

CZECH TECHNICAL UNIVERSITY IN PRAGUE  
FACULTY OF ELECTRICAL ENGINEERING  
DEPARTMENT OF CYBERNETICS



## BACHELOR THESIS

Practical application of AI methods in RTS  
games

Prague, 2010

Author: Viktor Chvátal



## Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze dne 25.5.2010



\_\_\_\_\_  
podpis



## Acknowledgement

I am heartily thankful to my supervisor, Mgr. Branislav Bošanský, whose encouragement, guidance and support enabled me to develop an understanding of the subject.

Lastly, I warmly thank to Mgr. Viliam Lisý, MSc. for his valuable advice and friendly help at the beginning of my work.



## Abstract

Real-time strategy(RTS) games are a growing part of video games industry. Artificial intelligence (AI) methods used in commercial games are made to entertain the user, but they are usually not able to solve other problems that are similar to problems in RTS. This work analyzes goals common to most real-time strategy games and studies AI methods for solving problems like path finding and decision making. After that, it presents the Open Real-Time Strategy (ORTS) environment which is used for development of new AI methods and is also used in competitions to compare the developed algorithms. An integration layer simplifying the access to the ORTS client code was developed and a simple AI player for the ORTS strategic combat scenario was made. The player was compared to two other AI players able to solve the scenario.





## Abstrakt

Real-time strategické hry jsou rostoucím odvětvím herního průmyslu. Umělá inteligence v komerčně připravovaných hrách je vytvořena pro zabavení hráče, ale není většinou schopna řešit obecnější úlohy vyskytující se v reálném světě. Tato práce analyzuje obecné problémy vyskytující se v real-time strategických hrách a prezentuje metody umělé inteligence použitelné na jejich řešení, jako jsou algoritmy pro hledání cesty a rozhodování. Práce seznamuje s prostředím Open Real-Time Strategy (ORTS), které je používáno pro vývoj nových metod umělé inteligence a jsou pomocí něho organizovány soutěže na porovnávání vyvinutých algoritmů. Pro zjednodušení přístupu ke klientskému kódu ORTS byla vytvořena integrační vrstva, pomocí které byl vytvořen jednoduchý hráč schopný řešit strategický scénář prostředí ORTS. Vytvořený hráč byl porovnán s dvěma dalšími hráči řešícími stejnou úlohu.



## BACHELOR PROJECT ASSIGNMENT

**Student:** Viktor Chvátal

**Study programme:** Electrical Engineering and Information Technology

**Specialisation:** Cybernetics and Measurement

**Title of Bachelor Project:** Practical Application of Methods of Artificial Intelligence  
in RTS Games

### Guidelines:

1. Analysis of the AI methods and basic algorithms used in the domain of RTS games.
2. Analysis of the ORTS environment with emphasis on possibilities of a new player Implementation.
3. Comparison of the existing AI methods and algorithms, and analysis of their practical usability in the tactical-combat scenario of RTS.
4. Implementation of a new AI-player for tactical-combat scenario in the ORTS environment using the selected method.

### Bibliography/Sources:

- [1] Buro, M.: Real-time strategy games: A New AI Research Challenge. In International Joint Conference On Artificial Intelligence, volume 18, pages 1534-1535, (2003).
- [2] Buro, M.: ORTS: A Hack-Free RTS Game Environment. Lecture Notes in Computer Science, pages 280-291, (2003).
- [3] Balla, R.-K. and Fern, A.: UCT for Tactical Assault Planning in Real-Time Strategy Games. In Proceedings of the International Joint Conference on Artificial Intelligence. Pasadena, California: Morgan Kaufmann, pages 40-45, (2009).

**Bachelor Project Supervisor:** Mgr. Branislav Bošanský

**Valid until:** the end of the winter semester of academic year 2010/2011

  
prof. Ing. Vladimír Mařík, DrSc.  
**Head of Department**



  
doc. Ing. Boris Šimák, CSc.  
**Head**

Prague, January 25, 2010



## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

**Student:** Viktor Chvátal

**Studijní program:** Elektrotechnika a informatika (bakalářský), strukturovaný

**Obor:** Kybernetika a měření

**Název tématu:** Praktické využití metod umělé inteligence pro řízení hráčů  
v real-time strategiích

### Pokyny pro vypracování:


1. Seznámení se s principy základních algoritmů pro řízení hráčů v oblasti real-time strategií (RTS)
2. Seznámení se s prostředím ORTS a možnostmi implementace nových řídicích algoritmů.
3. Porovnání těchto algoritmů, jejich vlastností a možností aplikace v rámci problematiky taktického boje v RTS.
4. Implementace zvolené metody pro řízení hráče řešícího taktický boj v prostředí ORTS.

