2.2.3 SlimDX

SlimDX is an open source project wrapping DirectX (very much like XNA). It provides class `Texture2D` for storing textures. Rotating a 2D texture works exactly the same way as in XNA, which means that we only have to specify the rotation angle while the texture is being drawn.

When investigating the documentation, we immediately notice that there is a detailed class reference, much like within the XNA. However, there is no in-depth documentation. On the other hand, according to the authors of SlimDX, the concept itself is very similar to XNA so that XNA and DirectX tutorials should be useful for SlimDX as well. SlimDX also provides the `Game` class (much like XNA and OpenTK) comprising methods for game logic update (called `Tick)` and for drawing the game (called `DrawGame`).

However, there is no support for Linux-based operating systems within SlimDX; also no content manager is present so that only loading bitmaps using `Texture2D.FromStream` is possible.

### 2.2.4 SharpDX

SharpDX is also an open source project encapsulating DirectX. It fully supports all Windows platforms including Windows Phone 8.

SharpDX also provides `Texture2D` class which wraps textures. There are also methods for loading textures directly from a BMP file such as `TextureLoader.LoadBitmap`. The texture can also be rotated while being drawn just by specifying the rotation angle; this is similar to SlimDX and XNA.

SharpDX also features a documentation on approximately the same level as SlimDX, but compared to XNA, it is rather limited. One of the main issues it the lack of code examples. However, the method interfaces are similar to XNA and SlimDX so that an XNA example should also apply for SharpDX.

SharpDX also provides the `Game` class with properties similar the SlimDX and XNA `Game` class (`Update`, `Draw`). It contains the `ContentManager` with the same function as in XNA.

### 2.2.5 Summary

We can say that all the graphics libraries fulfill our requirements, and even though there are differences between them, we can imagine building our application on the top of any of them. We summarize what we learned in Table 1.

| | **XNA** | **OpenTK** | **SlimDX** | **SharpDX** |
|---|---|---|---|---|
| **Create bitmaps** | ✓ | ✓ | ✓ | ✓ |
| **Load bitmaps** | ✓ | ✓ | ✓ | ✓ |
| **`ContentManager`** | ✓ | - | - | ✓ |
| **Easy rotation of textures** | ✓ | Matrix | ✓ | ✓ |
| **Documentation** | ✓ | Limited | Limited | Limited |
| **Underlying technology** | DirectX / OpenGL * | OpenGL | DirectX | DirectX |

*Table 1: Summary of how the graphics libraries fulfill our requirements.*
** XNA uses DirectX as underlying technology under Windows, and OpenGL under Linux (MonoGame).*

Also, for the sake of of extendability of the project, it would be beneficial if the game graphics would work under various platforms. We show the summary of the graphics libraries we investigated in Table 2.

| | **XNA** | **OpenTK** | **SlimDX** | **SharpDX** |
|---|---|---|---|---|
| **Windows** | ✓ | ✓ | ✓ | ✓ |
| **Linux** | ✓ | ✓ | - | - |
| **Windows Phone 7** | ✓ | - | - | - |
| **Mac OS** | ✓ | ✓ | - | - |
| **Xbox 360** | ✓ | - | - | - |
| **iOS** | ✓ | ✓ | - | - |
| **Android** | ✓ | ✓ | - | - |

*Table 2: Supported platforms*

First of all, we exclude OpenTK and SlimDX from the pool. The reason is the absence of the `ContentManager`, which might be a very useful component for example for loading fonts or music effects used in the game.

Then, XNA and SharpDX seem to be equivalent choices, but we also have to consider that there is much more support for XNA such as a large number of examples available online (forums, blogs, etc). XNA also has documentation on a higher level than SharpDX. Furthermore, Table 2 shows that XNA supports more platforms, which may be also advantageous for the extendability of our project. For this reasons, we choose the Microsoft XNA to be the graphics library used in our project.

## 2.3 Artificial intelligence (AI) for computer players

According to goal 1.4 of the thesis, the game has to contain more than one strategy for computer player. AI for computer players might be one of the key factors in the experience the user (i.e. gamer) receives from playing a computer game. According to Buckland [34], there are 2 different approaches to implementing the AI: (1) searching for the optimal solution and (2) designing suboptimal but entertaining AI.

**Searching for the optimal solution:** The main goal would be to design AI always able to find the optimal solution to a given problem, despite to time and hardware limitations. From a mathematical point of view, this might seem the right approach, and also, in some games (chess, for instance), it is. On the other hand, in RTS games, the reality is different due to the amount of resources (e.g. CPU properties) varying from machine to machine, and to the complexity of the game.

Furthermore, the gamer wants to be entertained and to have a fair chance to win the game, whereas competing against a player, which knows everything and always wins (because it knows the winning strategy, if there is such one) would be frustrating for the gamer.

**Designing suboptimal but entertaining AI:** Buckland [34] (page xx) says: "*If the player believes the agent he's playing against is intelligent, then it is intelligent. It's that simple. Our goal is to design agents that provide the illusion of intelligence, nothing more.*"

Here, a very important idea of the AI design is mentioned: The aim is to entertain the human player, not to find the always-winning strategy. The AI will not be judged on the quality of the solution; it will be judged on the ability to produce fun for the human player as he or she has to believe that the enemy is intelligent, which, incidentally, also includes making wrong decisions and mistakes. In contrast to the optimal solution, this approach also saves system resources thus being more suitable for RTS computer games.

### 2.3.1 Cheating AI

In order to enhance the impression of computer player's intelligence without increasing the cost of development or complexity, it is possible to allow the AI to cheat, which means that the computer player receives an advantage unavailable to the human player under the same conditions. However, human players must not loose their trust in the AI, which means that cheating strategies should not produce unrealistic behavior. There are some ideas on what could cause the human player loosing their thrust in the AI:

- Human player stands next to an enemy's building, which is being created at the moment and he or she sees that the construction runs very quickly (the building is finished almost immediately) compared to his or her own buildings whose construction takes much more time.
- Human player notices that if the computer player's unit is shot, the damage is

lower than the damage made to his or her own unit when shot by the same type of a game entity.

However, there are also some cheating strategies the human player will not notice; such strategies are often used in computer games. Two examples:

- Computer player may have more information than human players. For instance, it might know the position of all game entities in the game so that it can better prepare its defense.
- Computer players might enter the game with a higher amount of resources than human players. Due to it, they might seem stronger in the beginning; in this case, human players might even believe that the AI is more intelligent than it really is. This would give them a better feeling after defeating such an AI player. On this example, we see that a cheating AI might even enhance the experience a player obtains from playing the game.

To conclude the discussion, let's summarize the observations important for the design of our game:

- We aim to create an entertaining AI which does not have to provide an optimal solution. However, the human player should believe that he or she is up to an intelligent player.
- The AI will be allowed to cheat as long as it does not influence the human player's experience in a negative way.

### 2.3.2 Example of an AI: Erik, The Warrior

In order to illustrate the AI which may be used in our project, let us first think about a simple AI strategy. It is an easy strategy (to be illustrative) and it reflects our experience with computer players' strategies from various RTS games. The strategies very often have a defensive component and an offensive component, which define the player's behavior towards defending and attacking. For the sake of illustrative nature of the strategy we have chosen, we give up the defensive component of the computer player. We will call the player "Erik, The Warrior":

*Erik enters the game with a rather high amount of resources. He creates some (but not many) resource collectors and orders them to collect resources. Then, Erik constructs two or three buildings in which attack units are produced before producing a high number of attack units. At the end of the tenth minute (to give the human player enough time to prepare for defense), Erik sends its attack units to engage the human player. Then, every 3 minutes, a new attack is sent if the computer player has at least a medium number of attack units. Otherwise, more attack units are produced. If Erik does not have enough resources, he produces more resource collectors.*

Apparently, Erik does not poses an optimal AI (if he loses all his units while attacking, he would be defenseless and hence prone to being destroyed very quickly). However, the human player might have some hard time (and fun) fighting off his attacks and then preparing his or her own units to attack him. Here, we considered that Erik is in possession of a cheating AI, because he starts with "*a rather high amount of resources*". However, the same AI could work without cheating. It would only take Erik more time to collect enough resources to rear the desired amount of attack units. Let us now decompose Erik's behavior as if we were to make simple rules he has to follow:

- *If you do not have enough resource collectors, then build some.*
- *If you have less than 2 buildings where attack units are created, then build one more such building.*
- *If you have less than a high number of attack units, then create some.*
- *If at least 10 minutes have elapsed, attack the human player.*
- *If at least 3 minutes have elapsed since the last attack and you have less than a medium number of attack units, build some.*
- *If at least 3 minutes have elapsed since the last attack and you have at least a medium number of attack units, then attack the human player.*

In each Update, Erik acts according to the rules we gave him. Let us now consider what the rules actually mean. Then, we will investigate the approaches for implementing a system consisting of such rules.

### 2.3.3 Rules for decision-making

*"If you have less than a high number of attack units, then create some,"* we told Erik. But what is a high number of attack units? A very straightforward idea could be: "It is between 15 and 20". According to it, Erik would stop producing attack units if he had 15 of them or more. We could rewrite the rule in order that it would look like: "*If you have less than 15 attack units, then create some."* Even though this approach is often used in RTS games, it is not the only one possible. Some games use the concept of **Fuzzy logic.**

Fuzzy logic (also sometimes called many-valued logic or probabilistic logic), unlike classical logic with crisp sets, does not rely on absolute truth values. It introduces a *degree of membership* of an element into a set, which can be any number between 0 and 1 so that a player may have a *rather high* number of resources – let's say that the number of resources is 0.4 medium and 0.6 high, whereas in classical logic, the number of resources could only be either medium, or high. In Figure 5, we see an example of a fuzzy set, which represents the number of resources a player has. Between 0 and 3000, the number is considered to be low, it is medium between 1500 and 8000, and if the number is greater than 5000, it is considered to be high.

Due to its complexity, fuzzy logic might be a suitable option for AI to provide the impression of sensitive human-like decision making with ambiguous or vague input data such as "*If you have a great deal of resources and the enemy is at a rather medium distance, then build more attack units*" because this exactly is the way human players think. However, the complexity is one of its most significant drawbacks at the same time.

If we implemented fuzzy logic, we would obtain a very complicated system of AI which would simulate human decision-making, but on the other hand, the resulting gain for the quality of the AI would be very small compared to the complex implementation. This is mainly due to our intention to apply a simple, suboptimal AI which would mainly entertain the human player. Furthermore, goal 1.2 states that the game has to be simple and illustrative so that it is easy to comprehend by the community. On the other hand, we want to allow the community to extend the game by design by fuzzy logic.

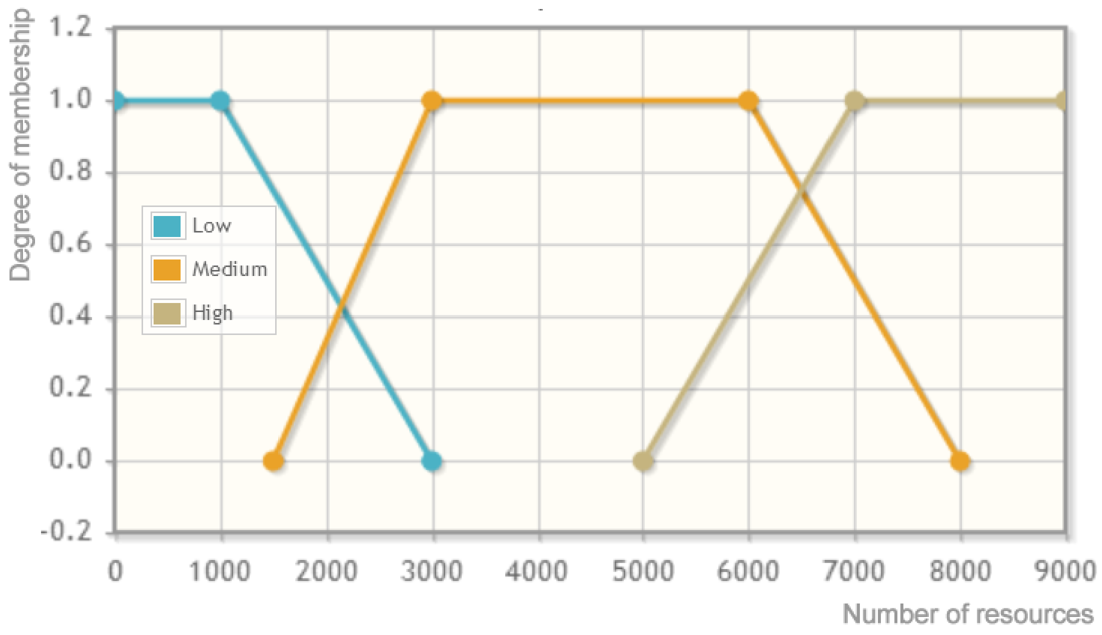


*Figure 5: The plot of a fuzzy set for the number of resources a player has. The plot was created using the free online tool "Fuzzy Logic Online Tool" available under the following link:* http://etools.elasticbeanstalk.com/faces/fuzzy.xhtml *on 11th of April, 2013.*

As a result, we decided not to use fuzzy logic, so that for instance the representation of the question "*Does the player have enough resources to build an army*" will be: "*is number of resources greater than N*" where N is a constant given by the AI designer.

For more information about Fuzzy logic, visit Fuzzy logic at Wikipedia [35] or read Buckland [34], chapter 10 (page 415).

### 2.3.4 Implementation

Now that we know how to understand the rules we gave to Erik, The Warrior, we should find an appropriate way to implement them. In RTS games, 2 approaches are used very often: (1) scripting languages [36] or (2) finite state machines [37] (FSM).

#### 2.3.4.1 Scripting language

A scripting language is a lightweight programming language intended to be easy to learn and to write programs in. To name but two examples, Lua [38] and Python [39] are used in scripting AI. Use of a scripting language brings some considerable advantages:

- It does not require AI designers to be skilled programmers.
- Scripts allow large changes to the logic without having to recompile the program as they are loaded on the fly.
- Scripts can be written in any text editor.

On the other hand, there are also disadvantages of scripting languages. First, scripting languages assume the presence of useful components written in the main programming language. Without this background library, scripting might be very inflexible. Also, scripts are very difficult to introspect and to debug, and the

```
FUNCTION AI_warrior ( player, timer )

** if the player does not have enough resource collectors, then create one
IF player.has_less_RC_than( 5 ) AND player.has_Resources_for_RC() THEN
player.create_new_RC()

** if the player has less than 2 buildings where attack units
** are created, then erect one more such building
IF player.has_less_buildings_than( ATTACK, 2 ) AND
player.has_resources_for_building( ATTACK ) THEN
player.create_new_building( ATTACK )

** if at least 10 minutes have elapsed and you have a high number of
** attack units, attack the human player.
IF player.has_more_attack_units_than( 10 ) AND
timer.elapsed_seconds( 60 * 10 ) THEN
player.attack( HUMAN )

END FUNCTION
```

*Figure 6: Some rules we gave to the Warrior AI written as a script*

maintainability of large scripts is limited.

Figure 6 shows us some rules for Erik's behavior. The function `AI_warrior` receives the player and a timer as arguments and it defines the player's actions. The script is very short, but if we wanted to define more complex behavior, it would get much less comprehensible.

#### 2.3.4.2 Finite state machine

An FSM is an abstract computational model for decision-making in a computer.

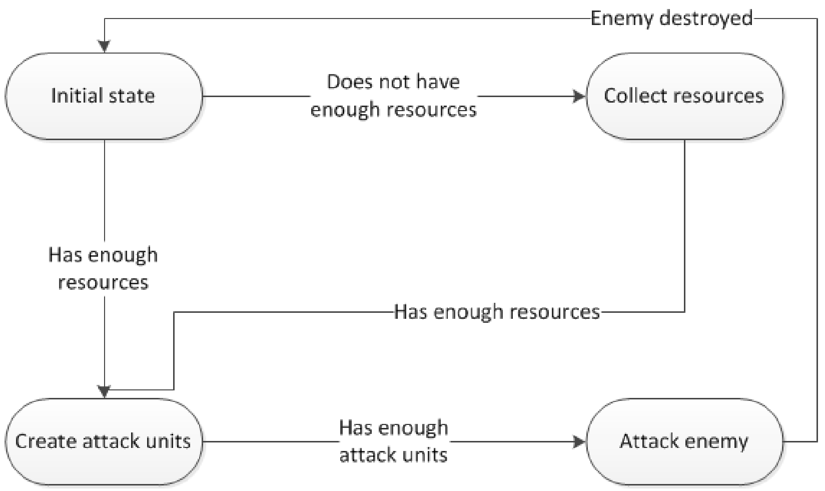


*Figure 7: A simplified example of an FSM.*

At any time given, the state machine is in one of a finite number of states. To change the state, the machine follows a transition. This happens if certain conditions are met. A sample FSM with four states and transitions between them can be found in Figure 7. We simplified some of the rules Erik received from us.

For instance, if Erik is in the initial state and he has enough resources, the FSM moves to the "Create attack units" state. Similarly, if it is in the "Create attack units" state and Erik already has enough attack units, then he can attack the human player; the FSM moves into the "Attack enemy" state.

Considering whether to use a scripting language or the FSM, we have to look at the rules we set up for Erik and in Figure 6 again. We realize that they, in fact, represent an FSM; each condition can be rewritten into a transition leading to a state of an FSM, just as we actually did in Figure 7.

To summarize our decision, the artificial intelligence in our game will be using a finite state machine; its transitions will contain conditions from classical logic.

## 2.4 Game entities as autonomous agents

In RTS games, the player controls game entities, makes decisions and issues commands to them. Then, it is a task of the game entity to fulfill the commands. For example, the player issues a command: "*Move to point P*" and he expects the entity to "do its best" to reach the point. Doing its best might mean finding a path or asking for path, evading dynamic obstacles or deciding what to do if the target turns out to be unreachable.

Gamers usually control a large number of game entities (in AoE, for example, it is up to 200 units and tens of buildings) and if they were to decide about all situations in the game, the gameplay would be very limited because the players would spend too much time with micromanagement of insignificant situations instead of concentrating on important strategy-related decisions. Therefore, the entities have to behave autonomously (to a certain degree). Here are some more examples of what the game entities have to be able to decide themselves:

- A unit is idle while attacked by an enemy unit. Should it stay idle, fight back or even pursue the attacker? For example in Warcraft, the unit does not defend itself and it is up to the player to issue the command. In Original War, units fight back if the attacking unit is within their reach.
- A resource collector (RC) collects resources. When its capacity is exhausted, it brings the resources to a building where they are stored. The player expects the RC to return to the place where it was collecting the resources to continue. After the source is consumed, the collector can start to search for other sources in its close surroundings.

The examples bring us to the idea that game entities need to have their own *internal artificial intelligence* which makes them behave like *autonomous agents*.

According to Buckland [34], "*An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*"

Game entities in our game will receive input in 2 different ways: (1) from the player (commands) and (2) from the environment (what they "sense"). Commands from the player give them the “agenda” (example of a command: *collect resources in a certain area*) and the entity then behaves autonomously and acts in pursuit of the agenda (collects resources, stores them in a certain structure and returns back for more resources). Let us now split the agenda of a resource collector into some very simple rules:

- *If you are told to collect resources in a certain area, go to the area and collect them.*
- *If your capacity is full, remember the position of the resources, return to the base and store the resources in a structure intended for this purpose.*
- *If you have a position of resources in memory, go to this position and continue collecting.*
- *If the resources have run out, search in the close surroundings. If you find other resources, go there and collect them. If you do not, stay idle.*

We notice that these rules lead to an FSM similar to the one we described in section 2.3.4.2. For this reason, we will be using the FSM not only for the AI of computer players, but also for the autonomous behavior of game entities.

### 2.4.1 The degree of independence

We can simplify the artificial intelligence for computer players while decreasing the amount of information (i.e. the number of commands) exchanged between the computer player and its game entities if we make the entities more independent. Imagine the following situation: *A resource collector finished collecting resources and is staying idle. If the player finds new resources in the map, they send the collector unit to the place to collect resources again.*

The (human) player has to issue a command for the resource collector to start collecting again (give it a new agenda). Here, we can make the task easier for the computer player's AI while making the (already autonomous) collector more autonomous: we give him one additional rule: *If you are idle, search for resources in a much larger area than just in the closest surroundings*. The computer player's AI will not have to take care about the collector, which will independently wander through the map, collect resources and bring them to the player's base. This approach is very advantageous because it allows subdivision of the decision-making between computer player and its game entities.

The only condition is that at the time computer player creates an entity, it is able to decide which from the list of available strategies will be assigned to the entity. The computer player will still be able to issue commands to the entity, but it will not have to do so as often as the human player.

## 2.5 Game map

Another problem to consider is the implementation of the map in our game. According to goals 1.8 and 1.9, it has to be rectangular and it has to contain one layer only; there will be no terrain profile. Let's first of all specify our requirements on it.

We already know that the game contains game entities and obstacles, which will be placed in the map. Using the map, the game has to be able to recognize accessible and inaccessible areas so that units are able to plan their path through the map (i.e.

change their position in the map while evading inaccessible areas). The process of planning the path through the map is called *pathfinding*.

Typically (and not only in RTS games), pathfinding uses graph algorithms, and the graph representing the map narrows the space which has to be searched. Such a graph is called a *navigation graph* (see an example in Figure 9); it is a weighted graph whose vertices stand for significant points in the map, and whose edges (weighted with the amount of time needed to pass the edge) represent free paths between these points. The navigation graph can be directed (e.g., if it has to represent one-way streets in a traffic network). The task of pathfinding is then to find the *shortest path*, which is a path on which the unit spends the least time possible.

There are 2 general ways of constructing a navigation graph: (1) either, it is dynamically created from the current state of the map, (2) or the navigation graph is static and it does not change in time.