### Seznam odborné literatury:

- [1] Buro, M.: Real-time strategy games: A New AI Research Challenge. In International Joint Conference On Artificial Intelligence, volume 18, pages 1534-1535, (2003).
- [2] Buro, M.: ORTS: A Hack-Free RTS Game Environment. Lecture Notes in Computer Science, pages 280-291, (2003).
- [3] Balla, R.-K. and Fern, A.: UCT for Tactical Assault Planning in Real-Time Strategy Games. In Proceedings of the International Joint Conference on Artificial Intelligence. Pasadena, California: Morgan Kaufmann, pages 40-45, (2009).

**Vedoucí bakalářské práce:** Mgr. Branislav Božanský

**Platnost zadání:** do konce zimního semestru 2010/2011

  
prof. Ing. Vladimír Mařík, DrSc.  
vedoucí katedry



  
doc. Ing. Boris Šimák, CSc.  
děkan

V Praze dne 25. 1. 2010



# Contents

<b>List of Figures</b>	<b>xvii</b>
<b>List of Tables</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Analysis of Goals in RTS Games</b>	<b>3</b>
2.1 Specifics of an Real-Time Strategy Game . . . . .	3
2.1.1 Resource Gathering and Management . . . . .	4
2.1.2 Decision Making Under Uncertainty . . . . .	4
2.1.3 Creating Reasoning Abstractions . . . . .	5
2.1.4 Team Collaboration . . . . .	5
2.1.5 Learning from the Opponent Behavior . . . . .	5
2.2 Differences Between Commercial and Academic AI . . . . .	6
2.2.1 Commercial Client Applications . . . . .	6
2.2.2 Academic Artificial Players . . . . .	6
<b>3 Basic Techniques Used in RTS Games</b>	<b>9</b>
3.1 Path-finding Algorithms . . . . .	9
3.1.1 Basic Path-finding Algorithms . . . . .	9
3.1.1.1 The A* Algorithm . . . . .	10
3.1.1.2 Dijkstra's Algorithm . . . . .	11

3.1.2	Cooperative Path-finding Algorithms . . . . .	12
3.1.2.1	Using a 3-Dimensional Reservation Table . . . . .	13
3.1.2.2	Potential Fields . . . . .	14
3.2	Decision-Making Algorithms . . . . .	15
3.2.1	Monte-Carlo Methods . . . . .	16
3.2.1.1	Monte-Carlo Planning . . . . .	16
3.2.1.2	UCT Tactical Planning . . . . .	17
3.2.2	Rule-Based Systems . . . . .	18
3.2.2.1	Declarative Scripting . . . . .	18
3.2.2.2	Finite-State Machine . . . . .	19
<b>4</b>	<b>Open RTS Game Environment</b>	<b>21</b>
4.1	ORTS Client-Server Architecture . . . . .	22
4.2	Client-Server Communication . . . . .	22
4.3	ORTS Competition Scenarios . . . . .	23
4.3.1	Game 1: Cooperative Path-finding . . . . .	23
4.3.2	Game 2: Strategic Combat . . . . .	24
4.3.3	Game 3: Real RTS Game . . . . .	24
4.3.4	Game 4: Tactical Combat . . . . .	25
<b>5</b>	<b>Simple AI Player Implementation</b>	<b>27</b>
5.1	Strategic Combat Scenario . . . . .	28
5.1.1	Map Tiles and Unit Positions . . . . .	28
5.1.2	Scenario Objectives . . . . .	29
5.2	AI Player Design . . . . .	32
5.2.1	Graph Discretization . . . . .	32
5.2.2	Creating Abstractions . . . . .	32
5.2.3	Opponent Modeling . . . . .	34



5.2.4	Navigating Units and Groups . . . . .	35
5.2.5	Decision-Making . . . . .	37
5.3	Practical Experiments . . . . .	38
<b>6</b>	<b>Conclusion</b>	<b>43</b>
<b>A</b>	<b>Implementation Notes</b>	<b>I</b>
A.1	Integration Layer . . . . .	I
A.2	Individual Layer . . . . .	II
A.3	Abstraction Layer . . . . .	III
A.4	Graphical Interface . . . . .	IV
<b>B</b>	<b>ORTS Installation</b>	<b>V</b>
<b>C</b>	<b>Attached CD Content</b>	<b>VII</b>



# List of Figures

3.1	A* search algorithm . . . . .	11
3.2	Dijkstra's search algorithm . . . . .	12
3.3	3-Dimensional reservation table method . . . . .	14
3.4	Example script for the rule-based system . . . . .	19
5.1	Client application structure . . . . .	28
5.2	ORTS map tiles . . . . .	29
5.3	Strategic combat scenario map size . . . . .	30
5.4	Example of strategic combat game scenario . . . . .	31
5.5	Searching graph discretization . . . . .	33
5.6	Making unit groups according to their positions . . . . .	34
5.7	Unit grouping and group path-finding . . . . .	35
5.8	Example of a path-finding method for the individuals . . . . .	36
5.9	Example of a beginning of the scenario . . . . .	40
5.10	Example of simple AI player facing the opponent . . . . .	41



# List of Tables

3.1	A simple finite-state machine example . . . . .	19
5.1	Designed player compared to player making random moves . . . . .	38
5.2	Designed player compared to Bleckinge team player . . . . .	39
A.1	Finite-state machine definition . . . . .	III



# Chapter 1

## Introduction

A strategy game is a video or board game in which the player's decision-making skill determines most of the outcome, and the luck has almost no significance. All players have similar degree of knowledge of the elements in the game.

In case of real-time strategy games, the game does not progress incrementally in turns, but all players perform actions any time they need, and the time moves continuously ahead. Most of the real-time strategy games are military simulations, where several players fight over resources placed over a two-dimensional terrain. The game-play includes setting up an economy, building units and sending them into battle where all actions are performed in real-time.

Real-time strategy games are a growing part of video games industry started by the title Dune II by Westwood studios and followed by million-sellers Starcraft by Blizzard Entertainment and Age of Empires by Ensemble studios. On the other hand, artificial intelligence techniques for real-time strategy games should be used for real military simulations in the future. Algorithms used for path-finding and planning could be used to control automatized units which can be used instead of humans operating on dangerous places. The units should work as soldiers at a war, explorers looking for survivals in buildings in fire or workers looking for mines in war zones.

The disadvantage of most commercial real-time strategy games is that their artificial intelligence(AI) is defined using scripts containing just a set of rules. Writing such scripts needs an expert knowledge of specific entities in the game world and the scripts cannot

be defined in more general way. That's why it is very difficult to reuse this type of scripts in another domain.

To change this situation, it is very important to look for new approaches and methods that can be used in real-time strategy games. To involve more AI developers, the Open Real-Time Strategy environment (ORTS) has been developed[3] and strategy game competitions are organized every year[4].

In this work, general goals of real-time strategy games are described in Chapter 2. In Chapter 3, several methods for solving path-finding and planning problems are presented. After that, Open Real-Time Strategy engine is described in Chapter 4. Chapter 5 describes an example solution of an artificial intelligence for a simple player, capable of joining its units into groups to be more powerful and able to navigate them against the enemy unit groups. Some experiments were made to compare the solution with other existing players. Chapter 6 summarizes the results of this work.



# Chapter 2

## Analysis of Goals in RTS Games

In a real-time strategy game, the players build and control units in order to defeat other players in the game. There can be two types of players acting in a real-time strategy game: the decisions can be made by humans, or by a computer algorithm. Both types of players have the same tasks they must deal with.

In this chapter, specific tasks of RTS games are discussed. After that, a difference between client applications for a human and an artificial player is described.

### 2.1 Specifics of an Real-Time Strategy Game

Basic player of a RTS game must be able to manage resources and construct the base to grow its economics. In the next stage, he must be able to produce offensive units to attack enemy bases. Even though this is satisfactory for a simple player, an advanced one should solve more types of problems.

In most real-time strategy games, the player must deal with incomplete information, because positions of opponent's units and bases are usually unknown before they are discovered. The player should also join units into groups to make them more powerful.

### 2.1.1 Resource Gathering and Management

Most of the real-time strategy games include some types of resources. The player usually starts with a small amount of resources necessary for building some basic buildings and units. At the first stage of the game, the player must be able to build more workers to increase the resource income. Gained resources are usually used for building military buildings and structures, constructing additional defenses and optionally researching new upgrades as the player economics climbs up the technology tree.

At the next stage of the game, most of the resources are used for military unit production. The player should be wise enough to leave some level of resources for repairing damaged buildings or rebuilding destroyed parts of the base, because the lost buildings may have been crucial for new units production or new technology development. Other critical need for resources can come when all workers have been destroyed by the enemy.