A disadvantage of the static navigation graph is that it can not reflect positions of entities (they change in time), which causes the pathfinding to be less effective or

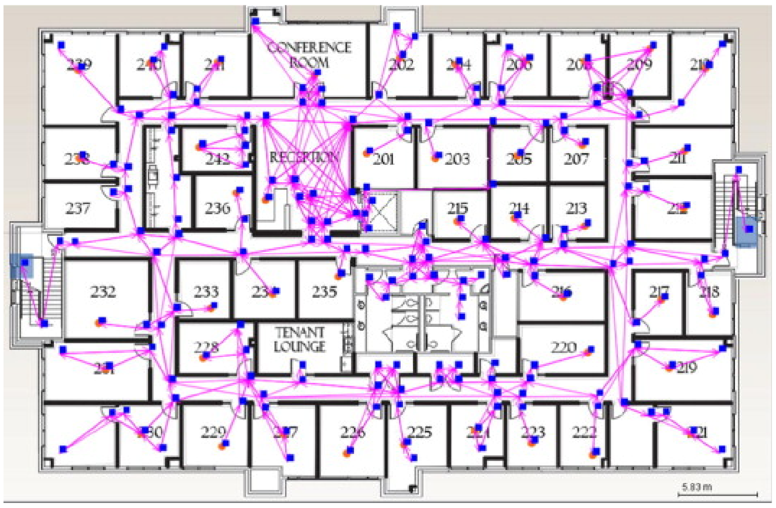


*Figure 8: An example of a navigation graph for pathfinding.*
*Source:* http://www.sciencedirect.com/science/article/pii/S1474034612000365

even not to work properly. For this reason, we want to be able to dynamically create the navigation graph from the map.

Let us now think about the nature of the map while considering the requirements.

### 2.5.1 Map without subdivision

The most simple possibility is a map without any primary subdivision. We have to consider how we find out whether an area in the map is free: to simplify the problem, we assume that any area in the map can be thought of as a convex polygon (in case of a different shape, we consider the convex hull of the shape).

We have to compute the polygon intersection of the given area with areas of all entities and obstacles within a certain range. The complexity of the algorithm equals *O(N)* where *N* is the number of all entities and obstacles combined, because all of them have to be taken into consideration by the algorithm.

**Dynamic construction of navigation graph**

An example of a typical dynamic construction of a navigation graph from a map without any subdivision is shown in Figure 9. As we can see, the vertices of the graph are placed in corners of the polygons, and edges connect each pair of vertices between which there is a free path, so that if there are *N* vertices, $N^2$ computations have to be performed (intersections of a segment with all polygons).

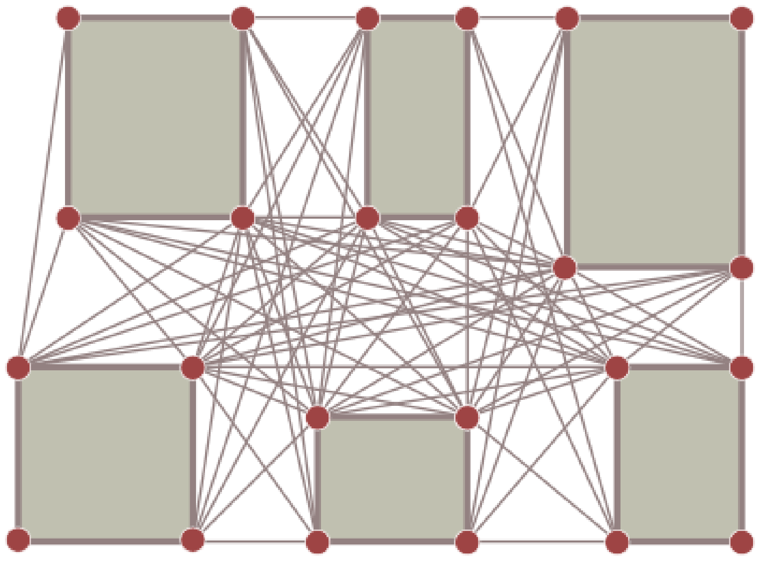


*Figure 9: Navigation graph based on polygons. Source: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html*

To illustrate the complexity of the computation, we will take an example from AoE, where in a typical game, up to 8 players with up to 200 units and at least 30 structures take part. For simplification, we consider each entity to be a square (having 4 corners). This example yields *8 * (200 + 30) * 4 = 7630* vertices in the navigation graph, with $7360^2$ *= 54 169 600* edges, for which, the free path would have to be determined in every round (for details on rounds, visit section 2.1). Notice that we did not consider obstacles, and the complexity of the algorithm is already very well visible. For this reason, the designers of RTS games use a different implementation of the map – they subdivide it into tiles.

### 2.5.2 Tile-based map

In order to improve the detection of free areas in the map and to make the construction of the navigation graph easier, designers of RTS games very often use a subdivision of the map into cells, which are called *tiles*. A tile may only be free or occupied, and any area in the map is free if all tiles it contains are free. Using this approach, we do not have to compute polygon intersections – we only have to go over the tiles. Tiles have also an effect on the nature of entity movement in the map, because moving entities can not move entirely freely – they have to move from one tile to an adjacent one.

**Shape of tiles**

To our best knowledge, 2 shapes of tiles are used in strategy games (see Figure

10): (1) square-shaped tiles and (2) hexagon-shaped tiles.

**Square-shaped tiles** allow the moving entity to move in 4 or 8 directions, depending on whether we allow diagonal movement. It is important to notice that the distance to the border is different along different directions.

**Hexagon-shaped tiles**, on the other hand, provide the moving entity with 6 directions, and the distance to the border is equal along all of them.

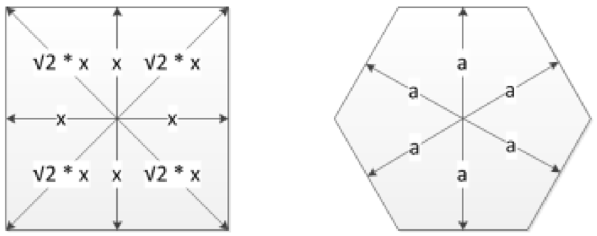


*Figure 10:Square-shaped and hexagon-shaped tiles. Notice that the distance to borders is different along different directions.*

While considering the choice of the shape, we have to think about two criteria. Square-shaped tiles can be easily ordered into a two-dimensional array, which is an advantage of them. On the other hand, different distance to the border along different directions is their greatest disadvantage.

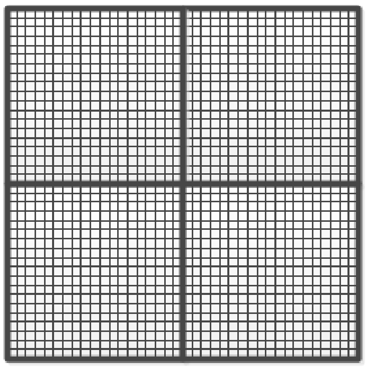


*Figure 11: Subdivision of tiles in RTS games*

We have to realize that in RTS games, tiles are subdivided into smaller units (in fact, pixels), as we show in Figure 11, and each moving entity has a property called velocity, which is given in units per round, so that the distance to the tile border does not play such an important role in RTS games (for contrast, in turn-based strategies, velocity is usually given in tiles per round and the distance to tile borders is very important).

The above discussion implies that if we decided to use a map based on tiles, the tiles would be of a square shape.

**Dynamic construction of navigation graph**

Navigation graph construction from a tile-based map is easier than from polygons: each free tile is converted into one vertex, and two vertices are connected with an edge if they represent adjacent tiles. We do not have to do complex computations to determine whether there is a free path between adjacent tiles, because it is if both of them are free.

Again, we will illustrate the approach with the example from AoE. The maximum map size is 512 x 512 tiles, and entities move in 8 directions from the tiles. Altogether, it yields a navigation graph of maximum *512 * 512 = 262 144* vertices, but only *262 144 * 8 / 2 = 1 048 576* edges. This example shows that the navigation graph constructed from a tile-based map consumes less computation time to be created and also to be searched.

For all the reasons listed above, we will use a map based on tiles. As we already decided, the tiles will be square-shaped.

### 2.5.3 Path

The map in our game will be composed of tiles so that a path in the map has to be represented as a list of tiles. We have found 2 possible representations of a path: (1) **relative** and (2) **absolute** representation.

We will explain the difference between them using the following example: a unit is moving from tile with coordinates `[50, 50]` over tile `[50, 51]` to tile `[50, 52]`. A relative path contains two points: `[0, 1]`, `[0, 1]`. An absolute path would also contain two point, but it would be `[50, 51]` and `[50, 52]`.

Let us consider following situations: following its path, a unit enters a teleportation device, which changes its position abruptly (such a device exists in Original War, for instance). After the unit appears at new position, we expect it to continue to move in the previous direction – and here, the relative path, which gives the direction and not the next point, comes in handy, because it produces the expected behavior, whereas the absolute path would have to obtain the unit's previous position to compute the correct direction (and rotation). For this reason, we will use the relative representation of path in our game.

### 2.5.4 Pathfinding in map

We already know that pathfinding algorithms are graph algorithms operating on navigation graphs. They range from simple ones (like choosing a random direction) to complex algorithms. We will describe 2 algorithms, which are used typically: (1) Dijkstra's algorithm [40], and (2) A* search algorithm [41].

#### 2.5.4.1 Dijkstra's algorithm

Dijkstra's algorithm computes distances (the cost of shortest path) between a given starting vertex ("origin") and all other vertices. If we want to compute the distance from the origin to only one vertex in the navigation graph, we can stop the algorithm after this vertex has been reached. The algorithm runs in *O(m + n*log(n))* [40], where *n* is the number of vertices and *m* the number of edges in the navigation graph.

We will briefly describe the algorithm. It uses two sets of vertices: S and Q, with the following rules:

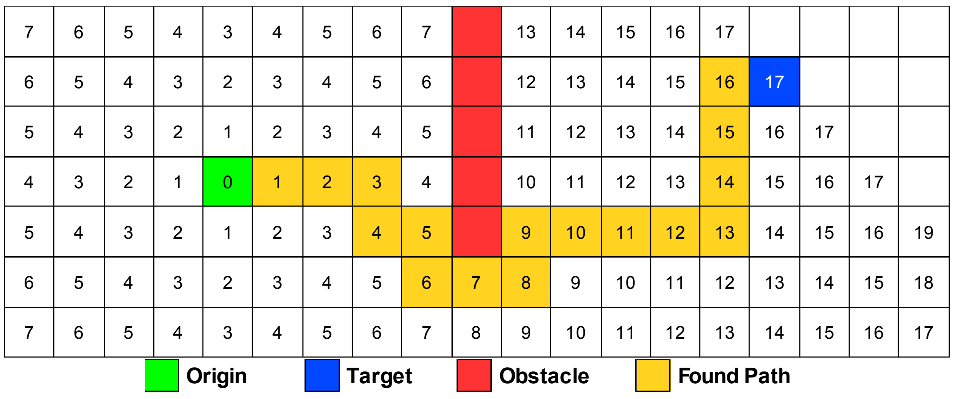
- Vertex v ∈ S if its shortest distance from the source has already been computed. After `Initialize(G,s)` it holds that S = ∅.
- Otherwise, v ∈ Q, so that Q = V \ S, where Q is implemented as a data structure supporting the search for a vertex with the lowest distance value (a min-heap, for example).

Method `Initialize` assigns infinity distance value to every vertex, and zero to the source. It marks all vertices unvisited. Let's now have a look at the pseudo-code of the algorithm:

```
Dijkstra (G,s)
     Initialize (G,s)
     S := Ø
     Q := V(G)
     while (Q ≠ Ø) do
           u := Extract-Min (Q)
           S := S ∪ {u}
           for each {v ∈ V(G) | (u,v) ∈ E(G)} do
                 Relax(u,v)
```

Method `Relax(u,v)` computes new distance value of vertex *v* and overwrites the previous distance value if it is greater than the new one.

We show an example of the pathfinding done by the Dijkstra's algorithm in a square-tile-based map in Figure 12. The numbers in tiles denote their distance from the source. Notice that the yellow path is not the only one possible – there are many equivalent paths.



| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 🟥 | 13 | 14 | 15 | 16 | 17 | | | | |
| 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 🟥 | 12 | 13 | 14 | 15 | 16 | 17 | | | |
| 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 🟥 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | | |
| 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 🟥 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 🟥 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 19 |
| 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 18 |
| 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

*Figure 12:Pathfinding using the Dijkstra's algorithm. It visited 111 tiles.*

### 2.5.4.2 A* search algorithm

Unlike the Dijkstra's algorithm, A* algorithm computes the shortest path in the navigation graph from the source to one target vertex only. The algorithm keeps a queue Q of visited vertices sorted by their fitness (F), which is a value obtained as F = G + H, where G is the cost of the path from the source to the vertex and H is an estimate of the cost of the shortest path from the vertex to the target – the heuristic. The choice of an appropriate heuristic plays an important role. After we have a look at the pseudo-code of the algorithm, we will investigate possible heuristics.

```
Astar(G,s,t)
     Q := {s}
     while (Q ≠ Ø) do
           Dequeue vertex v with lowest F from Q
           closed(v) := true
           For each neighbor u of v where not closed(u) do
                 If u == t then
                       ReconstructPath (t, s)
                       break
```

```
                else
                        F(u) = G(u) + H(u)
                        Q := Q ∪ {u}
```

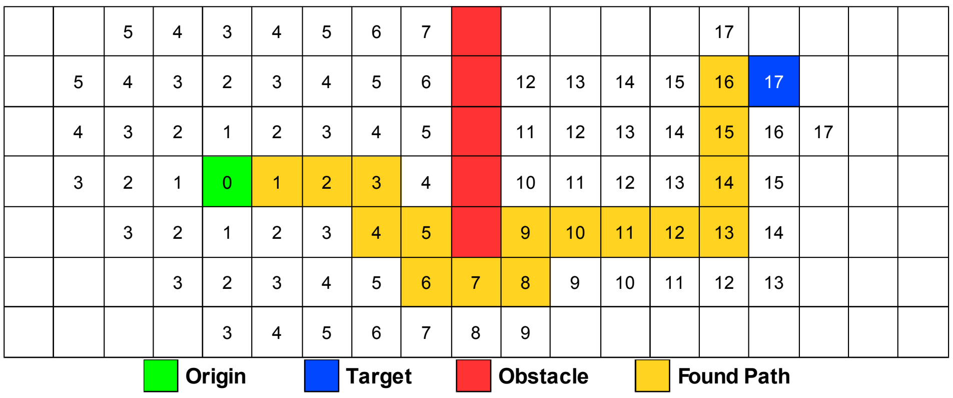
#### 2.5.4.3 Heuristic for A* and optimality of shortest path

Let's now investigate the choice of a heuristic for the A* pathfinding algorithm. The heuristic *H(u)* for a given vertex *u* (the estimate of the cost of shortest path from *u* to the target) plays an important role because it strongly influences the number of vertices which have to be visited before the target is reached, thus influencing the time the algorithm consumes.

The estimate *H(u)* may be of any non-negative value, with a simple rule: the higher the estimate is, the less vertices have to be visited before the path is found. For this reason, a large number of different heuristics for A* can be found; we will concentrate on their admissibility (*H(u)* is admissible [42] if it never overestimates the cost of reaching the target from vertex *u*) rather than on listing them.

**Admissible heuristics:** If the heuristic is admissible, then A* is guaranteed to find the shortest path. For a tile-based map, one of the heuristics which are used very often is the Manhattan heuristic (we will denote it as $H_M(u)$ for vertex *u*). It is simply the sum of x and y distances to the target.

To illustrate the use of the Manhattan heuristic, we use the example from Figure 12, but this time searched by the A* algorithm, in Figure 13. Notice that there are



| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 🟥 | | | | | 17 | | | | |
| | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 🟥 | 12 | 13 | 14 | 15 | 🟨 16 | 🟦 17 | | | |
| | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 🟥 | 11 | 12 | 13 | 14 | 🟨 15 | 16 | 17 | | |
| | 3 | 2 | 1 | 🟩 0 | 🟨 1 | 🟨 2 | 🟨 3 | 4 | 🟥 | 10 | 11 | 12 | 13 | 🟨 14 | 15 | | | |
| | | 3 | 2 | 1 | 2 | 3 | 🟨 4 | 🟨 5 | 🟥 | 🟨 9 | 🟨 10 | 🟨 11 | 🟨 12 | 🟨 13 | 14 | | | |
| | | | 3 | 2 | 3 | 4 | 5 | 🟨 6 | 🟨 7 | 🟨 8 | 9 | 10 | 11 | 12 | 13 | | | |
| | | | | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | | |

🟩 Origin 🟦 Target 🟥 Obstacle 🟨 Found Path

*Figure 13: Pathfinding performed using the A* with Manhattan heuristic. It visited 84 tiles.*

more than one optimal path.

**Inadmissible heuristics** provide no guaranty that the found path would be the shortest path in the navigation graph. On the other hand, they may save a great deal of computation time needed to find the (suboptimal) path because the algorithm visits less vertices.

As we see, with the choice of the heuristic, we also choose between the optimality of the path and the number of steps the algorithm has to undertake. A question arises: What would happen if we gave up some of the optimality for the sake of faster computation? An answer to this question is given by Christer Ericson in his article "*Don't follow the shortest path!*" [43]. Here, he mentions the following idea:

"*It may be less important to find a solution whose cost is absolutely minimum than to find a solution of reasonable cost within a search of moderate length. In such* *a case, one might prefer an A* that evaluates nodes more accurately in most cases but sometimes overestimates the distance to a goal, thus yielding an inadmissible algorithm.*"

The quotation says that the heuristic does not always have to be admissible, because the path we are searching for does not have to be optimal – it may be more important to save computation time than to provide an optimal solution.

In Figure 14, we illustrate the use of inadmissible heuristic $H(u) = 2*H_M(u)$ with the example used in Figure 12 and Figure 13. The values in tiles are presented as G+H, where G is the distance from source, and H the heuristic value. Notice that some tiles have a wrong G value (they are circled in Figure 14), which is one of the results of the inadmissibility of the heuristic. Furthermore, the algorithm did not find

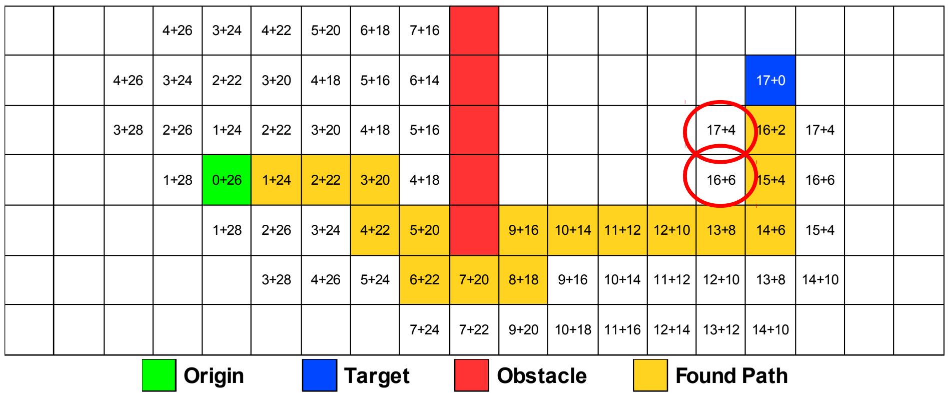


*Figure 14:Pathfinding using inadmissible heuristic $H(u) = 2*H_M(u)$. It visited 65 tiles.*

the path it found in Figure 12 and Figure 13 (yet it yielded an optimal path). On the other hand, the algorithm visited less tiles than A* with Manhattan heuristic and than Dijkstra's algorithm.

### 2.5.5 Pathfinding in our game

We described the Dijkstra's algorithm, whose purpose is to find distances from the source to all vertices, and the A* pathfinding algorithm with different heuristic, whose purpose is to find a path between source and target. We also presented examples (Figure 12, Figure 13, Figure 14) of how many tiles the algorithms have to visit in a typical case in order to find the path from source to target, and we have seen that the A* is more efficient than Dijkstra; for this reason, we will use A* for pathfinding in our game.

Let us now investigate 3 issues we have to handle in order that the pathfinding in our game works as players expect it to work without consuming too much computation time: (1) use of an appropriate heuristic, (2) collisions between moving units, and (3) dealing with an unreachable target.

#### 2.5.5.1 Use of heuristic

We first implemented A* with Manhattan heuristic, which always finds an optimal path and performs well over short and medium distances. At long distances, especially on a map with large obstacles, the heuristic often floods the navigation graph. While experimenting, we discovered that this effect may be reduced if the value of the Manhattan heuristics is multiplied by a constant $C > 1$. The greater $C$ is, the less vertices have to be visited in the navigation graph (and the less optimal path is found). In the current implementation, we are using $C = 4$, which is a constant resulting from our experiments as a good relationship between quality of the path and time consumed by the algorithm.

We also considered the option of using a dynamic coefficient, so that the quality of the path would depend on the amount of computation time available. This approach seems logical from a mathematical point of view, because it tries to find a path as optimal as possible regarding the current situation. However, it may also lead to an unpredictability of the path, because it depends on circumstances the player neither understands nor is able to influence.

We will explain the problem using following example: *A unit follows a very suboptimal path starting at point P, and it evades enemy's assets which would destroy it. Player, knowing that the unit has chosen such a path, sends another unit from point P to the same target while expecting it to follow the same path. However, this time, there is more computation time available at the CPU and the pathfinding algorithm finds an optimal path, which, by accident, leads very close to enemy's assets, so that the second unit gets destroyed.* This is a situation which we believe should not happen in RTS games, which is the reason why we use a static coefficient.

#### 2.5.5.2 Collisions

After implementing A*, we found out that the algorithm works well for a single unit. However, in modern RTS games, it is usual that the player controls a group of units; they consider each other static at the time they perform pathfinding. For this reason, many collisions between them occur once they start to move, which results into the need to recompute the path with respect to the current situation.

We found out that an option which improves the situation (decreases the number of collisions) is that the unit waits some time after collision before it runs the pathfinding algorithm again. Nevertheless, if two units collide with each other and they both wait the same amount of time, they are likely to run into each other again. To prevent this, we decided to generate a random wait interval after a collision, which significantly improved the behavior of units.

#### 2.5.5.3 Unreachable target

We also had to think about what happens if a unit is given a command to move to an unreachable target position. If there is no path available in the map, the A* algorithm has to investigate all reachable vertices in the navigation graph before it knows that there is no path. However, there may be a large number of tiles in the map and the algorithm may consume a great deal of time before it finds out that it can do nothing. To limit the number of vertices visited by the algorithm, we use a simple estimate based on the Manhattan heuristic. We set a constant $C$ (after some experimenting, we obtained an optimal value $C = 20$) and define the upper bound $UB$ of visited vertices for search from source $S$ to target $T$ as:

$$UB = (|S.x - T.x| + |S.y - T.y|) \cdot C$$

Using this formula, the upper bound is dependent on the distance between source and target. We also suppose that most of the searches for path are done at short to middle distance (they do not consume so much computation time), where this formula does the most work in preventing flooding if the target is not reachable.