Some strategy games include more types of resources which are available on different places. In that case, resource management also includes setting the priorities for different resource types. Resource income should be balanced to ensure all workers are effectively used. Unit production must be controlled accordingly to the availability of different types of resources.

### 2.1.2 Decision Making Under Uncertainty

In most strategy games, the information a player gets is not complete. The player is usually given only the information about parts of the map which are in sight of his units and buildings. This functionality is also known as the “fog of war”. This fact must be kept in mind in all stages of the game. When building base defenses, their location should be chosen according to the known position of enemy base. In case there is no information yet, the enemy base location should be estimated upon player’s position on the map.

When searching for new resources, their possible locations must be estimated or found by scouts of some kinds. Known resource locations can be left for future use and protected from enemy workers. Some units should be reserved for map exploration so it would be easier to find weaknesses in the enemy bases and possible to break their own resource gathering sources.

### 2.1.3 Creating Reasoning Abstractions

Spatial reasoning is an ability to think about a big number of entities in more general way, connect them together in mind and make decisions in more abstract layer of cognition. When a hundred of soldiers move across the map, a machine usually sees many single units with different speed and position. A human would see an army approaching to the base.

Basic spatial reasoning could be described as grouping entities together and deciding about unified groups, not about any single unit inside a group. This technique would approximate big number of trees as just a forest, group of units as an army or number of free map fields as a possible path. Using this kind of approximation would decrease number of decisions made during the game. Connecting units into groups makes them also much more powerful.

### 2.1.4 Team Collaboration

Game scenario may be designed for several teams to play together with the same goals and enemies. Different player economics develop almost independently, but there should be some kind of communication between them to use the advantage of being in a team.

It is very useful to synchronize all offensive actions through some kind of messages. Also the defense should be more unified to connect the forces of all players to defend the base being under enemy attack. In case the game supports resource tributes, players should ask each other to pay a needed amount of resources. Some commercial strategy games allow to control all units of other ally player, but this feature is mostly only by human players; connecting different AIs together in this way would be more difficult problem, because it requires communicating between AI players and negotiating the strategy.

### 2.1.5 Learning from the Opponent Behavior

A player able to learn from the observations has a big advantage over a player having the same behavior all the time. This ability is more common to human and animal cognition, which is able to change its behavior after a few observations. Commercial RTS players

usually make decisions based on a static set of rules and are not able to discover enemy weaknesses and change their behavior.

## 2.2 Differences Between Commercial and Academic AI

Although a computer and a human player have to face the same problems in the game, developing an artificial intelligence for a commercial RTS game is different from researching general AI algorithms. A commercial player focuses on entertaining the user, while academic researchers try to develop general algorithms reusable for different domains.

### 2.2.1 Commercial Client Applications

The main purpose of the client application for a human player is to entertain the user, so the most important part of such software is a graphical user interface, which is usually made to show a 3-dimensional picture of the game world to look more realistic. The user interface is also trying to make unit selection and control simpler.

Because the entertainment is the main purpose of a commercial RTS client, the developers try to make the AI looking enough intelligent and competitive for a human player. The AI is often not enough strong to compete the human players, so it usually has possibility to obey the game rules by receiving additional information or resources it should not get.

### 2.2.2 Academic Artificial Players

Academic researchers of artificial intelligence methods have different objectives than commercial game developers. Their goal is to develop general methods able to make strategic and tactical decisions even though they have only incomplete information. The artificial intelligence methods are developed to be enough general to be able to solve also some types of real world problems

The environment used for an artificial intelligence research is made to have no possibility to obey the game rules. All players get only the information and resources they

should have according to the game scenario.

Player behavior in commercial RTS game is usually defined by large amount of rules, which work with specific entities of the game and usually cannot be used for another game without redefining almost all rules. The task of artificial intelligence science is to develop algorithms working in more general way, which can be reused in different domains, such as robotics or real military simulations.



# Chapter 3

## Basic Techniques Used in RTS Games

Techniques used in RTS games can be divided into two levels of abstraction. The first level are the path-finding algorithms which try to find a route for a unit or a group of units to the given destination. At the second level, the decision-making algorithms control the units or unit groups by giving them orders to attack or to move.

### 3.1 Path-finding Algorithms

There are two types of path-finding algorithms. Basic path-finding algorithms try to find a route for a individual unit in a environment considered to be static. In contrast, cooperative path-finding algorithms try to find routes for multiple units moving together and having short distances between each other. The objective of the cooperative path-finding algorithm is to minimize or to completely avoid collisions of the units. This section describes some of the existing methods to solve both types of problems.