## 2.6 Network communication for remote game sessions

Goal 1.3 states that there has to be support for remote game sessions for multiplayer games. On a high level of abstraction, this means that there has to be a system of sharing information about the game progress over the network.

In order to find out what type of information can be shared in the game, let's take an example of a unit. The player wants the unit to move from point P to point Q, and he or she gives it command *C* to do so in round *r* (for details about rounds, see section 2.1). The unit moves while changing its position in the map every round until it arrives at point Q. To describe its movement, we define a sequence of positions $P = P_0, P_1, \ldots, P_N = Q$, where $P_i$ is the position of the unit in update $r + i$. ($P_i$ does not have to be aligned to a tile – it can be any position, depending on its representation in the game, e.g. given in pixels). We also define the difference $d_i$, which is the distance covered by the unit in update $r + i$. The process is illustrated in Figure 15.

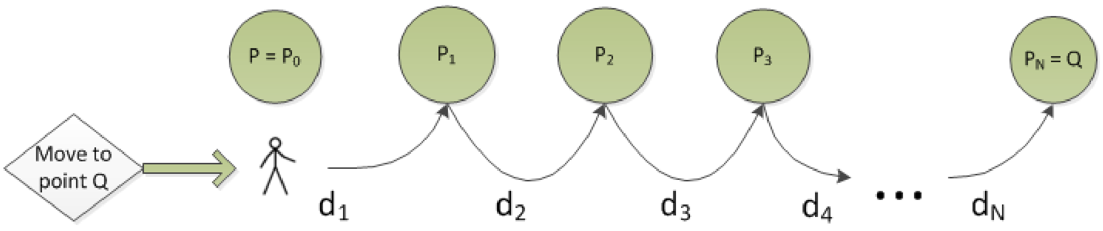


*Figure 15: A unit moving from point $P = P_0$ to point $Q = P_N$*

As we see from the description of the movement, there are 3 types of information in the game which can be shared:

(1) Absolute positions of the unit ($P_i$)

(2) Relative changes of the unit's position ($d_i$)

(3) Input from the player (the command *Move to point Q*)

While considering the options, we will pay especial attention to the following criteria:

- The quality of the game which can be achieved using the option (the experience delivered to the user is important, because the user wants to see a smooth and well-running game).
- We can not presume that the game will only be played in LANs, so that we have to pay attention to the amount of information shared across the network.
- The use of the underlying communication protocol (UDP [44] vs. TCP [45]).

**Sharing absolute positions**

Sharing absolute positions, we assume that they are computed at one of the communicating computers and broadcasted to all other computers in the network. In case of loss of a piece of information (e.g., one of the communicating computers receives the following sequence: $P_1$, $P_2$, *nothing*, $P_4$, i.e. the information about position $P_3$ has been lost), the game will not be incorrect (it will still be synchronized), although it may look slightly choppy. This may have a negative influence on the user's experience, especially if it should happen more often, because the unit may seem to be “jumping” instead of moving smoothly.

For this type of information shared, the UDP protocol can be used. The protocol does not guarantee the delivery of the information, but the game does not need this guarantee for the right functionality.

There will be a large amount of information shared between the games in each turn, because we suppose a large number of entities in the game – for each entity, the absolute position will have to be sent.

**Sharing relative changes**

Similarly to absolute positions, we assume that the relative positions are computed by one of the communicating computers and sent to all other computers in the network.

Unlike absolute positions, the relative positions have to be delivered all (with no data losses) and in the right order – to illustrate this, we will use the example from Figure 15: Let's suppose that $d_i = d_j$ for all *i,j* (the distances between $P_i$, $P_{i+1}$ are equal). In case one $d_i$ gets lost and does not reach some of the communicating computers, the unit arrives there at position $P_{N-1}$ instead at position $P_N$, which results into lost synchronization of the game. In order to ensure the correct delivery of all information and its right order, we would have to use the TCP protocol.

Compared to sharing absolute positions, this approach decreases the amount of information sent via network, because it only has to send the updates – and it is very likely that there are many entities whose state does not change in every turn. On the other hand, the use of TCP may cause some lagging because the delivery may be slower (due to the properties of TCP protocol) and the game has to wait for all packets in one turn to arrive before proceeding into next turn.

**Sharing input from players**

If the games share the players' input (in Figure 15, it is the command *C* for the unit to move to point Q), the evaluation of the command is processed in a distributed manner – at each of the computers connected in the network. We have to ensure that the command is delivered to all the computers (using TCP) and that the computation proceeds equally at all of them.

This approach considerably decreases the use of network resources as we only send small pieces of information – the players' input, which will eliminate the problem with lagging caused by the TCP protocol described in previous section.

**Sharing information in our game**

After we have described the various possibilities, we have to choose one of them. We believe that sharing relative changes is the least optimal option, because it makes use of TCP protocol while sharing a relatively large amount of information, thus potentially slowing down the game progress or causing lagging.

The option of sharing absolute information seems to be more suitable, because it does not cause lagging. On the other hand, it may let the game look choppy as much more information has to be sent via network than if we just shared the relative information, and any lost data packet causes the game to look less smooth.

For this reason, we choose the third approach (sharing input from players and distributed evaluation of them) to be used in our game, as it limits the amount of shared information to the minimum. Any time a command is issued by a player, it will be transmitted to all other games, so that the input in all the communicating games is the same.

### 2.6.1 Enhancements in network communication

After we implemented the system for network communication we have chosen, we found out that under certain circumstances, it may cause the game to lag – especially if there are differently powerful computers communicating in the network. Then, it may happen that one of the computers has to perform a very demanding computation in one of the rounds, which takes a large amount of computation time.

In their article "*1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond*", [46] Mark Terrano and Paul Bettner offer a solution to this problem. They introduce a communication offset (*CO*) for commands so that a command

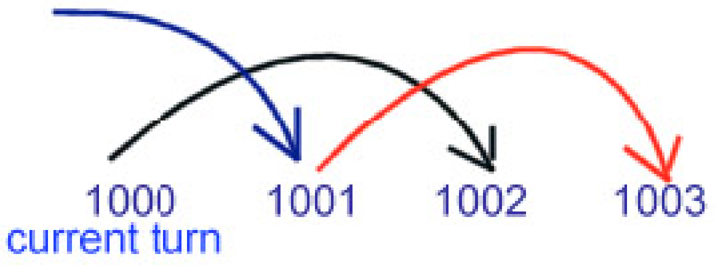


*Figure 16: The communication offset for commands issued in the current turn. Here, it is equal to 3. Source: http://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php*

issued in round *N* is evaluated in round *N* + *CO*, as can be seen in Figure 16.

This technique improved the quality of network communication in our game significantly, as it really prevented lagging. We are using CO = 3, which means that if a demanding computation occurs at one computer in round N, it has still 3 rounds to catch up with other computers without causing any lagging.

### 2.6.2 Correctness of information shared across games

In goal 2.5, we state that network communication can be prone to mistakes which are difficult to debug and that we want to include a possibility of testing the correctness of all information shared across the games.

We decided that the games will share players' inputs (commands) and that the computation will be performed by each of the games separately. For this reason, we have to ensure that each game receives the same input (the same sequence of commands) in each round and that the computation based on the state of the game and the input yields the same results.

To achieve this, we have to seed random number generators so that if anything is controlled by random numbers, every computer produces the same random number at the same time. In order to test the correctness of all random numbers, we have to log them; we have to provide a tool for comparing them on the right order.

Furthermore, it is essential that all communicating computers obtain the same sequence of commands. In order to compare the sequences of commands, we have to log all the commands processed by the games in the order the game processes them. We have to provide a tool which will compare the logged sequences and decide if all logged sequences at all computers are in the same order.

# 3 Development documentation

In this part, we will introduce the structure of the program and its implementation. To develop the program, Visual Studio 2010 Ultimate with .NET Framework 4 was used. It is also necessary to have the Microsoft XNA Game Studio 4.0 installed for the project to open correctly.

## 3.1 Program structure

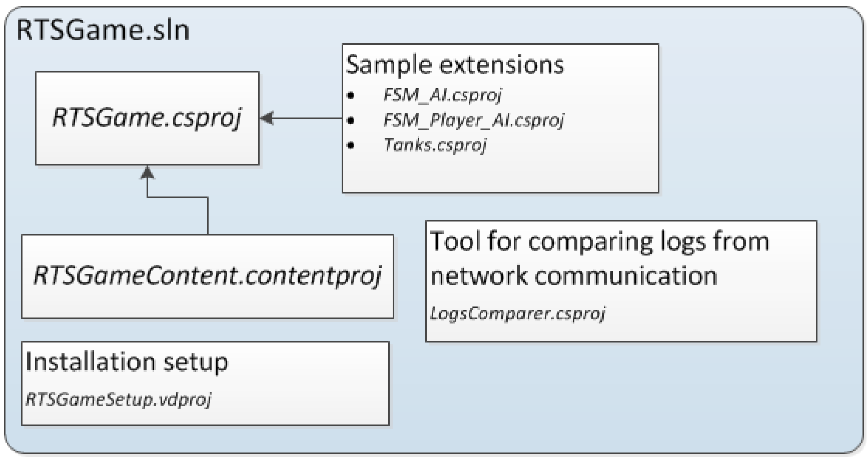


*Figure 17: The structure of the solution* `RTSGame.sln`

The solution `RTSGame.sln`, whose structure is shown in Figure 17, is accessible from Attachments [A] (directory `\RTSGame`). It contains the main project `RTSGame.csproj`, 3 projects with sample extensions, the `ContentManager` (`RTSGameContent.contentproj`), which is a run-time component loading content (e.g. images, fonts, sounds, etc.) from files produced by the design time content pipeline, project `LogsComparer.csproj` providing a tool for comparing logs from network communication, and project `RTSGameSetup.vdproj`, which creates an installation packet.

We will describe the structure of the project `RTSGame.csproj`, because it contains all the important namespaces and types. The description of the projects with sample extensions will be provided along the description of the particular components whose extensions they contain. Details about projects `LogsComparer.csproj` and `RTSGameSetup.vdproj` will be given in sections 3.5.8, and 3.8, respectively.

### 3.1.1 RTSGame.csproj

This project represents the core of the whole application. The entry point into the

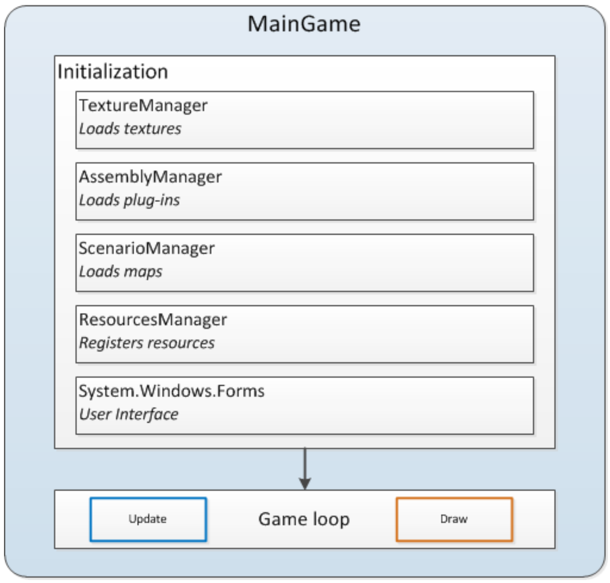


*Figure 18: Initialization of the game*

program is class `MainGame` (see Figure 18), which runs the main game loop after it initializes the program.

During the initialization, `AssemblyManager` loads extensions of AI for computer players from directory `.\PlayerAI`, extensions of AI for game entities from directory `.\AI`, and extensions of game entities from directory `.\obj`. `ScenarioManager` loads maps for the game from directory `.\scenarios`.

The extensions of maps and AI for players are then offered to the player before he or she starts the game as available settings for the game – the player chooses the number of AI players, the AI for them and the map where the game will be played.

`ResourcesManager` registers a `ResourceHolder` for each to-be-collected resource, which is a class containing information about the nature of the resource such as whether it is an obstacle in map, if it is destroyed if run over, if it grows in time, if its amount decreases while it is being collected, etc:

```
class RubberHolder : ResourceHolder {...}
ResourcesManager.registerResource(new RubberHolder());
```

*Code example from class MainGame, file MainGame.cs*

The main game loop (Figure 19) implements the subdivision of the continuous time progress into rounds (for details, see section 2.1) with methods `MainGame.Update` (computes one step in the game) and `MainGame.Draw` (displays the current state of the game). In order that the game runs smoothly, variable `MainGame.IsFixedTimeStep` is set to `false`, which enforces calling `Draw` regularly after `Update`. However, due to the fact that the default rate of 60 fps is too high, the interval for calling `Update` has been increased to 33 ms per frame (variable `mMillisecondsPerFrame`), which decreases the `Update`-rate to 30 fps. Method `Draw` may be called more often to provide a smoother impression of the game, but between two `Update`s, at least one `Draw` is always called.

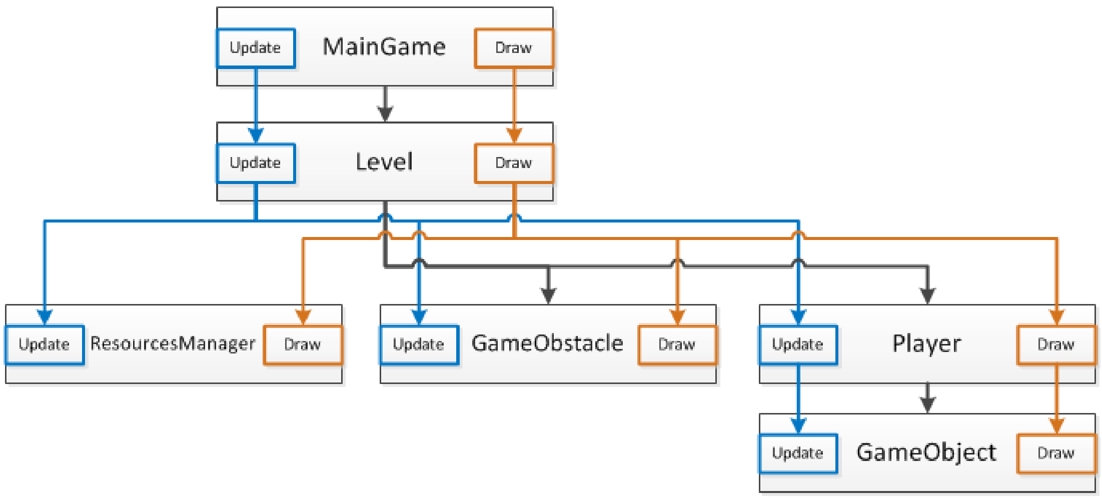


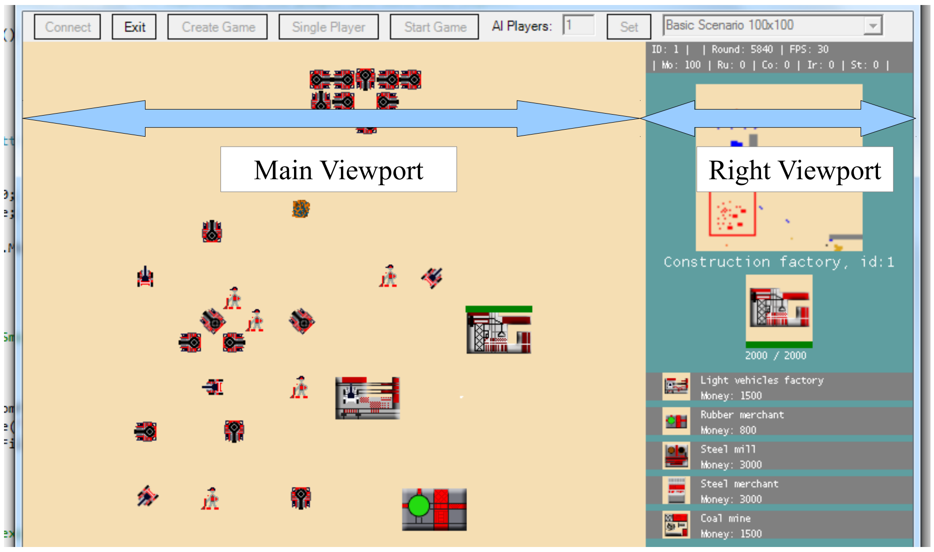
*Figure 19: The structure of the Update and Draw subroutines performed in one round*

While class `MainGame` is responsible for initialization of the program and for providing the user interface, class `Level` represents one session played in the game. It bears the most important components of the game and connects them together: it holds the map (`IGrid<SquareTile> mGrid`), list of obstacles (`ListOfGameObjects mObstacles`) present in the map, and list of players playing the game (`List<Player> mPlayers`). Human players are represented by class `Player`, computer players by `AIPlayer : Player`.

`Player` encapsulates the list of entities it controls (`ListOfGameObjects mEntities`), and `AIPlayer` extends `Player` with artificial intelligence (described in section 2.3.4) provided by `FiniteStateMachine mStateMachine`.

Entities and obstacles are represented by class `GameObject` or its successors. There are entities of 3 types in our game: (1) static entities, (2) workers, and (3) vehicles (specified in `enum ObjectType`). Entities have their own artificial intelligence (`FiniteStateMachine mInnerLoop`), so that they behave as (partially) autonomous agents, as described in section 2.4.

### 3.1.2 Game window



```
ID: 1 |  | Round: 5840 | FPS: 30
| Mo: 100 | Ru: 0 | Co: 0 | Ir: 0 | St: 0 |
Construction factory, id:1
2000 / 2000
Light vehicles factory
Money: 1500
Rubber merchant
Money: 800
Steel mill
Money: 3000
Steel merchant
Money: 3000
Coal mine
Money: 1500
```

*Figure 20: The screen of the game is divided into two `Viewports`.*

The result of rendering (`MainGame.Draw`) is displayed in the game window, which is divided into two `Viewports`, as shown in Figure 20. Right `Viewport` shows information about the player, the amount of resources the player has, map thumbnail, and information about active entities (health, thumbnail, name, etc.). The class responsible for displaying the information in the right `Viewport`, called `RightColumn`, is owned by `MainGame` (member variable `mRightColumn`).

Main `Viewport`, which shows the game progress, contains the result of rendering the `Level` (`Level.Draw`).

### 3.1.3 Namespaces in RTSGame.csproj

The project `RTSGame.csproj` is subdivided into 6 namespaces logically grouping types to make the project clear and understandable:

- **namespace RTSGame**
  This namespace is the umbrella namespace of the project. It contains types which are not specific enough to belong into any other namespace.
- **namespace RTSGame.FSM**
  Contains types related to the finite state machine which we use for artificial intelligence of computer players and for the internal intelligence of entities.
- **namespace RTSGame.GameMap**
  Contains types related to the tile-based map, tiles, path and pathfinding.
- **namespace RTSGame.GameObjects**
  Contains types related to game entities and obstacles and to their extendability.

- **namespace RTSGame.ResourceManagement**
  Contains types related to resource management, apart from types for resource collectors and resource transporters, which are included in namespace `RTSGame.GameObjects`.
- **namespace RTSGame.Networking**
  Contains types related to network communication.

## 3.2 Identification of types

For the sake of extendability of the program with run-time extensions, we had to develop a concept of type identification without neither us, nor the compiler knowing whether the types exist in the type system at compilation time. We will explain the problem we faced using an example from projects `Tanks.csproj` and `RTSGame.csproj`.

The project `RTSGame.csproj` contains class `FSM_AI_Player_NoActivity`, which implements one of possible strategies for computer players in the game. The designer of this class wants it to be able to create attack units of types `SmallTank` and `BigTank`, which are included in project `Tanks.csproj` (loaded as a run-time extension), so that the compiler does not know about them at the time project `RTSGame.csproj` is being compiled.. For this reason, we represent the types as strings and provide a method for instantiating them at run-time from this representation:

```
Type smallTankType = AssemblyManager.getTypeFromString("SmallTank");
Type bigTankType = AssemblyManager.getTypeFromString("BigTank");
```

*Code example from class FSM_AI_Player_NoActivity, file FSM_AI_Player_NoActivity.cs*

The method `AssemblyManager.getTypeFromString` receives the string representation of the type and returns a `Type` or `null`. Notice that the type is identified by the name of the class only – this is due to the fact that in the initialization phase of the game, we create a dictionary `Dictionary<string, Type> AssemblyManager.mDictionaryOfSimplifiedTypes`, which contains `Types` present in the game identified by their class names. In case the `Type` is not found based on its class name, all assemblies plugged into the game are iterated while trying to find it according to its full name (which includes the name of the namespace – i.e., in case of class `SmallTank`, its full name is `RTSGame.GameObjects.SmallTank`). For this reason, the type can also be identified by its full name, and the following piece of code is equivalent to the previous one:

```
Type smallTankType = AssemblyManager.getTypeFromString("RTSGame.GameObjects.SmallTank");
Type bigTankType = AssemblyManager.getTypeFromString("RTSGame.GameObjects.BigTank");
```

*Modified code example from class FSM_AI_Player_NoActivity, file FSM_AI_Player_NoActivity.cs*

The approach we described in this section is used in numerous places in the project – for example to identify FSM for entities (for details, see section 3.6.6), to identify entities created in `Buildings` (see section 3.3.6), etc.

## 3.3 GameObject

`GameObject` is an abstract class representing a common ancestor for entities and obstacles (grouped into the namespace `RTSGame.GameObjects`). The tree of types inheriting from it is shown in Figure 20, where fields with white background contain

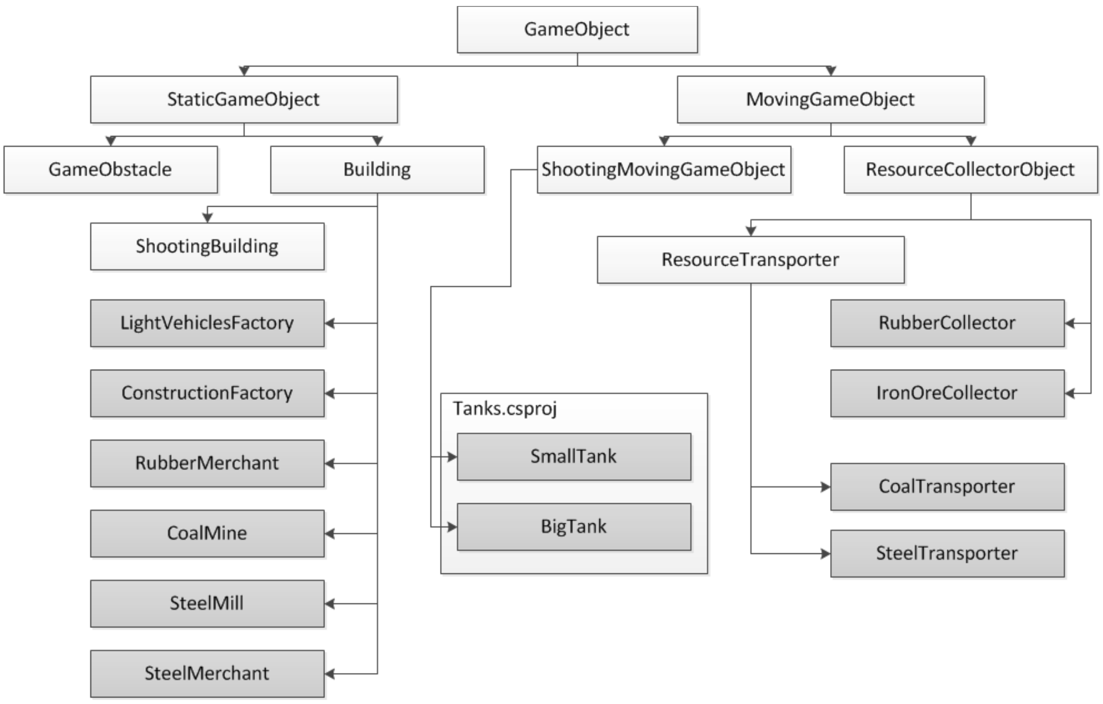


*Figure 21: Classes extending the* `GameObject` *class*

abstract classes, whereas the gray background stands for real units and buildings present in the game.

We also provide sample extensions of entities in project `Tanks.csproj`, which contains types `SmallTank` and `BigTank`. The extensions have to be inserted into directory `.\obj` in form of .dll files. The project `Tanks.csproj` fulfills this condition, because it is set up to compile as a .dll file to directory `RTSGame\RTSGame\bin\x86\XXX\obj\`, where `XXX` stands for `Debug` or `Release` (depending on settings of the project `RTSGame.csproj`).

### 3.3.1 Initialization

All properties of the game entity (name, number of hit points, damage it does to other units, etc) have to be specified in `GameObject.initGameObject`, unless we want them to have default values, because the method is called in the parameterless constructor, which is in turn needed for loading run-time extensions by `System.Reflection`. The following code shows the method `initGameObject` used in one of the units from our game:

```
public override void initGameObject() {
     mSpeed = 3;                                  // in pixels per second
     mName = "Scorpion";                          // the name given to the entity
     mNumberOfHitPoints = 80;                     // the total amount of hit-points
     mNumberOfRemainingHitpoints = 80;            // the current amount of hitpoints
     mObjectType = objectType.vehicle;            // type of the entity
```

```
    mCost = new Resources();                    // reset the cost
    mCost.setResource(ResourceTypes.Money, 180);   // the cost in Resources
}
```

*Code example from class SmallTank, file Tanks\Tanks.cs*

Furthermore, method `loadContent` has to be overwritten to load textures. Notice that the method loads textures either from the `ContentManager`, or textures directly inserted into the project, whose Build Action (to be found in Properties) is set to Embedded Resource.

```
public override void loadContent() {
    mTxtMainTexture = LoadTextureFromFile("Tanks.SmTankChassis.png");
    mTxtColors = LoadTextureFromFile("Tanks.SmTankColors.png");
    mTxtThumbnail = LoadTextureFromFile("Tanks.SmTankThumb.jpg");
}
```

*Code example from class SmallTank, file Tanks\Tanks.cs*

The textures `mTxtMainTexture` and `mTxtColors` are used for drawing the unit into the main `Viewport`, `mTxtThumbnail` is displayed in the right `Viewport` (for details on `Viewports`, see section 3.1.2) as a thumbnail of the entity.

### 3.3.2 Class MovingGameObject

`MovingGameObject` is the abstract representation of a unit – in contrast to a `StaticGameObject`, a `MovingGameObject` is able to change its position – it moves along its path (`MovementPath<Point> mMovementPath`).

While a `MovingGameObject` is moving, it is heading to the `mOriginalTarget`. If the `MovingGameObject` is given a command to move towards a new target, it may obtain the command just in the middle of interpolation between tiles. It must first finish the interpolation before it may change its direction to produce smooth movement. In order to save a new target until the interpolation has been finished, it uses `mNewTarget`.

### 3.3.3 Interactions between game entities

In section 1.1.3, we described interactions between game entities as the ability of game entity A(ttacker) to attack game entity D(efender), i.e. A performs such an action that D gets damaged; D looses some of its health (`mNumberOfRe-mainingHitpoints`). `AttackingBuilding` and `AttackingMovingGameObject` are able to attack other game entities; they have an instance of `AttackAttributes`, which encapsulates all member variables and methods related to attacking.

If A attacks D, then A calls `D.receiveAttack(A.mAttackAttributes)`. The attacked entity D computes the resulting damage and subtracts it from its remaining hit points.

In order to define new interactions between game entities, we have to work with the method `GameObject.receiveAttack`.

### 3.3.4 Resource collectors and transporters

Resource collectors collect resources available in the map, resource transporters transport resources between structures. We represent them with `ResourceCollector-Object` and `ResourceTransporter : ResourceCollectorObject`, respectively.

We will explain the methods and member variables of `ResourceCollector-Object` using an example: the collector receives a command to go on a specific tile to collect resource R. It arrives on the given tile, and if its `mResourceCapacity` of the resource R is greater than zero, it collects the amount of resource R given in `mResourceTransfer` at the rate (per round) given in `mTransferInterval` until its `mResourceCapacity` is reached. Then, the collector searches for an appropriate structure accepting resource R using 3 rules (Figure 22).

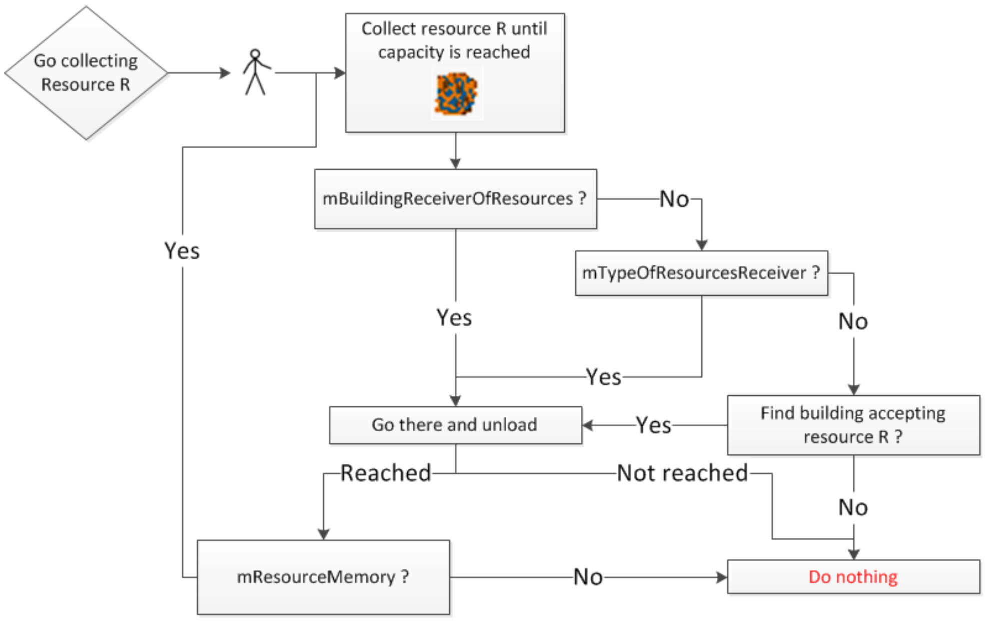

Go collecting Resource R
Collect resource R until capacity is reached
mBuildingReceiverOfResources ?
No
mTypeOfResourcesReceiver ?
Yes
Yes
No
Yes
Go there and unload
Yes
Find building accepting resource R ?
Reached
Not reached
No
mResourceMemory ?
No
Do nothing

*Figure 22: Rules for resource collectors*

(rule 1) If the `mBuildingReceiverOfResources` is not null, the resource collector sets off to this building.

(rule 2) If the `Type` of receiver (`mTypeOfResourcesReceiver`) is set, the collector searches for any building of that type. If non is found, it does not perform any action. This is useful if the collector should cooperate with one type of building only (and ignore buildings of other types).

(rule 3) If nothing is set, the collector searches for the closest building accepting its resources, no matter what type of building it is.

In order to find the building for unloading resources, the collector uses method `findBuildingToUnloadResources` implementing the rules (rule 1) – (rule 3). Having unloaded the resources, the collector checks its `mResourceMemory`, and if it remembers a `ResourceGroup` for collecting, it goes back there.

Resource collector may be able to collect more types of resources at different rates and in different amounts, and it may have different capacities with respect to different resource types. This is one of important parts of the design-extendability of the program. Furthermore, the properties `mBuildingReceiverOfResources` and `mTypeOfResourcesReceiver` may be set for each resource type separately, and the collector chooses the option based on `mResourceType`. This is due to the fact that variables `mBuildingReceiverOfResources` and `mTypeOfResourcesRe-ceiver` are implemented as properties over the dictionaries:

```
public Dictionary<ResourceTypes, int?> mBuildingIDsForResourceTypes;
public Dictionary<ResourceTypes, Type> mResourceReceiversTypes;
```

*Code example from class ResourceCollectorObject, file ResourceCollectorObject.cs*

There are not many differences between resource collectors and resource transporters; only the resource collector collects resources present in a map, whereas the resource transporter considers a building to be the source of resources (`mBuildingSourceOfResources`). If the transporter is given a command to obtain resources produced in a building, it waits at the building until the resource is available.

The behavior of both `ResourceCollectors` and `ResourceTransporters` can be changed in their `mInnerLoop`.

### 3.3.5 Class StaticGameObject

`StaticGameObject` serves as a common ancestor of the classes `Building` and `GameObstacle` extending `GameObject`; it sets the following member variable:

```
mObjectType = ObjectType.staticObject;
```

Any `GameObject` inheriting from this class is considered to be a static object – it does not move and there is no possibility that it would move out of its position.

### 3.3.6 Class Building

Buildings are represented by class `Building`. They have 3 main functions:

(f1) They are used to secure the player's position in certain areas – in such a case, we use `AttackingBuilding`, which contains an instance of `AttackAttributes`, as described in section 3.3.3.

(f2) They are used to produce entities. For this purpose, each building stores a list of entities it constructs (`ListOfObjsConstructedInBuilding mObjsCon-structedInBuilding`), which is filled with game entities by the method `ObjsConstructedInBuilding`. For example, in `LightVehicleFactory`, two types of units, a small tank (`SmallTank`) and a big tank (`BigTank`) can be created:

```
public class LightVehiclesFactory : Building {
     public override void ObjsConstructedInBuilding() {
           setGameObjConstructedInBuilding("BigTank");
           setGameObjConstructedInBuilding("SmallTank");
    }
}
```

*Code example from class LightVehiclesFactory, file LightVehiclesFactory.cs*

For details about identification of types in the example (`SmallTank` and `BigTank`), see section 3.2.

In order that the players do not have to pay too much attention to micromanagement of the construction process, they can place up to 10 units into the `ConstructionQueue mConstructionQueue`, and the units are then created one after another.

(f3) Buildings are also used as an important part of the resource management (for details about resource management, see section 3.4), as they accepts resources denoted as `mInputResources`. This variable is a Boolean function, so that a resource is accepted if and only if it has a non-zero value in `mInputResources`. Variable `mProvidableResources` stands for the resources the building provides; it is usually the result of resource transformation. The amount of resources the building currently stores is denoted as `mCurrentResources`.

The building might require to be placed on a tile containing a certain resource – for example, a *coal mine* has to be placed over the resource *Coal*. This is specified in `mRequiredUnderlyingResource`, where the amount of resources stands for the number of tiles with the resource required to be beneath the `Building`.

Resource transformation is proceeded using the `ResourcesTransformer mTransformer` (described in section 3.4.2). For instance, *steel mill* transforms 100 units of *Coal* and 100 units of *Iron* into 100 units of *Steel* – *Coal* and *Iron* are input resources, *Steel* is output resource. On the other hand, if we declare input resources to be zero, we create a building which produces resources – a mine, for instance, but the building may require to be placed on a certain resource type. This example shows that the design of resource management within buildings is easily extendible.

### 3.3.7 Obstacles

Obstacles have to have the following properties:

(1) they have to be present in the map

(2) game entities can not enter a tile where an obstacle is placed

(3) no structure can be built on a tile where an obstacle is placed

(4) obstacles may be removed using a special unit (for instance, in Original War, some obstacles may be removed by *bulldozer*)

The properties (1) – (4) are the properties of a `StaticGameObject`, if we allow the object to be destroyed just by special units. Therefore, we implemented `GameObstacle` as a class inheriting from `StaticGameObject`, but with a few rearrangements. Any `StaticGameObject` placed in the map may be identified by the ID of its player owner and its own ID (which is unique within the instance of the `Player` class). In order to identify the obstacle, we have to set its member variable `mPlayerID = -1`, and `mPlayerOwner` to `null`.

Removing an obstacle has to be implemented in the method `GameObsta-cle.receiveAttack`, and the unit removing obstacles has to be given `AttackAt-tributes`.

## 3.4 Resource Management

All types related to resource management are grouped into namespace `RTSGame.ResourceManagement`, apart from resource collectors and resource transporters, which are grouped in namespace `RTSGame.GameObjects`.

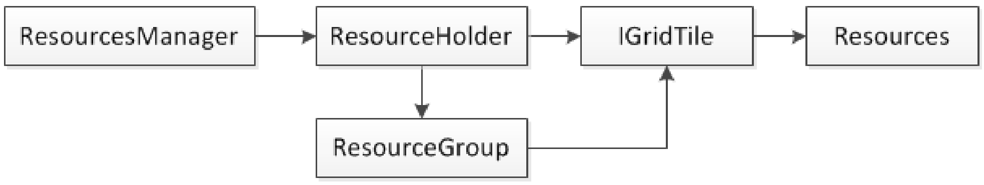


*Figure 23: Structure of resource management*

The structure of resource management is shown in Figure 23. Class `Resources` specifies the amount of resources of all resource types (defined in enum `ResourceTypes`). In the game, we define five resource types: *Money*, *Rubber*, *Coal*, *Iron*, and *Steel* (as we show in the following example).

```
enum ResourceTypes { Money, Rubber, Coal, Iron, Steel }
```

*Code example from file Resources.cs*

Let's now have a look at the chain of resource management. There are 2 classes of resources present in the game: (1) resources collectable from map, and (2) resources created in structures.

### 3.4.1 Resources present in map

Some types of resources are collected from the map – i.e. each tile (represented by interface `IGridTile`) of the map (for details about tiles, see section 2.5.2) has an instance of class `Resources`. If the amount of resource *R* is non-zero on the tile, then the resource *R* can be collected by a `ResourceCollectorObject` (for details, see section 3.3.4).

`Resources` are not present entirely separately in the map – they may be grouped in a `ResourceGroup`. It is a class which contains at least one tile with resources. For example, we want a type of resource to grow in bunches (in our game, it is *Rubber* – see Figure 24), so that the resource has to grow at adjacent tiles. For this reason, we use the `ResourceGroup` and put all *Rubber* in a bunch into the same `ResourceGroup`.

Class `ResourceHolder` (see its structure in Figure 25, where fields with white background contain abstract classes, gray background stands for real `ResourceHolders` used in the game) represents a resource type present in map (for each member of enum `ResourceTypes`, there is maximum one `ResourceHolder`, and it has to be registered in `ResourcesManager`). `ResourceHolder` contains information

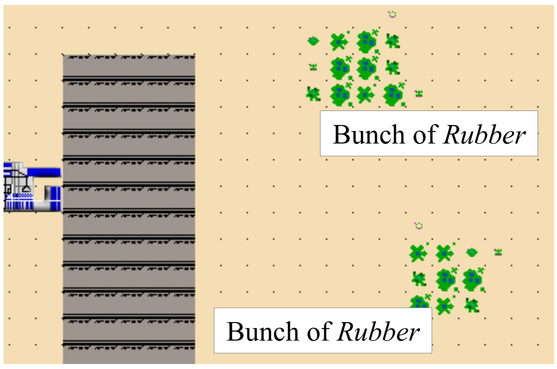


*Figure 24: Bunches of Rubber in our game*

about the nature of the resource such as whether the resource is an obstacle in map, if it is destroyed if run over, etc.

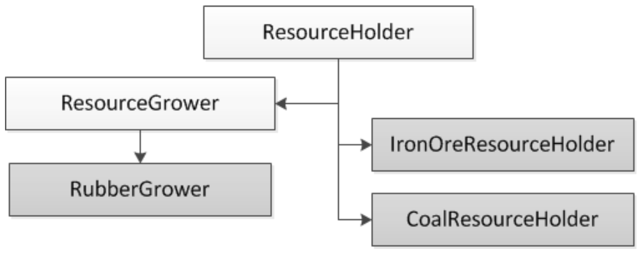


*Figure 25: Structure of class* `ResourceHolder`

`ResourceHolder` contains a list of all `ResourceGroups` (`mGroupsOfResources`) in which the given resource is stored, as well as a list of all tiles on which the resource is present (`mResourceFields`).

Class `ResourceGrower : ResourceHolder` represents growing resources (i.e. resources whose amount increases in time up to a certain limit). The increase of growing resource is defined as `mResourceIncrease` and their growing interval as `mGrowInterval`.

In our game, we use `IronOreResourceHolder` (see example bellow) for the resource *Iron*, `CoalResourceHolder` for the resource *Coal*, and `RubberGrower` for the resource *Rubber*.

```
public class IronOreResourceHolder : ResourceHolder {
      public IronOreResourceHolder() : base(ResourceTypes.Iron) {
            /* Iron ore is an obstacle */
            isObstacle = true;

            /* Its amount does not decrease while it is being collected */
            mDecreasesWhileHarvested = false;

            /* It will have orange-red color in the map thumbnail */
            mColorInMapThumbnail = Color.OrangeRed;

            /* It can not be destroyed if run over */
            isDestroyableByMovingGameObject = false;
      }

      /* Iron is given from the beginning and it does not appear at new tiles */
      public override void createNewResource() { }
}
```

*Code example from class IronOreResourceHolder, file ResourceHolder.cs*

Each `ResourceHolder` has to be registered in `ResourcesManager`:

```
ResourcesManager.registerResource(new IronOreResourceHolder());
ResourcesManager.registerResource(new CoalResourceHolder());
ResourcesManager.registerResource(new RubberGrower());
```

*Code example from class MainGame, file MainGame.cs*

### 3.4.2 Resources produced in structures

Not all resources have to be collected from the map – the chain of resource management also knows resources created in buildings (with or without any resource input). For such resources, there is class `ResourcesTransformer`.

Its method `transformResources` receives input resources (or nothing), calls the method `StartTransformation` and after `mTransformationLenght` of rounds, method `FinishTransformation` is called. This approach allows an easy extendability of the program with new `ResourceTransformers`. In `SteelMill` in our game, we define transformation of *Coal* and *Iron* into *Steel*, so that 100 units of *Iron* and 100 units of *Coal* are transformed into 100 units of *Steel*:

```
class CoalAndIronToSteelTransformer : ResourcesTransformer {
      public CoalAndIronToSteelTransformer() {
            mTransformLenght = 300;                 // number of rounds
            mInputResources = new Resources();
            mInputResources.setResource(ResourceTypes.Coal, 100);
            mInputResources.setResource(ResourceTypes.Iron, 100);

            mOutputResources = new Resources();
            mOutputResources.setResource(ResourceTypes.Steel, 100);
      }
}

/* building to proceed the transformation, it also stores the resources */
class SteelMill {
      Resources mOwnedResources;      // resources stored in the building
      ResourceTransformer mTransformer = new CoalAndIronToSteelTransformer();
      public override void transformResources() {
            /* the structure keeps the transformation result */
            mTransformer.transformResources(mCurrentResources);
      }
}
```

*Code example from classes CoalAndIronToSteelTransformer and SteelMill, file SteelMill.cs*

The transformation result may be also directly transmitted from the structure to resources of the player controlling the structure. Then, method `transformResources` would be the following:

```
public void transformResources () {
      /* this transmits the resources to the player (owner of this structure) */
      mTransformer.transformResources(mOwnedResources,  mPlayer.mResources);
}
```

*Code example from class SteelMerchant, file SteelMerchant.cs*

There are 3 possibilities of extending the behavior of this class:
(1) change `mInputResources` and `mOutputResources`
(2) overwrite methods `StartTransformation` and `FinishTransformation`
(3) overwrite method `transformResources`

## 3.5 Network communication

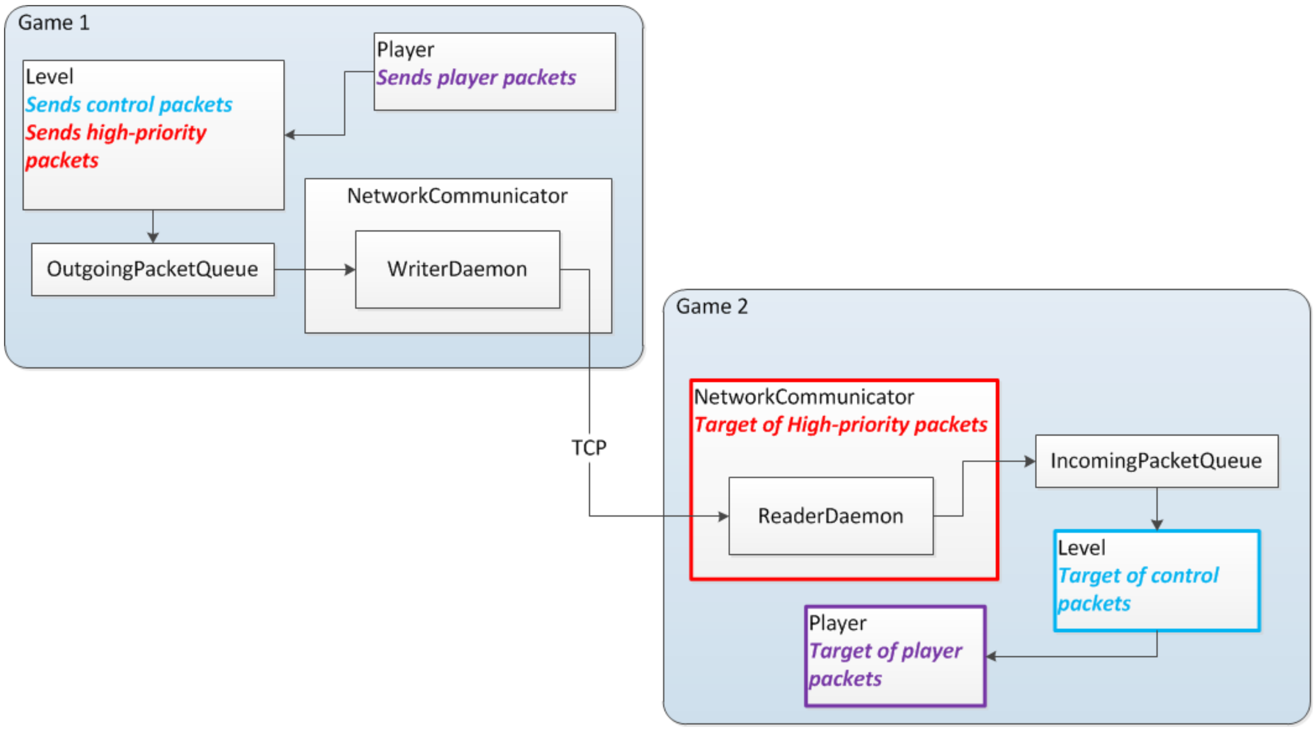


*Figure 26: Network communication – the path of one* `GamePacket`

Network communication (see Figure 26) is used for information exchange between communicating games. Types related to network communication are grouped in namespace `RTSGame.Networking`. We define a piece of information as an instance of class `GamePacket`, which represents one command (for details about commands, see section 2.6). The `GamePacket` is sent from *Game 1* to *Game 2* using synchronous communication and the TCP protocol (as declared in section 2.6).