#### 3.1.1 Basic Path-finding Algorithms

Any real-time strategy game client must be able to find a path from one location of the map to another. Map field in a RTS strategy is usually divided into small squares called map tiles. To find a path from one map tile to another, map tiles can be treated as nodes of a graph, and possibility to move from one tile to another can be described by edges

between the nodes. The graph can be searched by different methods depending on the goal of the search.

The A\* method[9] is very efficient when looking for a path between two given points, the Dijkstra's algorithm can be used to find paths to all places on the map, or to find the first occurrence of a given goal.

### 3.1.1.1 The A\* Algorithm

In real-time strategy games, A\* method can be used to find a path from a given starting point to a given destination.

In this case, each node  $n$  is characterized by an evaluation function  $f(n)$ , which estimates the total distance from starting point to the destination when going through the node  $n$ . The real distance is known only for already passed part of the path. The rest of the path from node  $n$  to the destination is only estimated, so the real distance of the whole path is not known before the path is found to the destination.

For the A\* algorithm, the estimation function consists of the cost already used  $g(n)$  to get to the node  $n$ ; and the heuristic function  $h(n)$  estimating the distance to get from the node  $n$  to the destination  $g$

$$f(n) = g(n) + h(n) \quad (3.1)$$

In case of real-time strategy games, the heuristics is usually equal to the Euclidean distance to the destination

$$h(n) = \sqrt{(n_x - g_x)^2 + (n_y - g_y)^2}, \quad (3.2)$$

where the actual point position is equal to  $n = [n_x, n_y]$ , the position of the destination is  $g = [g_x, g_y]$ .

The A\* algorithm optimizes the length of the path if and only if the function  $h(n)$  is an admissible heuristic, that means that it never overestimates the real distance to the destination. As long as the graph has cycles, the algorithm must avoid searching the same



---

```

function A*(initial_state) returns a solution, or failure
  closed ← empty set
  open ← INSERT(initial_state)
  loop do
    if EMPTY(open) then return failure.
    node ← REMOVE_BEST(open)
    if GOAL(node) then return SOLUTION(node)
    if node is not in closed
      closed ← INSERT(node)
      open ← INSERT_ALL(EXPAND(node))

```

---

Figure 3.1: A\* search algorithm using the *open* and *closed* list

node multiple times. Nodes going to be expanded are added to a list usually called *open*. Once they are expanded, they are added to a list called *closed*. A node can be added to the *open* list only if it was not added to a *closed* list before, so every node is expanded only up to once. The A\* algorithm with *open* and *closed* list is shown in Fig.3.1, where the REMOVE\_BEST function selects the node  $n$  with the smallest  $f(n)$  and the INSERT\_ALL takes all the results of the EXPAND function and inserts them into the *open* set of nodes. If the destination state is found, it's immediately returned as a success. In case the *open* list is empty, it means that all possibilities were searched and no solution was found, then the failure state is returned.

In RTS games, the search algorithm is usually called multiple times in a very short time window, so the efficiency of the algorithm is very important. The selection of the best node takes  $O(n)$  cycles when searching the whole *open* list containing the  $n$  nodes. This might not be enough sufficient when searching for paths between the nodes of a huge graph. However, when using a binary sorted tree to store the nodes, the time needed for the best node selection should drop to  $O(\log n)$ .

### 3.1.1.2 Dijkstra's Algorithm

Sometimes there is no need to search for a path between two endpoints, but it is needed to find a nearest occurrence of a goal, which can be a nearest enemy, or a nearest resource. Dijkstra's algorithm is an effective way how to achieve it, as long as there exist a path

---

```

function DIJKSTRA_SEARCH(initial_node, graph) updates all reachable nodes
initial_node.cost ← 0
for each node in graph
    node.cost ← ∞
loop do
    if EMPTY(graph) then return done.
    node ← REMOVE_BEST(graph)
    for each successor in node.successors
        successor.cost ← UPDATE_COST(node.cost + COST_BETWEEN(node, successor))

```

---

Figure 3.2: Dijkstra's search algorithm

from initial node to all nodes of the graph. The algorithm is described in Fig. 3.2.

Every node is rated by a value called cost. In RTS games, the cost has usually meaning of a distance from the starting position to the node. At the beginning of the search, the cost is set to  $\infty$  for all nodes except the initial one, which has cost of 0. If the new cost using the alternate way is lower than the previous or the initial one, it is updated by the UPDATE\_COST function. The function COST\_BETWEEN( $n_1, n_2$ ) computes the cost to travel from node  $n_1$  to  $n_2$ .