There are 3 types of `GamePacket` within the system of network communication:

(1) ***High-priority packets***

(2) ***Control packets***

(3) ***Player packets***

They have different roles and they are also handled differently:

***(1)*** High-priority packets (e.g. a ping, which checks if the game has not been disconnected) have to be evaluated in the round they arrive, so that they are dealt with by the `NetworkCommunicator` in its method `processHighPriority-GamePackets`.

***(2)*** Control packets, produced and evaluated by `Level` in its method `processOne-Packet`, carry information about the game progress, but they do not contain any specific commands (for information about commands, see section 2.6) for entities (for example, a `StartGamePacket` carries the information that the game has to be started from server to clients).

***(3)*** Player packets, produced and evaluated by `Player`, carry information from players for players or for entities controlled by the player. `Player` reads the commands for the current round from `mPacketsForCurrentTurn` and evaluates them in method `evaluateGamePackets`. For example, player packet `TurnDonePacket` carries information that the sender-player has finished the current round, a `MoveToGamePacket` carries a move-to command for an entity.

`WriterDaemon` and `ReaderDaemon` are separate threads operating over the `OutgoingPacketQueue` and `IncomingPacketQueue`, respectively. They regularly check if there are any data to be written to the stream or to be read from it.

`OutgoingPacketQueue` is implemented as a `GamePacketLindedList`, which is a sorted queue of `GamePackets` using a double-linked list; method `AddLast` inserts `GamePackets` in sorted way according to their `mOrdinalNumber`, and it iterates the sorted queue from behind (from highest `mOrdinalNumbers`) because it is very likely that the `GamePackets` arrive in a sequence similar to an increasing sequence and our experiments have shown that inserting `GamePackets` this way saves a considerable amount of time.

`IncomingPacketQueue` is implemented as `GamePacketQueue`, which is a data structure for storing incoming packets implemented as a linked list of `GamePacket-LinkedLists`. In each of the `GamePacketLinkedLists`, it stores packets from one round (in a sorted way) to allow quick access to them. This solution is advantageous because at any moment, we only store packets from a few rounds in advance due to the communication offset (presented in section 2.6.1). Compared to a simple sorted queue of `GamePackets`, this data structure saved up to 75% of time needed to access the `GamePackets` stored in it.

### 3.5.1 Network topology

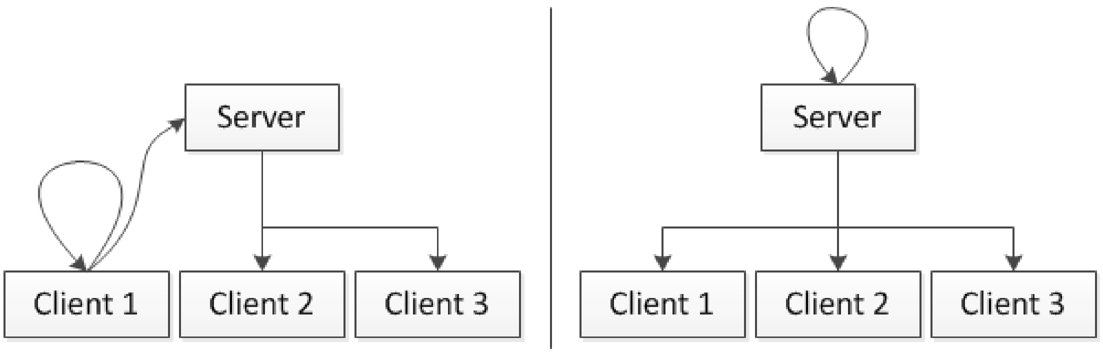


*Figure 27: Network topology. Left: Client 1 sending `GamePacket` to all games. Right: Server sending `GamePacket` to all games.*

The network has a star topology (shown in Figure 27) with one central point (server). If a client game is sending a `GamePacket` to all other games, it sends the `GamePacket` to itself and to the server, which forwards it to all clients apart from the sender (left part of Figure 27). If server sends a `GamePacket`, it sends it to itself and to all clients (right part of Figure 27).

Server game is using class `Server : NetworkCommunicator`, client games use `Client : NetworkCommunicator` as the `NetworkCommunicator` shown in Figure 26.

Server game can choose from different sending options for each `GamePacket` (stored in `enum SendingOptions`), so that the `GamePacket` can be sent to one client only (`SendToOne`), to all but the server (`ServerForwarding`), or to all and to the server (`BroadcastAndToMe`). The option `Broadcast` is used by client games.

### 3.5.2 Server and client communication

#### Initialization

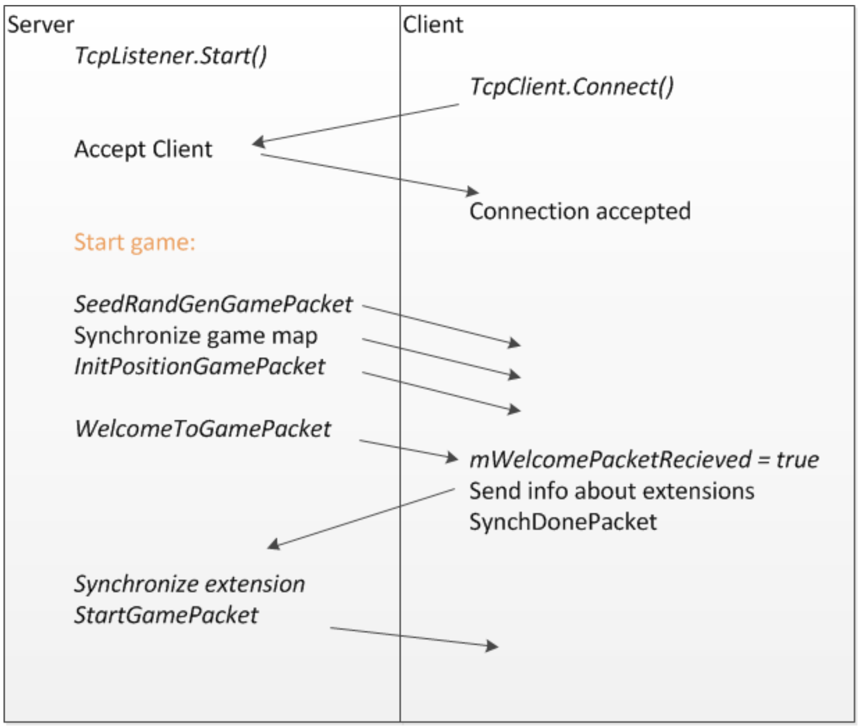


*Figure 28: Establishing connection between server and client games*

While creating the connection (see Figure 28), users choose one game to be the server game, which then listens for connections from all client games. Once all clients are connected, the server game initializes the network session with `SeedRandGenGamePacket` (seeds all random number generators in client games). Then, the game synchronizes the map using `DefineMapGamePacket`, `ObstacleGame-Packet`, and `ResourceGamePacket`, which carry the information about the map size, obstacles in the map, and resources present in the map. Then, the server sends `InitPositionGamePacket` (tells the games initial positions of all players in the game), and `WelcomeToGamePacket`, which tells client games:

- that the game has just been launched
- ID of the local player playing the client game
- total number of players in the game

Upon receiving the `WelcomeToGamePacket`, each client informs the server about the extensions available to the client (using `TypeGuidPacket`) followed by `SynchDonePacket`, which in turn informs the server that the list of extensions available to the client has been sent. After all client games have finished the synchronization, the server determines the set intersection of the extensions and informs the games about the extensions which are to be used in the game (using `TypeGuidPacket`). Then, it starts the game with a `StartGamePacket`.

### Communication during the game

The game progress is subdivided into rounds (for details, see section 2.1); in each round, each player does one turn, after which, a `TurnDonePacket` is sent to inform other games in the network that the current player's turn has been finished. After all players have sent their `TurnDonePacket`, the round is finished and next one can be started.

In section 2.6.1, we introduced communication offset *CO*, so that if round *R* has been finished, round *R* + *CO* can be started. In Figure 29, we show an example for *CO* = *3*. *Game 1* has finished its turns in rounds *N*, ..., *N* + *3* and is waiting for *Game 2* to finish its turn in round *N* + *1*. Once *Game 1* has received the `TurnDonePacket` from *Game 2* in round *N* + *1*, it will be allowed to enter round *N* + *4*.

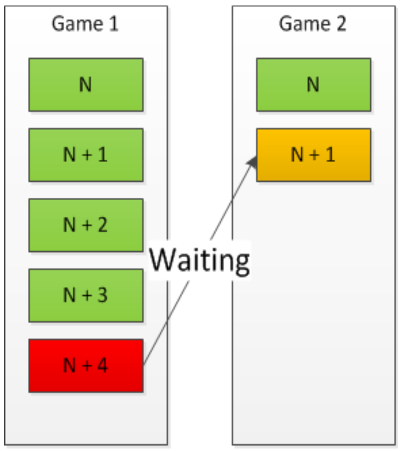


*Figure 29: Game 1 has finished rounds N, ..., N + 3 and waits for Game 2 to finish round N + 1*

### Loss of connection

There are 2 possible scenarios related to the loss of connection.

(1) If client game looses connection to the server, the game is terminated.

(2) If server looses connection to a client game, the player at the client game is considered to be dead and the game continues without this player.

The disconnection of a client game is announced by `PlayerDisconnectedGamePacket`, which is sent to the server and then forwarded to all (connected) client games.

If the server waits too long for a `TurnDonePacket` from a client game, it starts sending `PingGamePackets` to it to find out if the game is still reachable, and if it finds out that the connection to the game has been lost, it disconnects it sending a `PlayerDisconnectedGamePacket` to all other clients.

### Identification of types

The command to create an entity with AI for its internal intelligence is represented by `UnitWithAITypePointGamePacket` with the following variables:

```
int mUnitTypeIndex
int mAITypeIndex
```

The variables stand for indices into `AssemblyManager.mIndexedListOfTypes`, from which, the type of the entity and the type of its AI is taken. For this reason, `AssemblyManager.mIndexedListOfTypes` has to contain exactly the same list of types in the same order at all communicating games, which is processed in the synchronization phase of the initialization of the network communication.

### 3.5.3 GamePacket

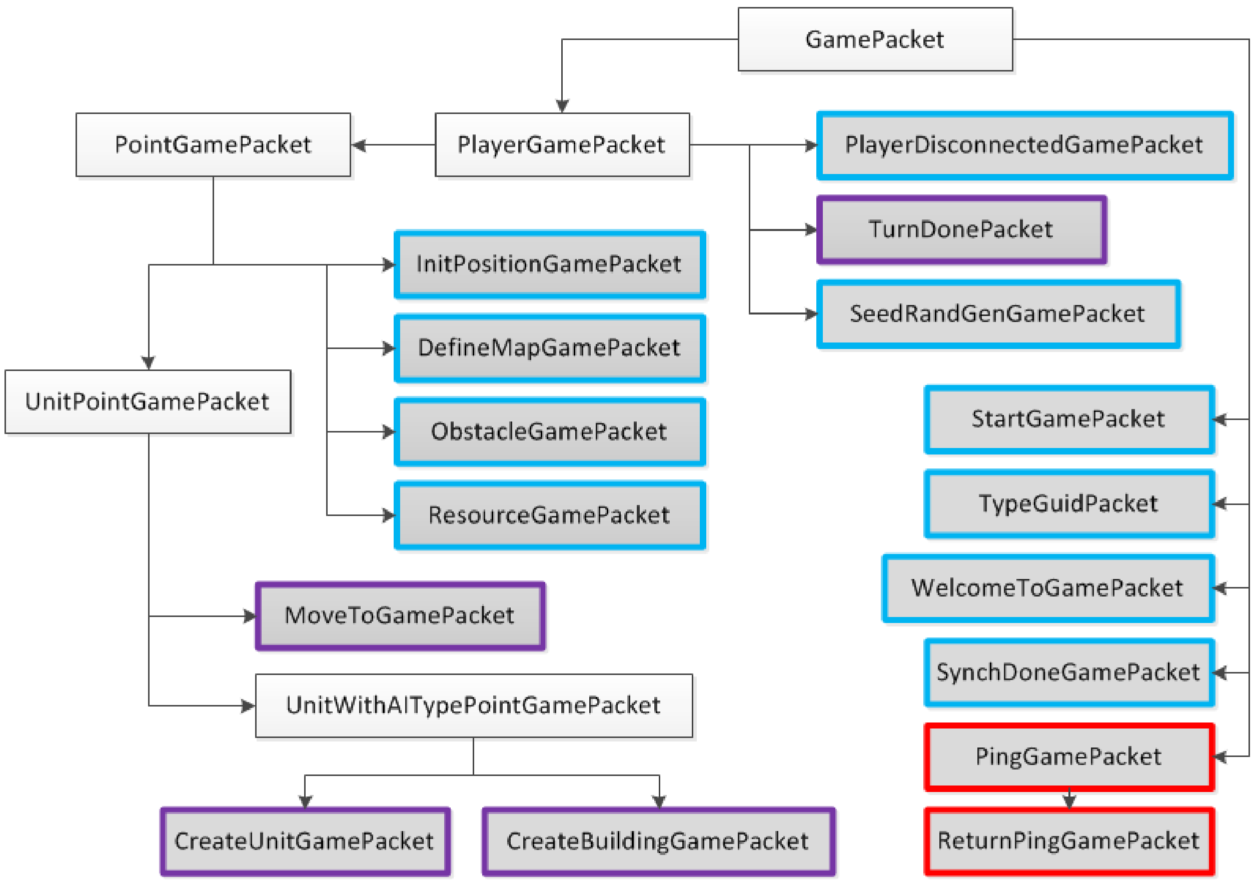


*Figure 30: Types extending* `GamePacket`

In Figure 30, we see types extending the `GamePacket` class. ***Red border*** stands for ***high-priority packets***, ***blue border*** denotes ***control packets*** and ***violet border*** means ***player packets***. Fields with white background contain abstract classes, gray background stands for real packets used in the game.

### Class GamePacket

This class represents one command sent via network to be executed in a certain round (denoted by `mTimeStamp`). The information contained in `GamePacket` is encoded (transformed into byte-array) and decoded (transformed from byte-array) using methods `Encode` and `Decode`, respectively. In inheriting types which add a new variable to the set of information sent over network, these methods have to be overridden – e.g., class `PlayerGamePacket` adds variable `Value` to `GamePacket`:

```
public class PlayerGamePacket : GamePacket {
      public int mPlayerID;
      public override int Encode() {
            int nextIndex = base.Encode();
            nextIndex = encodeNumberIntoArray(mPlayerID, byteCode, nextIndex,
                                              EncodingType.encodingAsInt);
            return nextIndex;
      }
}
```

*Code example from class PlayerGamePacket, file GamePacket.cs*

Method `encodeNumberIntoArray` accepts parameter `EncodingType`, which determines the number of bytes the variable needs in the `mByteCode`. The same `EncodingType` has to be used for decoding the variable from the `mByteCode`:

```
mPlayerID = decodeFromArray(out nextIndex, mBinaryData, nextIndex,
                                                  EncodingType.encodingAsInt);
```

*Code example from class PlayerGamePacket, file GamePacket.cs*

If a `GamePacket` is being encoded into byte-array and decoded from it, it needs to know the resulting length of the information (the number of bytes) contained in it (`mNumberOfBytes`), because the length of it varies across different types of `GamePackets`. This information is stored in static dictionary `GamePacket.mDictOf-InformationLenght`, which is dynamically created during the initialization phase of the game by static method `GamePacket.initializeDictOfInformationLenght`. From this dictionary, each `Type` can easily read the length of the information contained in it.

The `GamePacket` has a property `mOrdinalNumber`, which is its ordinal number within the scope of one player's turn (i.e. unique for player and round). This property is used to control that the packets are received in right order.

### Identification of GamePacket

In order to identify the `GamePacket` over the network from its binary representation, it has to have an *identification value*, which is unique for each `Type` of `GamePacket`. The *identification value* is generated by method `Type.GUID.GetHash-Code`, and we assume that the value is unique within the scope of the game.

To convert from the *identification value* to `Type`, we create static `Dictionary<int, Type> mDictionaryOfGamePacketTypes` in the initialization phase of the game, and the *identification value*, called `mIndexIntoTypesArray`, is used as the key of the dictionary.

After a `NetworkCommunicator` receives a `GamePacket` in its binary form, it first reads the first 4 bytes and converts them into the *identification value*. Then, it converts the identification value into `Type` (using static method `GamePacket.getGamePacketTypeFromIndex`) and uses the `Type` to create the `GamePacket` using static method `GamePacket.createPacketFromType`.

### 3.5.4 Types extending GamePacket

Let us now briefly describe types extending the `GamePacket` class. These are simple types only adding a piece of information to their ancestors, which has to be sent over the network.

**Class PlayerGamePacket**

Contains integral value `mPlayerID`, which stands for the ID of a player.

**Class PointGamePacket**

Contains integral values `mPlayerID`, which stands for the ID of a player, and a `Point`.

**Class UnitPointGamePacket**

Contains integral values `PlayerID` and `UnitID`, `Point`, and index into `AssemblyManager.mIndexedListOfTypes`, which denotes the type of unit.

**Class UnitWithAITypePointGamePacket**

Contains integral values `PlayerID` and `UnitID`, `Point`, and two indices into `AssemblyManager.mIndexedListOfTypes`, which denote the type of unit and the type of AI for it.

**Class MoveToGamePacket**

Represents a move-to-position command to a unit denoted by `PlayerID` and `UnitID`.

**Class CreateBuildingGamePacket**

Represents a command to building denoted by `PlayerID` and `UnitID` to create a building of type denoted by `mUnitTypeIndex` with internal AI denoted by `mAITypeIndex` (both indices into `AssemblyManager.mIndexedListOfTypes`).

**Class CreateUnitGamePacket**

Represents a command to building denoted by `PlayerID` and `UnitID` to create a unit of type denoted by `mUnitTypeIndex` with internal AI denoted by `mAITypeIndex` (both indices into `AssemblyManager.mIndexedListOfTypes`).

**Class TurnDoneGamePacket**

Used to inform all games that the player denoted by `PlayerID` has finished the round in which the packet has been sent.

**Class PlayerDisconnectedGamePacket**

Sent to inform all games that the player denoted by `PlayerID` has been disconnected from network communication.

**Class StartGamePacket**

This `GamePacket` is sent by server at the very beginning of the game to inform network clients that the server has just started the game. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class PingGamePacket**

This high-priority packet is used by server to find out whether a network client is still connected in case the server is waiting too long for a response from the client.

**Class ReturnPingGamePacket**

This high-priority packet is used to response to a `PingGamePacket`.

**Class WelcomeToGamePacket**

Used for initializing the game and it is only sent by server. The packet contains player-specific information, for example ID of the receiving player, the number of other players playing the game, etc. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class InitPositionGamePacket**

This packet initializes the positions of all players in the network. Server has to inform all players about all positions, because it is essential for the synchronization of the game. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class DefineMapGamePacket**

This packet is used in the initialization phase of the game to inform all clients about the size of the map. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class ObstacleGamePacket**

This packet informs all clients about an obstacle present in the map. It does not use an ordinal number because it is sent in the initialization phase of the game.

**Class ResourceGamePacket**

This packet informs all clients about a resource present in the map – it carries an index into the `enum ResourceTypes` (`mIndexIntoTypesEnum`), index into the list of `ResourceGroups` held by the `ResourceHolder` to indicate the `ResourceGroup` into which the resource belongs (`mIndexIntoListOfGroups`), and its amount (`mAmount`). This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class SeedRandGenGamePacket**

This packet is used for sending the seed of random numbers generator to all games connected in network, which is essential for the synchronization of the game. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class TypeGuidPacket**

Used for synchronizing extensions available to communicating games. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

**Class SynchDoneGamePacket**

Used to inform server or client that the synchronizing process has been finished. This packet does not use an ordinal number because it is sent in the initialization phase of the game.

### 3.5.5 Logging of player packets

If the value of `Globals.LOG_GAME_PACKETS` is set to `true`, the game logs player packets from communication between games in network (i.e. commands issued by players: `MoveToGamePackets`, `CreateUnitGamePacket`, `CreateBuildingGamePacket`, and `TurnDoneGamePacket`) into log files in the root directory of the game. The logs can be compared using method `LogsComparerTool.compareGamePacketLogs`, which is encapsulated by a command-line application for comparing logs in project `LogsComparer.csproj`.

Comparing logs of player packets is important to verify that the network communication is synchronized, for example after a new strategy for computer players has been added into the game.

### 3.5.6 Position of computer players in the network

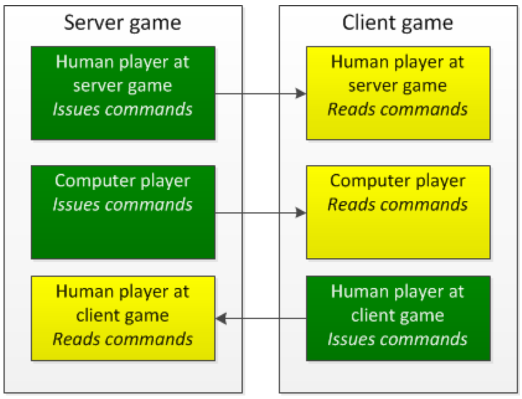


*Figure 31: Position of players in network. Green fields stand for players issuing commands, yellow fields represent players reading commands.*

In Figure 31, we see an example of communication between a server game and a client game with altogether 3 players – two human and one computer player. All computer players are always accommodated by the server game, where their decision-making is proceeded. They issue commands (as if they were human players) and client games read them. On the other hand, commands from the human player at client game are issued by this player, and they are read (and evaluated) by the server game.

## 3.5.7 Use of random numbers in the game

In section 2.6, we described the system of sharing information between games as sharing commands (i.e. `GamePackets`). In order to keep the game synchronized, it is necessary to provide the same evaluation of the commands at each of the game, which may be complicated if want to use random numbers. We will now describe the rules for using random numbers in the game.

There are 2 types of random numbers in the game, as we show in Figure 32:

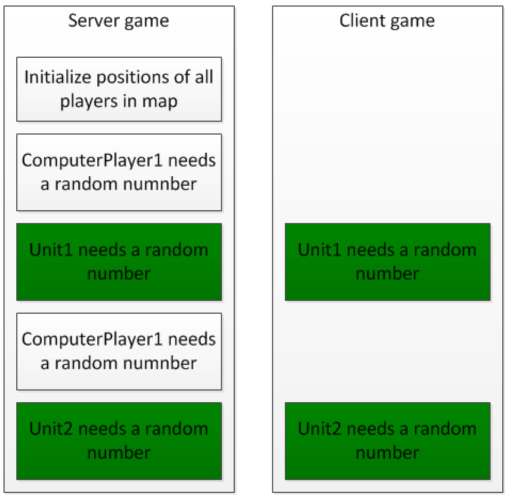

Server game
Initialize positions of all players in map
ComputerPlayer1 needs a random numnber
Unit1 needs a random number
ComputerPlayer1 needs a random numnber
Unit2 needs a random number
Client game
Unit1 needs a random number
Unit2 needs a random number

*Figure 32: Use of random numbers. Notice that those represented by green fields have to be synchronized.*

(1) **Random numbers used in one instance of the game only** (white background in Figure 32). For example, in the beginning of the network session, the server initializes the position of all players in the map – it is only done by the server, so that these numbers can not be synchronized with other games (and the server then sends `InitPositionGamePacket` to all clients to inform them about the positions of players). Another example is the reasoning of a computer player. If the player wants to issue any command, it sends a `GamePacket` to all games in the network, as if it was a human player. Any decision of the computer player based on random numbers does not need the synchronized random numbers, because this decision is made at one of the communicating computers only.

(2) **Random numbers used in all instances of the game at the same time** (green background in Figure 32). In contrast to computer players, entities make their decision at all communicating games and need to obtain the same random number at all of the games at the same time. For example, *Unit1* collides with another entity, and chooses a random interval to wait before it runs the pathfinding algorithm again (this behavior is described in section 2.5.5.2). As we show in Figure 32, the unit needs the random number both at server and at client game. If it obtains a different random number at different games, the game looses its synchronization and is very likely not to regain it again.

In order to obtain a synchronized random number, the entity has to use static method `Level.nextRand`, which provides random numbers from seeded random numbers generator (seeded by `SeedRandGenGamePacket` in the initialization phase of the multiplayer game).

If the value of `Globals.LOG_RANDOM_NUMBERS` is set to `true`, the game logs random numbers generated by the method `Level.nextRand` into log files in the root directory of the game. These logs can then be compared using method `LogsComparerTool.compareRandNumbersLogs`, which is encapsulated by a command-line application for comparing logs in project `LogsComparer.csproj`.

## 3.5.8 Project LogsComparer.csproj

As we already mentioned in sections 3.5.5 and 3.5.7, project `LogsComparer.csproj` is used for comparing logs of random numbers with each other and logs of player packets with each other.

The project is set up to compile into the root directory of the RTSGame (`RTSGame\RTSGame\bin\x86\XXX\`, where `XXX` stands for `Debug` or `Release` – depending on settings of the project `RTSGame.csproj`) – as a console application `LogsComparer.exe`, which accepts 3 commands:

(1) `QUIT` terminates the applications

(2) `RANDNUM file1 file2 file3 … fileN` compares logs of random numbers with names *file1*, *file2*, *file3*, …, *fileN* using method `LogsComparer-Tool.compareRandNumbersLogs`.

(3) `PACKETS file1 file2 file3 … fileN` compares logs of player packets with names *file1*, *file2*, *file3*, …, *fileN* using method `LogsComparer-Tool.compareGamePacketLogs`

The application either writes message "*Logs OK*" to state that the logs are equal, or indicates the first occurrence of an error in the compared log files.

## 3.6 Artificial intelligence

The artificial intelligence in our game, implemented with the use of a finite state machine (for details, see section 2.3.4.2), is used for two purposes: for artificial intelligence of computer players and for the autonomous behavior of entities (details in section 2.4.1). All types related to the artificial intelligence in the main project, `RTSGame.csproj`, are stored in namespace `RTSGame.FSM`.

We also provide sample extensions of artificial intelligence in projects `FSM_Player_AI.csproj` (artificial intelligence for players) and `FSM_AI.csproj` (artificial intelligence for entities).

The extensions of AI for computer players have to be inserted into directory `.\PlayerAI`, extensions of AI for entities have to be inserted into `.\AI` in form of .dll files. The projects `FSM_Player_AI.csproj` and `FSM_AI.csproj` fulfill these conditions, because they are set up to compile as a DLL files to directories `RTSGame\RTSGame\bin\x86\XXX\PlayerAI\`, and `RTSGame\RTSGame\bin\x86\XXX\AI\`, respectively, where `XXX` stands for `Debug` or `Release` (depending on settings of the project `RTSGame.csproj`).

### 3.6.1 Class FiniteStateMachine

This is the main class for the finite state machine. It contains a list of states represented by class `FSMState` (identified within the `FiniteStateMachine` through their unique `StateId`), which are inserted into the FSM with the method `Insert`. The `FiniteStateMachine` changes the states in its method `Update` (see Figure 33), calling the method `FSMState.Update` of the current state `mCurrentState`, which returns a reference to the next `FSMState` or null, if there is no suitable transition.

In order to simplify the design of the FSM, we allow it to change the states as long

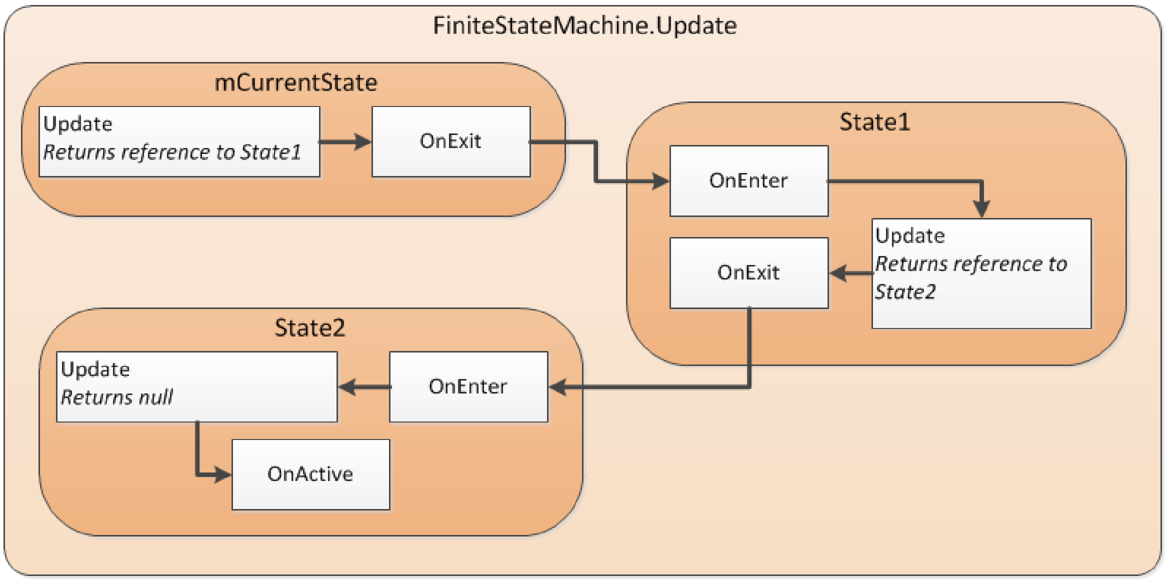


*Figure 33: Example of an Update of FSM, which jumps from* `mCurrentState` *over State1 to State2*

as the change is possible – i.e., as long as the method `mCurrentState.Update` returns a non-null value. In Figure 33, the `FiniteStateMachine.Update` starts in `mCurrentState` and jumps over the `State1` into `State2`, i.e., method `mCurrent-`

`State.Update` finds a transition into `State1`, and `State1.Update` finds a transition into `State2`. From `State2`, no suitable transition is possible, so that the method `State2.Update` returns `null`, and the `Action State2.OnActive` is performed.

In case of a wrong design of the FSM, an infinite loop may occur, which we prevent by using the variable `jumpsRemaining`, which can be set to any integral number (or -1 which means that there is no upper bound for the number of jumps in one `FiniteStateMachine.Update`).

The `FSMState` contains 3 important instances of type `Action`:

- Action `OnEnter`, which is performed if the state is being entered
- Action `OnExit`, which is performed if the state is being exited
- Action `OnActive`, performed if the FSM remains in the state after the `FiniteStateMachine.Update` method, processes the action which is to be done by the artificial intelligence (either an action of the player or an action of an entity, such as keep moving, attack an entity, etc.)

### 3.6.2 FSMState

In our program, we use 2 successors of the abstract class `FSMState`:

(1) `ExplicitTransitionFSMState` with explicitly given transitions to other states.

(2) `ImplicitTransitionsFSMState` with implicitly given transitions to other states.

#### Explicit transitions

The state with explicit transitions uses the `Action OnUpdate`, which is called in its `Update` method.

```
public class ExplicitTransitionFSMState : FSMState {
	public Func<FSMState> OnUpdate { get; set; }
	public override FSMState Update() { return OnUpdate(); }
}
```

*Code example from class ExplicitTransitionFSMState, file ExplicitTransitionFSMState.cs*

An example of possible use of the `Action OnUpdate` can be found within the class `FSM_AI_Player_NoActivity`, where we use it for the `initialState`:

```
stateInitial.OnUpdate = () => {
	if (rubberMerchantGroup.Count == 0) return stateCreateRubberMerchant;
	if (harvestersGroup.needsToConstructMoreUnits()) return stateCreateHarvesters;
	if (getLVF() == null) return stateCreateLVF;
	/* etc. … */
	return null; // this says: stay in this state and go on to OnActive
};
```

*(Slightly modified) code example from class FSM_AI_Player_NoActivity, file FSM_AI_Player_NoActivity.cs*

`ExplicitTransitionFSMState` may be very useful, if the transition function is easy to understand. For more complicated transition functions, we use `ImplicitTransitionsFSMState`.

#### Implicit transitions

To represent implicit transitions between states, we use class `ImplicitTransi-`

tionsFSMState.Transition. In the method ImplicitTransitionsFSMState.Update, the FSMState iterates over all its Transitions (stored in List<Transition> mImplicitTransitions) until it reaches a transition whose condition is met (i.e. the Func<bool> Condition returns true), which is taken (Action OnFollow is performed before the method ImplicitTransitionsFSMState.Update returns FMSState NextState).

A transition is added to a state using method AddTransition. In the FSM for FSM_MovingObject, we set the transition from stateMoving to stateWaiting as follows:

```
protected ImplicitTransitionsFSMState stateMoving;

/* Transition into stateWaiting if the condition Obj.mPathFinished is met. Set varia-
ble Obj.mPathFinished to false while taking this transition: use Action OnFollow.*/
stateMoving.AddTransition(stateWaiting,  () => Obj.mPathFinished,
                                         () => Obj.mPathFinished = false);
```

*Code example from class FSM_MovingObject, file FSM_MovingObject.cs*

### 3.6.3 Initialization of FSM

The class FiniteStateMachine is initialized by its method initialize, (called by the constructor of the FSM), which initializes all states of the FSM and defines transitions between them. This approach, combined with the use of ImplicitTransitionsFSMState, allows easy extendability of the FSM, which we will explain using the following example:

In section 3.3, we presented a MovingGameObject, which is able to move in the map, and AttackingMovingGameObject, which extends the MovingGameObject with the ability to attack other entities.

We use FSM_MovingObject for the AI of MovingGameObject, and FSM_AttackingMovingGameObject for AI of AttackingMovingGameObject:

```
public class FSM_MovingObject : FSM_DoNothing {
     public override void initialize(GameObject aObj) {

          /* Run-time check of the argument */
          MovingGameObject Obj = aObj as MovingGameObject;
          if (Obj == null) throw new ArgumentException();

          base.initialize(Obj);

          stateMoving = new ImplicitTransitionsFSMState("stateMoving");
          Insert(stateMoving);

          stateWaiting.AddTransition(stateMoving,
                                     () => Obj.needsNewPath(),
                                     () => Obj.resetHowLongWaitForFreePath());
          stateWaiting.OnActive = () => Obj.resetPath();
     }
}
```

*Code example from class FSM_MovingObject, file FSM_MovingObject.cs*

The following piece of code shows how we easily extend the state `stateWaiting` from the previous code with new transition in class `AttackingMovingGameObject` (inheriting from `FSM_MovingObject`):

```
public class FSM_AttackingMovingGameObject : FSM_MovingObject {
     public override void initialize(GameObject aObj) {

           /* Run-time check of the argument */
           AttackingMovingGameObject Obj = aObj as AttackingMovingGameObject;
           if (Obj == null) throw new ArgumentException();

           base.initialize(Obj);
           stateWaiting.AddTransition(stateAttacking,
                       () => Obj.mAttackAttributes.findRandomEnemyObjectInRange());
     }
}
```

*Code example from class FSM_AttackingMovingGameObject, file FSM_AttackingMovingGameObject.cs*

Notice the run-time checks of the argument for method `initialize`. This is due to the fact that it is not possible to provide a more specific argument to the method `initialize` in a more specific successor of `FiniteStateMachine` (for example, in `FSM_MovingObject`, we want to use `initialize(MovingGameObject aObj)`, which is not supported by the language specification).

### 3.6.4 Types extending FiniteStateMachine

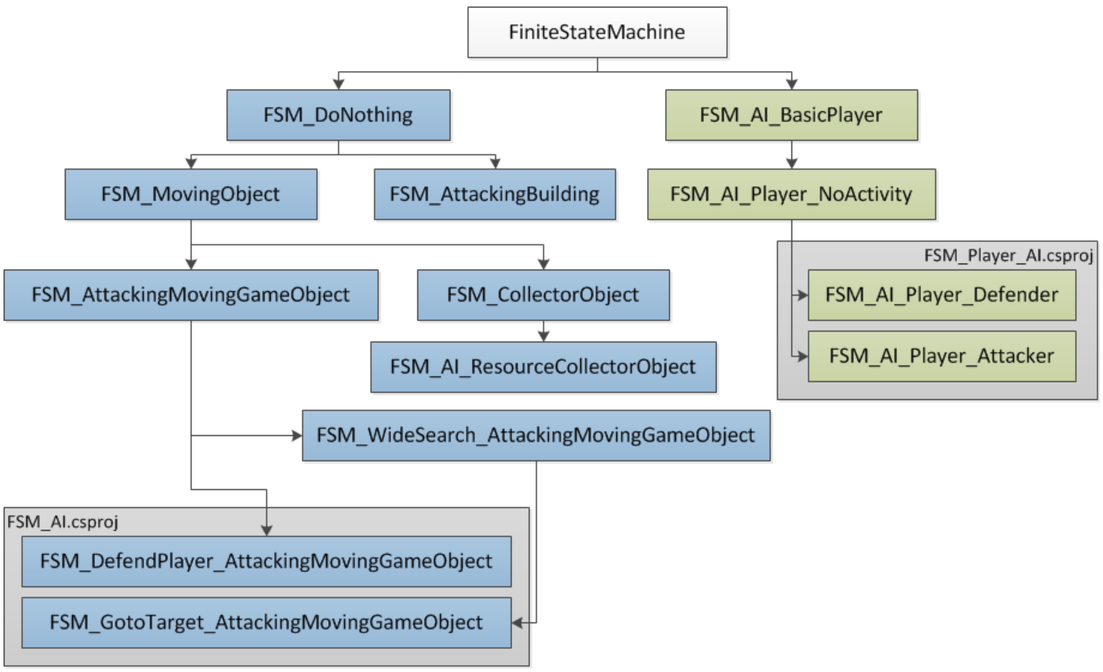


*Figure 34: Types extending `FiniteStateMachine`.*

In Figure 34, we show the tree of types extending `FiniteStateMachine`. Fields with blue background represent the AI for entities, whereas green background stands for AI for computer players.

### 3.6.5 AI for entities

The game provides a variety of AIs for entities, with different degrees of independence and with different use – some of them are supposed to be used for entities created by human players and some of them are meant for entities created by computer players. We will now describe them briefly while explaining their usage.

**Class FSM_DoNothing**

Being the default value for `GameObject.mInnerLoop`, this class represents internal intelligence which does not perform any activity – it only has state `stateWaiting`, in which it stays idle.

**Class FSM_MovingObject**

This class, whose use is restricted to `MovingGameObject` (and its successors), implements the internal intelligence of a moving entity. It extends `FSM_DoNothing` with 2 states: `stateMoving`, in which it processes the movement, and `statePathSearching`, in which it searches for path in map (i.e. runs a pathfinding algorithms – for details, see section 2.5.4).

Notice that this class also extends the behavior of the `stateWaiting` defined in `FSM_DoNothing` with a transition to `stateMoving` and with redefinition of `stateWaiting.OnActive`:

```
stateWaiting.AddTransition(stateMoving, () => Obj.needsNewPath(),
                                         () => Obj.resetHowLongWaitForFreePath());
stateWaiting.OnActive = () => Obj.resetPath();
```

*Code example from class FSM_MovingObject, file FSM_MovingObject.cs*

**Class FSM_AttackingMovingObject**

This class, whose use is restricted to `AttackingMovingGameObject`, implements the behavior of a `MovingGameObject` which is able to damage other entities (it has `AttackAttributes` – for details, see section 3.3.3). It extends the `FSM_MovingObject` with `stateAttacking`, in which, it performs the attack.

If an entity using this AI (in state `stateWaiting`) finds an enemy entity within its range of attack (specified in `AttackAttributes`), it starts to attack it.

In our game, this class is used for types `BigTank` and `SmallTank` (stored in `Tanks.csproj`) created by human players.

**Class FSM_DefendPlayer_AttackingMovingGameObject**

This extension of `FSM_AttackingMovingObject` is used for some units produced by computer players – it has an enhanced internal intelligence for autonomous behavior (its degree of independence is higher than the one of `FSM_AttackingMovingObject` – for details, see section 2.4.1), because it autonomously searches for intruders within the computer player's base and attacks them. In order to save system resources, it does so only once upon a time (`mCheckInterval`). The higher degree of independence of this AI produces an enhanced impression of the intelligence of computer players using units with this internal intelligence (for discussion about the impression of computer players' intelligence, see section 2.3).

**Class FSM_WideSearch_AttackingMovingGameObject**

This class extends the behavior of `FSM_AttackMovingObject` with the ability to search for enemy entities within a broader range than only the range of attack (specified in `AttackAttributes`) and attack them if in `stateWaiting`. Although this is not a great change in the behavior of attacking entities, it increases the impression of the owning player's intelligence, so that this FSM may be used for some units created by computer players.

**Interface IPointStateMachine**

This interface defines one point, `mTarget`, allowing the FSM implementing it to include this specific point into its behavior.

**Class FSM_GotoTarget_AttackingMovingGameObject**

This class, implementing the interface `IPointStateMachine`, makes the attacking unit to protect the point `mTarget` specified in the interface. It fixes the point so that any time the unit is in `stateWaiting`, it returns to it. This is especially useful for units produced by computer players to protect the base. This behavior of entities also enhances the impression of intelligence of their owners – computer players.

**Class FSM_AttackingBuilding**

This class implements the behavior of a static entity which attacks enemy entities coming into its range of attack (defined in `AttackAttributes`).

**Class FSM_CollectorObject**

This class implements the behavior of a resource collector and resource transporter (described in section 3.3.4), which should be used for resource collectors and transporters created by the human player.

**Class FSM_AI_ResourceCollectorObject**

This class implements the behavior of resource collector and resource transporter, which is supposed to be used for units created by computer player, because it extends the autonomous behavior of `FSM_CollectorObject` so that if the collector stands in `stateWaiting`, it starts to search for new resource again – thus being more independent and enhancing the impression of computer player's intelligence.

### 3.6.6 Defining FSM for GameObject

As we already explained in section 3.2, due to the run-time extendability of the game, the compiler may not know about the existence an FSM at compilation time. For this reason, it can be added to entities using the `AddFSMAttribute`:

```
[AddFSM("FSM_AttackingMovingGameObject")]
    public abstract class AttackingMovingGameObject : MovingGameObject
```

*Code example from class AttackingMovingGameObject, file AttackingMovingGameObject.cs*

The attribute defines the given FSM for the `GameObject` and for all types inheriting from it, unless they are given another `AddFSMAttribute`.

### 3.6.7 AI for computer players

In the game, we provide 4 predefined AI strategies for computer players. As we described in section 2.3.1, we allow the AI for computer players to cheat. On one hand, the AI has access to the whole game via `AIPlayer.mLevelOwner` and it is only up to the AI designer which information will be used, on the other hand, in section 3.6.5, we described AIs for entities with an increased degree of independence, which are meant to be used for entities created by computer players. The independent behavior of entities provides an enhanced impression of their owner's (computer player's) intelligence. Furthermore, computer players receive more resources in the beginning than human players. Let's now describe the strategies for computer players.

**Class FSM_AI_BasicPlayer**

Class `FSM_AI_BasicPlayer` serves as the frame for all other AI strategies while providing methods which implement the basic functionality of the AI for the player such as construction of units and buildings, giving commands to units, searching for buildings of specific types etc.

**Class FSM_AI_Player_NoActivity**

This class stores a (cheating) intelligence of computer players able to collect resources (in this case, the resource *Rubber*), to defend themselves and to create units to attack other players.

In order to make the management of entities easier for the designer of AI, we use class `GroupOfEntities`, which stores entities of the same `Type` and with the same internal AI. The player can set the maximum number of entities stored in the `GroupOfEntities` (`mMaxNumberOfEntities`), the minimum acceptable number of them (`mMinAcceptableNumberOfEntities`), the `Type` of the entities (`mTypeOfEntity`) and the `Type` of the AI for them (`mTypeOfFSM`).

For each `GroupOfEntities`, the AI checks if there are enough (i.e. more than `mMinAcceptableNumberOfEntities`) entities, and if not, it starts creating entities of the given type while going into the state where the units are created.

In `FSM_AI_Player_NoActivity`, there are 4 groups of entities specified:

```
GroupOfEntities harvestersGroup;       // Rubber Collectors
GroupOfEntities staticDefendersGroup;  // Defenders with a given point to wait on
GroupOfEntities activeDefendersGroup;  // Defenders searching for enemy units
GroupOfEntities attackingUnitsGroup;   // Attack units
```

*Code example from class FSM_AI_Player_NoActivity, file FSM_AI_Player_NoActivity.cs*

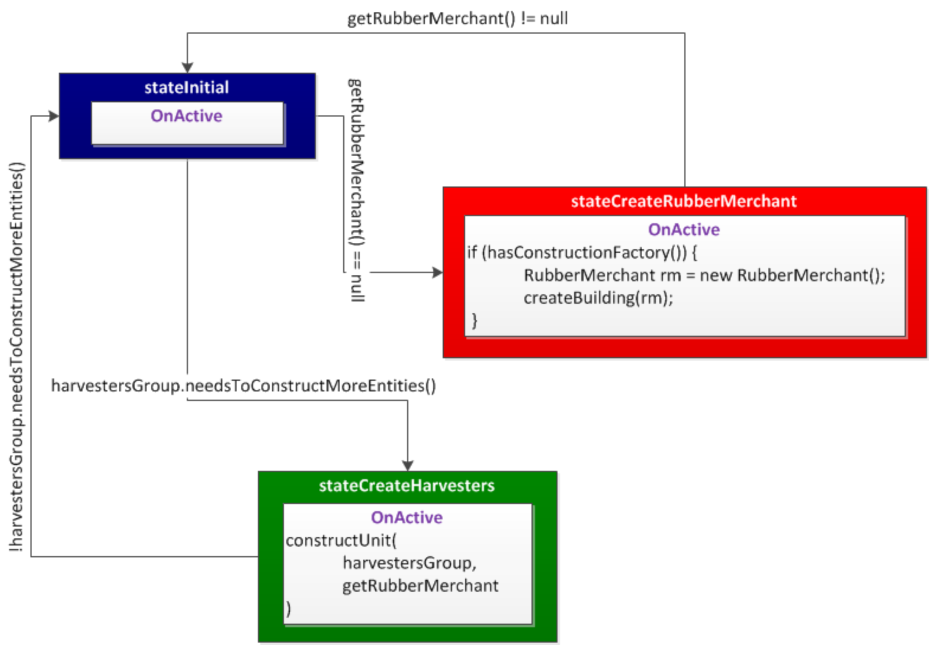

getRubberMerchant() != null
stateInitial
OnActive
getRubberMerchant() == null
stateCreateRubberMerchant
OnActive
if (hasConstructionFactory()) {
RubberMerchant rm = new RubberMerchant();
createBuilding(rm);
}
!harvestersGroup.needsToConstructMoreEntities()
harvestersGroup.needsToConstructMoreEntities()
stateCreateHarvesters
OnActive
constructUnit(
harvestersGroup,
getRubberMerchant
)

*Figure 35: Simplified FSM of a player*

In order to provide an understandable example of how the `FSM_AI_Player_No-Activity` works, we will restrain the strategy to creating a `RubberMerchant` and `RubberCollectors` (i.e. states `stateInitial`, `stateCreateRubberMerchant` and `stateCreateHarvesters` from the original FSM) – see Figure 35. Type `rmType` is the `Type` of `RubberMerchant`, `Type harvesterType` is the `Type` of `ResourceCollector` and `Type harvesterFSM` is the `Type` of `FSM_CollectorObject`. In the following example, we show the code behind the FSM from Figure 35 with highlighted states:

```
stateInitial.OnUpdate = () => {
      if (getRubberMerchant() == null) return stateCreateRubberMerchant;
      if (harvestersGroup.needsToConstructMoreEntities())
            return stateCreateHarvesters;
      return null;
};

stateCreateRubberMerchant.AddTransition(stateCreateHarvesters,
                                               () => getRubberMerchant() != null);
stateCreateRubberMerchant.OnActive = () => {
      if (hasConstructionFactory()) { createBuilding(new RubberMerchant()); }
};

stateCreateHarvesters.AddTransition(stateCreateRubberMerchant,
                                               () => getRubberMerchant() == null);
stateCreateHarvesters.AddTransition(stateInitial,
                              () => !harvestersGroup.needsToConstructMoreEntities());
stateCreateHarvesters.OnActive = () => {
      constructUnit(harvestersGroup, getRubberMerchant);
};
```

*Code example from class FSM_AI_Player_NoActivity, file FSM_AI_Player_NoActivity.cs*

**Defensive and offensive strategies**

In class `FSM_AI_Player_NoActivity`, we set the required numbers of all entities in the groups to zero, so that the AI does not create any of them. AI designers can extend this AI so that they set the desired numbers of entities in the groups according to their preferences – as we do in the project `FSM_Player_AI.csproj`, where we define `FSM_AI_Player_Defender` and `FSM_AI_Player_Attacker.`

`FSM_AI_Player_Defender` is a defensive computer player strategy, which requires more defensive entities and less offensive ones:

```
/* The constructor has 4 arguments: type of the entity to be created, minimum
acceptable number of entities in the group, maximum number of entities in the group,
and type of FSM for the entities in the group. */
harvestersGroup = new GroupOfEntities(harvesterType, 2, 4, harvesterFSM);
activeDefendersGroup = new GroupOfEntities(smallTankType, 8, 12, activeDefenderFSM);
staticDefendersGroup = new GroupOfEntities(bigTankType, 8, 16, staticDefendersFSM);
attackingUnitsGroup = new GroupOfEntities(bigTankType, 3, 7, wideSearchAttackersFSM);
```

*Code example from class FSM_AI_Player_Defender, file FSM_AI_Player.cs*

`FSM_AI_Player_Attacker` is a strategy requiring a larger number of offensive entities and less defensive ones:

```
harvestersGroup = new GroupOfEntities(harvesterType, 1, 3, harvesterFSM);
activeDefendersGroup = new GroupOfEntities(smallTankType, 4, 6, activeDefenderFSM);
staticDefendersGroup = new GroupOfEntities(bigTankType, 8, 8, staticDefendersFSM);
attackingUnitsGroup = new GroupOfEntities(bigTankType, 10, 20,
                                                            wideSearchAttackersFSM);
```

*Code example from class FSM_AI_Player_Attacker, file FSM_AI_Player.cs*

## 3.7 Map and pathfinding

All types related to map and pathfinding are grouped in namespace `RTSGame.GameMap`. All types related to the work with tiles are generic, so that the types from this namespace can be used with tiles of any shape (represented by interface `IgridTile`).

As we described in section 2.5.2, we use square-shaped tiles (`SquareTile : IgridTile`). From a square tile, a moving entity can move into 8 directions, which are specified in class `SquareTileDirections.`

The map is encapsulated by class `Grid<Tile>`, which orders the tiles into a two-dimensional array and makes them conveniently accessible through the indexer `this[Point p]`. This class is also responsible for management of the map (inserting and removing entities and obstacles, finding out if an area in the map is free or what unit is at a given point in the map, etc.). In the game, we use the class `Grid<SquareTile>`.

We will now briefly describe the most important types contained in the namespace `RTSGame.GameMap`.

**Class MovementPath<Vector>**

This class represents a relative path through the map (see section 2.5.3 for details). In the game, we use class `MovementPath<Point>`.

**Class AStar<Tile>**

This class, inheriting from abstract class `PathSearchingAlgorithm<Tile>`, which stands for general pathfinding algorithm, represents the algorithm A* (described in section 2.5.4.2). In its main method `FindPath`, it initializes the pathfinding using method `initSearching`, then, method `searchForPathUntil-TargetIsReached` is run. If the target is reached, the path is assembled in method `reconstructPath`.

The navigation graph for the pathfinding (described in 2.5.4) is represented by class `DynamicNavigationGraph<Tile>`. The vertices and edges of the graph are determined on the fly from the original map (`DynamicNavigation-Graph<Tile>.mOriginalGrid`).

For keeping the queue of visited vertices sorted by their fitness, we use a heap implemented in `GridTileHeap<Tile>`, whose nodes are represented by class `GridTileHeap<Tile>.HeapNode`, which also carries the information (`mCameFrom`) about the `HeapNode` from which the tile stored in the current `HeapNode` has been visited. This information is then used for the reconstruction of the path in method `reconstructPath`.

The `GridTileHeap<Tile>` uses a cache (`mSimpleCache`) to store the `HeapNode` with the lowest fitness value. This approach considerably decreases the number of times when the node with the lowest fitness has to be taken from the heap, thus decreasing the amount of time the algorithm consumes.

**Class GridThumbnail**

This class creates the thumbnail from the map, whose presence is required in goal 1.10. In its method `CreateThumbnail`, the class uses unsafe code to accelerate the creation of the thumbnail.

If the user clicks into the thumbnail, method `clickedIntoMap` centers the screen to the point into which the user clicked, using `ScrollManager.setPosition`.

### 3.7.1 Loading maps

In the initialization phase of the game, the program has to load maps to offer them to the player, who chooses which map to play. Maps, stored as XML documents, are loaded by class `ScenarioManager` from directory `.\scenarios`. The XML is then read by `Level.initLevelMap`, where the map for the Level is initialized.

As examples, we provide 3 sample maps in directory `RTSGame\RTSGame\RTSGame\Scenarios`. Let's describe one of them: `1-scenario-100x100.xml`:

```
<scenario>        The root element is called scenario
     <name>Basic Scenario 100x100</name>  This name is displayed the players
     <width>100</width>                   Width of the scenario in tiles
     <height>100</height>                 Height of the scenario in tiles

     <obstacles>             List of obstacles
           <obstacle>              One obstacle
                 <x>20</x>                Its x-coordinate is 20 (in tiles)
                 <y>30</y>                Its y-coordinate is 30 (in tiles)
                 <width>20</width>        Its width is 20 tiles
                 <height>4</height>       Its height is 4 tiles
           </obstacle>
     </obstacles>
     Definition for resource Rubber. Always define resources with their name from
enum TypesOfResources, but in lowercase:
     <rubber>
           <group>     GroupOfResources containing Rubber
                 <tile>     Tile in the group at [30, 30]
                       <x>30</x>
                       <y>30</y>
                       <amount>90</amount> Initial value: 90 units of Rubber
                 </tile>
           </group>
           <tile>
                 A tile without any group: a group will be created for it
                 <x>200</x> Tiles with wrong coordinates will be ignored
                 <y>500</y>
                 If there is no initial value, we use the default value
           (ResourceHolder.mInitialResourceValue).
           </tile>
     </rubber>
</scenario>
```

*XML example from file RTSGame\RTSGame\RTSGame\Scenarios\ 1-scenario-100x100.xml*

XML Schema for validating maps in the format we described above can be found in Attachments [D].

## 3.8 Deployment – RTSGameSetup.vdproj

The deployment of the application is processed in setup project `RTSGameSetup.vdproj`, which is set up to compile into directory `RTSGame\RTSGameSetup\XXX\`, where `XXX` stands for `Debug` or `Release` (depending on settings of the project `RTSGame.csproj`).

It creates two files: `RTSGame.msi` and `setup.exe`, which together compose the installation packet of the game.

The setup project defines 3 prerequisites for successful installation of the game:

(1) Microsoft .NET Framework 4 Client Profile (x86 and x64)

(2) Microsoft XNA Framework Redistributable 4.0

(3) Windows Installer 3.1

If any of the prerequisites is missing at the user's computer, the installer `setup.exe` downloads and installs it.

The current installation packet can be also found in Attachments [C].

# 4 User documentation

## 4.1 Installation

The installation packet (accessible from Attachments [C]) contains 2 files: `RTSGame.msi` and `setup.exe`. Run the file `setup.exe` to begin the installation.

The program needs the following prerequisites before the game can be installed. If any of them missing, the installer will download and installs them for you:

(1) Microsoft .NET Framework 4 Client Profile (x86 and x64)
(2) Microsoft XNA Framework Redistributable 4.0
(3) Windows Installer 3.1

## 4.2 Initial screen

After starting the program, you see the initial screen (Figure 36), where you can choose from several options to customize the game according to your preferences.

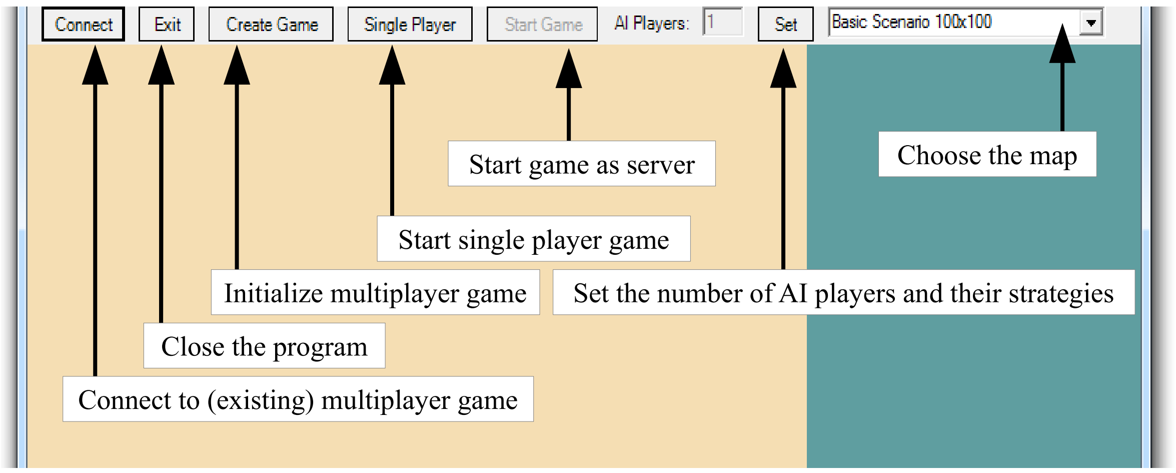


*Figure 36: The initial screen of the game*

### Number of computer players and their strategies

After clicking on the button *Set*, you can set the number and strategies of computer players in a dialogue box shown in Figure 37.

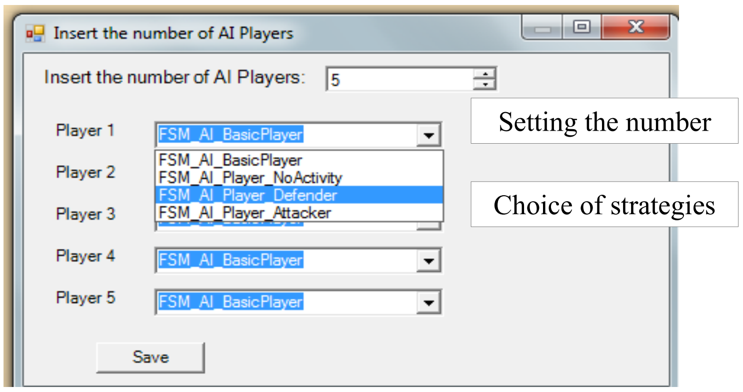

Insert the number of AI Players
Insert the number of AI Players: 5
Setting the number
Player 1 FSM_AI_BasicPlayer
FSM_AI_BasicPlayer
FSM_AI_Player_NoActivity
FSM_AI_Player_Defender
FSM_AI_Player_Attacker
Choice of strategies
Player 2
Player 3
Player 4 FSM_AI_BasicPlayer
Player 5 FSM_AI_BasicPlayer
Save

*Figure 37: The choice of number and strategies of computer players*

### Single player game

If starting a single-player game, choose the number and strategies of computer players, and the map in which the game will be played. After you click on button *Single Player*, the game will start.

**Multiplayer game**

If more players want to play a multiplayer game, they have to choose one among them who will set up the game and take the role of a server game (use of the most powerful computer is recommended). This player then chooses the number and strategies of computer players and the map before he or she initializes the server game while clicking on button *Create Game*. After this button has been clicked, a message appears: "Game created on IP: *IPAddress*:*Port*". After the message has appeared, other players can connect to the game using the button *Connect*, which makes the dialogue box from Figure 38 appear. Here, insert the *IPAddress* and *Port* from the message the server game has shown.

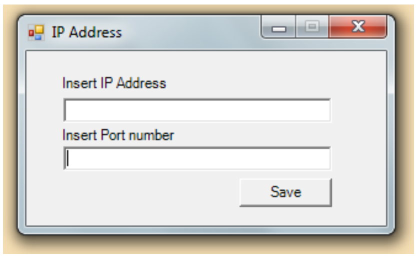


*Figure 38: Dialogue box for inserting IP Address and port*

After all clients have connected to the game, the player at the server game starts the game pressing the button *Start Game*, which is now active (unlike in the initial screen in Figure 36).

## 4.3 Gameplay

After the game has been started, the player plays against the opponents (computer or human players). The goal of the game is to destroy all entities of all opponents – only an opponent whose entities has been totally destroyed is considered dead.

As we thoroughly described in Chapter 1, the player has to collect resources in order to be able to pay for the construction of buildings and units to defend his or her base or to attack the opponents' entities. We will first describe the entities and then, the chain of resource management, because most of the entities present in the game are important parts of the resource management.

### 4.3.1 Entities

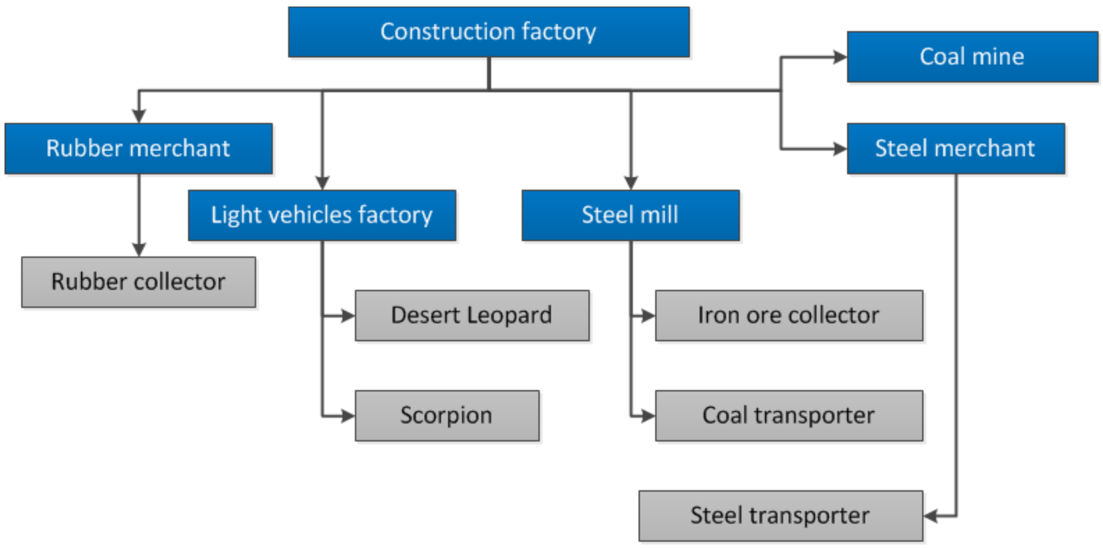
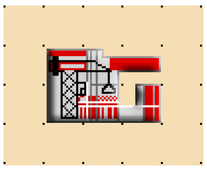


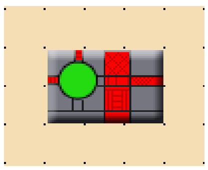
*Figure 39: The tree of buildings and units in our game*

In Figure 39, we show the structure of buildings (blue background) and units (gray background) in the game. Arrow from entity A to entity B symbolizes that entity B is constructed in entity A. We will now present a brief description of each of the entities.



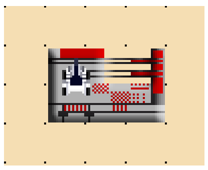
**Construction factory**

Construction factory is the main building of the game, because it constructs all other buildings available in the game, and it can not be constructed in any other building. It the player looses this building, his or her possibilities are very limited.



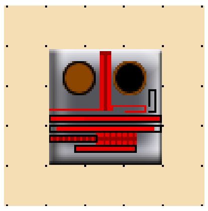
**Rubber merchant**

Rubber merchant collects resource *Rubber* from Rubber collectors and transforms it into *Money*, which is transferred to the player's resources. It also creates Rubber collectors.



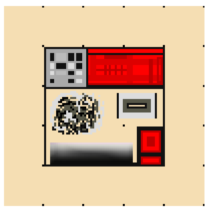
**Light vehicles factory**

Light vehicles factory is a building creating attack units – Desert leopards and Scorpions.



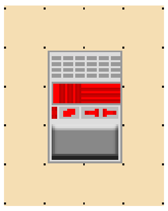
**Steel mill**

Steel mill is a building which receives resources *Coal* and *Iron* and transforms them into *Steel*. It creates units Iron ore collector and Coal transporter.



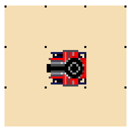
**Coal mine**

Coal mine mines resource *Coal*. It has to be placed on an area in the map where *Coal* is present.



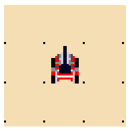
**Steel merchant**

Steel merchant receives resource *Steel* transported by Steel transporter from Steel mill and transforms it into *Money*, which is then transferred to the player's resources. It creates unit Steel transporter.



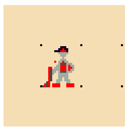
**Desert leopard**

Desert leopard, constructed in Light vehicles factory, is one of two attack units present in the game. It is slower and more expensive, but also more powerful than Scorpions.



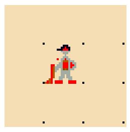
**Scorpion**

Scorpion is another tank constructed in Light vehicles factory. It is faster, cheaper, and less powerful than Desert leopard.



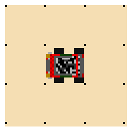
**Rubber collector**

Rubber collector collects resource *Rubber* and brings it to the Rubber merchant, where it is transformed into *Money*, which is then transferred to the player's resources.



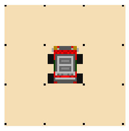
**Iron ore collector**

Iron ore collector collects resource *Iron* from areas in the map and brings it into the Steel mill.



**Coal transporter**

Coal transporter transports *Coal* from Coal mine into Steel mill.



**Steel transporter**

Steel transporter transports *Steel* from Steel mill to the Steel merchant.

### 4.3.2 Resources

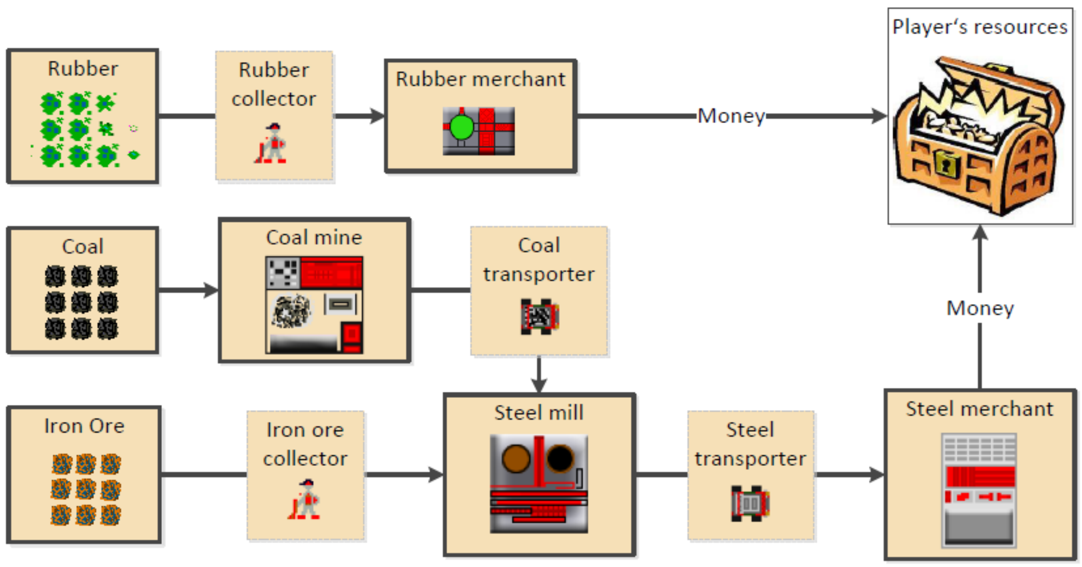


*Figure 40: Chain of resource management in our game*

We show the chain of resource management in our game in Figure 40. There are 5 types of resources in the game:

**Money**

*Money* is the resource which is used to pay for units and structures, and the players receive some initial amount of it to be able to start their economy. The player can obtain money in 2 buildings:

(1) **Rubber merchant** receives *Rubber* from Rubber collectors and transforms it into *Money*.

(2) **Steel merchant** receives *Steel* from Steel transporters and transforms it into *Money*.

**Rubber**

*Rubber* grows in bunches in the map, and it can be collected by Rubber collectors which then bring it to the Rubber merchant.

*Rubber* growing in the map is destroyed by a unit (except for Rubber collector) which drives over it.

**Coal**

*Coal* is a resource present in map, and it can be mined in building Coal mine, which has to be constructed over fields with *Coal* on them. *Coal* is then transported to Steel mill, where it is, together with *Iron*, transformed into *Steel*.

**Iron**

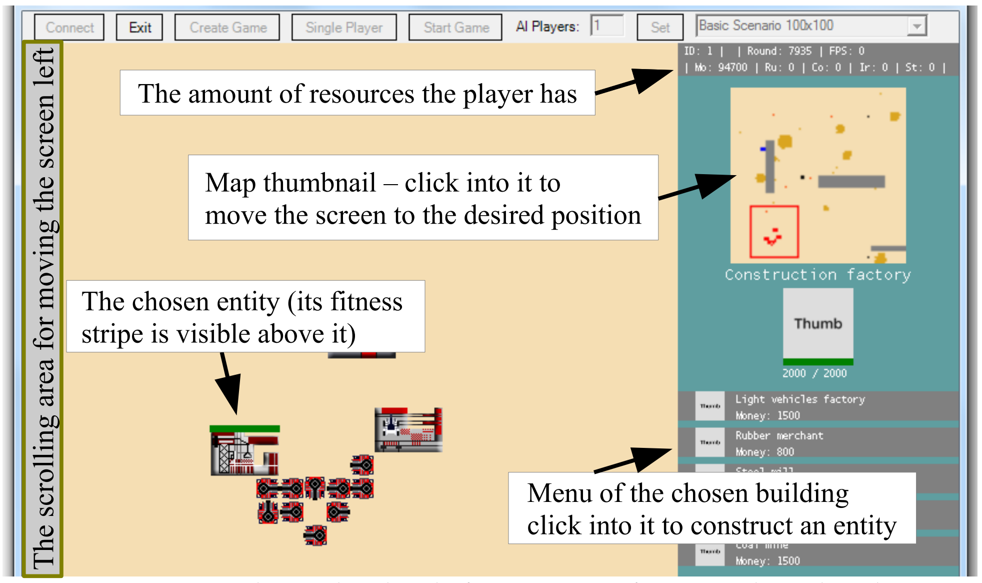
*Iron* is a resource present in map, and it can be collected by Iron ore collector. It is then brought to Steel mill, where it is, together with *Coal*, transformed into *Steel*.

**Steel**

*Steel* is created in Steel mill from *Coal* and *Iron*. It is then transported to the Steel merchant, where it is transformed into *Money*, which is transferred to the player's resources.

## 4.4 Player's input

During the gameplay, the player gives commands to his or her entities and to the game using mouse. A game window is shown in Figure 41: we see the game progress



Connect
Exit
Create Game
Single Player
Start Game
AI Players:
1
Set
Basic Scenario 100x100
ID: 1 | | Round: 7935 | FPS: 0
| Mo: 94700 | Ru: 0 | Co: 0 | Ir: 0 | St: 0 |
The amount of resources the player has
Map thumbnail – click into it to move the screen to the desired position
Construction factory
Thumb
2000 / 2000
The chosen entity (its fitness stripe is visible above it)
The scrolling area for moving the screen left
Light vehicles factory
Money: 1500
Rubber merchant
Money: 800
Money: 1500
Menu of the chosen building click into it to construct an entity

*Figure 41: Game window with a detail of Construction factory in the right column.*

in the main window, and the map thumbnail and a detail of Construction factory in the right column.

### Choosing object

Left mouse click on an object (entity, resource, obstacle) chooses it and displays its details in the right column. For choosing more units, drag over them or hold CTRL and click on the units you want to choose (this only works for units, not for buildings).

### Constructing entities

If a building is chosen, the player can construct entities from the building menu (right mouse click). In Figure 41, there is a choice of 5 different entities to be constructed.

In case of construction of a building, the player has to choose where to place the building – some buildings may have certain requirements, for example, a Coal mine has to be placed on resource *Coal*. The area is chosen with a left mouse click, the construction can be canceled with the right mouse button.

If the player constructs a unit, the unit is placed at the closest free tile to the building.

### Move-to command

If a unit is chosen, the player can instruct it to move to any point in the map with the left mouse click. Attack units (i.e. Desert leopard and Scorpion) start to attack enemy entities if the user left-clicks on the enemy entity, resource collectors start to collect resources if the player left-clicks on resources, and resource transporters transport resources from the building the user left-clicks on (if there is suitable resource type available).

**Moving the screen**

There are generally two ways of moving the screen. Either, the player can click into the map thumbnail and the map will be centered to the given point, or the player can move the mouse to the border area of the game window and the screen will start scrolling (in Figure 41, we show the scrolling area for moving the screen left – other 3 directions are in the same area, but at the respective borders of the screen).

# 5 License

The product of this thesis is an open-source RTS game intended as a contribution to the community. For this reason, it is licensed with the MIT License published on the official website of the Open Source Initiative (www.opensource.org).

# 6 Comparison with similar products

In this chapter, we compare the product of out thesis with the most popular open-source RTS games and engines. We first describe them and then present a table in Figure 42 summarizing the most important features of the products.

**Spring [47]**

The Spring, published under the GPL license, is a cross-platform (Windows and Linux) 3D RTS game engine written primarily in C++ and using the Lua language for game-specific code. It is used as the base for many other computer games, such as Kernel Panic or Zero-K. It allows customization of almost all components, such as GUI, unit AI or pathfinding.

The Spring engine supports both on-line and LAN multiplayer games and single player games with limitation for the number of entities used in the game as high as 5000. Considering the maps, Spring allows large and highly-detailed maps with terrain profile and 3 layers: land, water and air.

**0 A.D. / Pyrogenesis [48]**

0 A.D. is an open-source 2D RTS game based on the engine Pyrogenesis, which a custom engine developed for 0 A.D. only. The engine, written mostly in C++ and using JavaScript for scripting, is a cross-platform project supporting Windows, Linux and Mac OS X.

The game supports single player and multiplayer games using peer-to-peer networking without a central server.

In the game, two types of licenses are used: GPL license for the engine and CC-BY-SA license for the game content – the art.

**Stratagus [49]**

The Stratagus, formerly known as FreeCraft, is an open-source 2D RTS engine, which has been used as the base for many RTS games (e.g. Bos Wars). The engine is referred to as cross-platform, even though we have not found any list of platforms it supports. The homepage of Bos Wars says that the game runs under Linux, MS Windows, BSD, and Mac OS, from which we assume that the Stratagus engine supports these platforms, too.

The engine has been licensed under GPL license. It uses C++ as the main language, and Lua as the configuration language. It also supports both single player games with computer opponents and multiplayer games over the local network. The homepage says that a server for on-line game sessions is under heavy development.

**ORTS [50]**

The ORTS (Open Real-Time Strategy) is an engine currently under development at the University of Alberta, Edmonton, Canada. It is intended as a scientific project for studying problems connected to real-time AI, such as pathfinding, scheduling, planning, etc in the environment of 2D and 3D computer games.

The engine supports both single player and multiplayer games, and it presents a different approach to network communication. In contrast to the most popular RTS games, whose network communication is based on peer-to-peer technology with distributed computation, the ORTS engine leaves all computations to the server, only distributing its results to client games. With this approach, the clients can not hack their software so as to reveal a piece of information which is supposed to be hidden from them (it would give them an unfair advantage), which makes games based on the ORTS engine suitable for internet-based competitions.

The engine is written in C++ and it uses its proprietary scripting language. The game is cross-platform, as it can be used under Windows, Linux and Mac OS.

**Comparison**

After we described the products, we will summarize what we learned in the following table:

| | Spring | 0 A.D. / Pyrogenesis | Stratagus | ORTS | Our project: RTSGame |
|---|---|---|---|---|---|
| Programming languages | C++, Lua | C++, JavaScript | C++, Lua | C++ | **C#** |
| Online games | ✓ | - | - | - | - |
| LAN multiplayer | ✓ | ✓ | ✓ | ✓ | ✓ |
| Single player games | ✓ | ✓ | ✓ | ✓ | ✓ |
| 2D support | ✓ | ✓ | ✓ | ✓ | ✓ |
| 3D support | ✓ | - | - | ✓ | - |
| Windows supported | ✓ | ✓ | ✓ | ✓ | ✓ |
| Linux supported | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mac OS supported | - | ✓ | ✓ | ✓ | ✓ |
| License | GPL | GPL | GPL | GPL | **MIT** |

*Figure 42: The summary of comparison with similar products*

As we can see, our project comprises certain advantages over the other products.

The most important difference is that we developed the whole game for .NET Framework' modern programming language C#. This, compared to languages such as C++ and Lua, enables the developers to work in a state-of-the-art, user-friendly programming environment.

Also, we publish the project under the MIT license, which allows anybody who obtains a copy of the software to "*use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software*" without any limits. This makes the project more accessible than other products we compared, because GPL license prevents the code from being used in any type of proprietary software, whereas MIT license does not, so that the game can also be turned into such type of software.

# 7 Conclusion

To conclude the Thesis, let us examine how it fulfilled the goals we set up in Section 1.3:

We developed an illustrative and extendable bird's-eye projected RTS computer game for .NET framework and its modern programming language C#, which can be further used by the community and tailored to their needs. To enable an unrestricted use of the project, we published it under the MIT license, which is less restrictive than the GPL license used for most of open-source RTS games.

The game contains both a single player mode, and a multiplayer mode. Players in both modes can choose from 3 different strategies for computer players (artificial intelligence strategies). The system of artificial intelligence is easily extendible with plug-ins, and the game contains two computer player strategies loaded as plug-in `FMS_Player_AI.dll`.

The game comprises three main components of RTS games:

- **Base building:** The game contains 6 buildings, which allow the player to construct new entities and which are a part of the chain of resource management. In the beginning of the game, the player receives one building, which can not be created in any other building.
- **Elements of war tactics:** The player can construct 2 types of attack units, which can be used both for defending the base and attacking enemy entities. We also defined some basic interactions between entities. The system of entities is extendable with plug-ins, and attack units are loaded as a plug-in `tanks.dll`. In the game, entities have their own internal artificial intelligence, which enables them to behave as autonomous agents. We provide a sample extension of the AI for entities loaded as plug-in `FMS_AI.dll`.
- **Resource management:** The game contains 5 types of resources, from which, 3 types (one of them growing, two not-growing) can be collected from the map (using resource collectors) and 2 types have to be created from collected resources in buildings. The resources (especially one of them – Money) are used as a general currency in the game; players use them to pay for construction of entities. Resource management can be easily extended by design.

The game also contains a map with obstacles, which has one layer and no terrain profile. Maps are loaded from XML files containing the definition of obstacles and resources present in the map from the beginning. The XML can be verified by attached XML Schema file. There is a map thumbnail available to improve player's orientation in the map.

In the development documentation, we provide a detailed description of the concepts used in the game with numerous examples in order to explain the concept of extendability of the game to the community.

We provide a tool for testing correctness of information exchanged between games communicating over network, which is a command-line application `LogsComparerTool.exe`. The tool also tests logs of random numbers used in communicating games, in order to verify their correct use.

We can conclude that all the project goals set up in section 1.3 have been fulfilled.

# 8 Recommendations for future work

Although the project we created is a fully functional computer game, there is a large number of possible improvements:

- Many RTS games use a level editor for designing maps. This feature has to be able to output a map in the XML format we introduced in Section 3.7.1. The level editor would be a great extension, because it would enable users without any experience with XML to add new maps.

- The game currently only offers the possibility to play a single-level game, but no campaigns. We believe that the possibility of playing a campaign would bring the game closer to modern RTS games.

- Loading and saving the game is currently not possible. While it is usually not possible to save multiplayer sessions in RTS games, saving a single player game and reloading it is considered to be a standard feature, which is missing in our project.

- As we explained in Section 1, we have given up the terrain profile of the map for the sake of the illustrative nature of the game. However, in most commercial RTS games, the terrain profile plays a significant role. Altering the game map so that it bears terrain profile would be a very useful supplement to the game, which would increase its quality.

- The game map currently has only one type of obstacles, but this is usually not the case in RTS games. Adding a variety of different obstacles would certainly improve the player's experience from playing the game.

- In order to improve the gameplay and to enhance the player's experience, the system of units and building in the game needs an extension and improvements. Along with additional entities, it will be necessary to extend the game with new strategies (internal intelligence) for their autonomous behavior. The entities could also do with a new design.

- We provide only a basic definition of interactions between entities. In numerous RTS games, the system of interactions is very elaborated, and it has a strong influence on the strategy players have to take to win the game. Defining more complex interactions between entities will definitely enhance the entertainment effect on players.

- The set of artificial intelligences for computer players is very limited (there are only four of them). The strategies only know how to collect rubber, and they are not able to use other resources. Also, they use simple logic, so that they only repeat certain steps, and human players will easily find effective strategies against the computer players. Adding more computer player strategies, which would be more elaborated than the ones currently present in the game, will certainly improve the overall quality of experience the human player gains from playing the game.

- It is common in RTS games that players can form alliances, so that they do not attack each other and cooperate in attacking their enemies. However, the possibility of alliances is missing in our game. A useful hint may be, that the core of implementation of alliances can be done in method `Payer.isEnemeyPlayerID`, which identifies enemy and friendly units according to the ID of their player owner.

- The current implementation of the game does not contain the “fog of war”, which usually covers areas which do not have to be seen by players – i.e. unexplored or invisible areas of the map. We did not use it to enable the players to see the moves of enemies, mainly for the sake of the illustrative nature of the game. Adding the fog of war would not be complicated, because it would only require to mark tiles in the map either visible, or invisible, or unexplored, and draw the map according to it in method `Level.Draw`.

# 9 Attachments

Attachments, which can be found on the attached CD:

## A. The implementation of RTSGame

The solution `RTSGame.sln` can be found in folder `\RTSGame`. It contains source-code of the RTSGame with XML comments, and compiled binary files.

## B. Documentation created from the source-code of the project

The documentation, which has been created from the source-code using tool Sandcastle [51], can be found in folder `\Documentation`. It contains following files:

- `RTS_Game_documentation_private_members.chm`
  This file contains documentation of all members and methods, including private ones.
- `RTS_Game_documentation_public_and_protected_members.chm`
  This file contains documentation of public and protected members and methods, excluding private ones.

## C. Setup

The installation files, which can be found in folder `\setup`: `RTSGame.msi` and `setup.exe`. The begin the installation, the file `setup.exe` has to be run. The installation progress is described in section 4.1.

## D. XML Schema

XML Schema for validating maps (which are XML files) can be found under `\Schema\map_schema.xsd`.

# 10 Bibliography

[1] List of open-source video games, online document available from www: https://en.wikipedia.org/wiki/List_of_open-source_video_games

[2] .NET Framework, online document available from www: http://msdn.microsoft.com/en-us/vstudio/aa496123.aspx

[3] The State of the RTS, online document available from www: http://www.ign.com/articles/2006/04/08/the-state-of-the-rts

[4] Urban Dictionary on RTS, online document available from www: http://www.urbandictionary.com/define.php?term=rts

[5] Real-time strategy, online document available from www: http://en.wikipedia.org/wiki/Real-time_strategy

[6] SimCity, online document available from www: http://en.wikipedia.org/wiki/SimCity

[7] Pharaoh, online document available from www: http://en.wikipedia.org/wiki/Pharaoh_(video_game)

[8] Real-time tactics, online document available from www: http://en.wikipedia.org/wiki/Real-time_tactics

[9] The Total War series, online document available from www: http://en.wikipedia.org/wiki/Total_War_(series)

[10] Warhammer 40,000, online document available from www: http://en.wikipedia.org/wiki/Warhammer_40,000

[11] Turn-based strategy, online document available from www: http://en.wikipedia.org/wiki/Turn-based_strategy

[12] The Lord of the Rings: The Battle for Middle-earth II, online document available from www: http://en.wikipedia.org/wiki/The_Lord_of_the_Rings:_The_Battle_for_Middle-earth_II

[13] God Games, online document available from www: http://en.wikipedia.org/wiki/God_game

[14] Black & White, online document available from www: http://en.wikipedia.org/wiki/Black_%26_White_(video_game)

[15] Dune II, online document available from www: http://en.wikipedia.org/wiki/Dune_II

[16] Stronghold, online document available from www: http://en.wikipedia.org/wiki/Stronghold_(2001_video_game)

[17] Age of Empires , online document available from www: http://en.wikipedia.org/wiki/Age_of_Empires

[18] Original War, online document available from www: http://en.wikipedia.org/wiki/Original_War

[19] Warcraft, online document available from www: Warcraft

[20] Military tactics, online document available from www: http://en.wikipedia.org/wiki/Military_tactics

[21] Symmetric relation, online document available from www: http://en.wikipedia.org/wiki/Symmetric_relation

[22] Technology tree, online document available from www: http://en.wikipedia.org/wiki/Technology_tree

[23] 2.5D, online document available from www: http://en.wikipedia.org/wiki/2.5D

[24] Axonometric projection, online document available from www: http://en.wikipedia.org/wiki/Axonometric_projection
[25] RGBA color space, online document available from www: http://en.wikipedia.org/wiki/RGBA_color_space
[26] Bitmap Class, online document available from www: http://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.90).aspx
[27] XNA Framework Homepage, online document available from www: http://msdn.microsoft.com/en-us/centrum-xna.aspx
[28] OpenTK Homepage, online document available from www: http://www.opentk.com/
[29] SlimDX Homepage, online document available from www: http://slimdx.org/
[30] SharpDX Homepage, online document available from www: http://www.sharpdx.org/
[31] DirectX, online document available from www: http://en.wikipedia.org/wiki/DirectX
[32] MonoGame Homepage, online document available from www: http://monogame.codeplex.com/
[33] OpenGL, online document available from www: http://cs.wikipedia.org/wiki/OpenGL
[34] Buckland, Matt, Programming Game AI by Example
[35] Fuzzy Logic, online document available from www: http://en.wikipedia.org/wiki/Fuzzy_logic
[36] Scripting languages, online document available from www: http://en.wikipedia.org/wiki/Scripting_language
[37] Finite-state machine, online document available from www: http://en.wikipedia.org/wiki/Finite-state_machine
[38] The Lua programming language, online document available from www: http://www.lua.org/
[39] Python Programming Language, online document available from www: http://www.python.org/
[40] A Note on Two Problems in Connexion with Graphs, online document available from www: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf
[41] A* search algorithm, online document available from www: http://en.wikipedia.org/wiki/A-star_search_algorithm
[42] Admissible heuristic, online document available from www: http://en.wikipedia.org/wiki/Admissible_heuristic
[43] Don’t follow the shortest path!, online document available from www: http://realtimecollisiondetection.net/blog/?p=56
[44] User Datagram Protocol, online document available from www: https://en.wikipedia.org/wiki/User_Datagram_Protocol
[45] Transmission Control Protocol, online document available from www: http://cs.wikipedia.org/wiki/Transmission_Control_Protocol
[46] 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond, online document available from www: http://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php
[47] The homepage of the Spring engine, online document available from www: http://springrts.com/

[48] The homepage of 0 A.D., online document available from www: http://play0ad.com/
[49] The homepage of Stratagus, online document available from www: http://stratagus.com/
[50] The homepage of ORTS, online document available from www: https://skatgame.net/mburo/orts/
[51] The homepage of Sandcastle, online document available from www: http://sandcastle.codeplex.com/