As for the previous algorithm, searching for the best node naively takes  $O(n)$  cycles. When storing all the nodes in a heap structure, the complexity of removing the best node drops to  $O(1)$  and the node cost update takes  $O(\log n)$  node switches, where  $n$  is the total number of nodes.

### 3.1.2 Cooperative Path-finding Algorithms

In a real-time strategy game, units are usually formed into groups where all units have to reach the same destination. In this case, every single unit in the group must find a path to its destination, but has to avoid any collisions with other units, otherwise both units stay blocked.

A 3-Dimensional Reservation table[10] is able to find paths for units of a group so they avoid collisions between each other, but all other obstacles are treated as stationary

objects, so there is a risk that paths will have to be re-planned. On the other hand, potential fields[8] are very powerful method to navigate units in a dynamic environment, but there are some problems to be solved before the method becomes usable.

### 3.1.2.1 Using a 3-Dimensional Reservation Table

Using an A\* method finds the shortest path of a unit to the destination, but ignores the movement of other units in a group. A 3-Dimensional Reservation Table method take in count all units in the group and their positions in different time windows. Collisions with other units and moving obstacles do not occur so often, so they are treated as stationary objects. In case of such collision the paths of all units in a group are forced to be re-planned.

In case of two-dimensional map where every point  $p(x, y)$  has up to four neighbours, only four expand functions can be applied:

$$\begin{aligned}
 \text{ExpandNorth} &: p(x, y) \rightarrow p(x, y - 1) \\
 \text{ExpandEast} &: p(x, y) \rightarrow p(x + 1, y) \\
 \text{ExpandSouth} &: p(x, y) \rightarrow p(x, y + 1) \\
 \text{ExpandWest} &: p(x, y) \rightarrow p(x - 1, y)
 \end{aligned} \tag{3.3}$$

This method is not able to calculate with different map tile occupation in different time. A 3-Dimensional reservation table method extends the map with a third dimension which stores the information about map tile occupation through the time. The point in time is determined by three coordinates  $p(x, y, t)$ . In this case, there are five different expand functions possible

$$\begin{aligned}
 \text{ExpandNorth} &: p(x, y, t) \rightarrow p(x, y - 1, t + 1) \\
 \text{ExpandEast} &: p(x, y, t) \rightarrow p(x + 1, y, t + 1) \\
 \text{ExpandSouth} &: p(x, y, t) \rightarrow p(x, y + 1, t + 1) \\
 \text{ExpandWest} &: p(x, y, t) \rightarrow p(x - 1, y, t + 1) \\
 \text{Wait} &: p(x, y) \rightarrow p(x, y, t + 1)
 \end{aligned} \tag{3.4}$$

Routes for units of a group are searched one by one and reached positions  $p(x, y, t)$  are marked as occupied. During the search, only non-occupied nodes are expanded to ensure there is not going to be any collision. A simple 3-Dimensional reservation table for two moving units is shown in Fig. 3.3.

This method is able to avoid unit collisions, but requires previous map discretization, so units can move only in orthogonal directions, and the path found is not the shortest one possible. It could be possibly extended by adding more expand functions to move the units also in diagonal directions, but the unit moving diagonally would reserve four map tiles instead of two, so it should be usable only for an environment with a lot of free space usable for the unit movement.

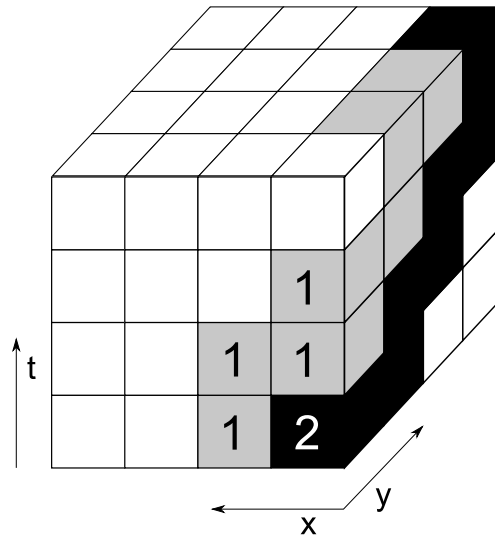


Figure 3.3: 3-Dimensional reservation table for two units

### 3.1.2.2 Potential Fields

Using potential fields is another method capable of navigating an object to a goal point on the map. The difference with previous methods like A\* and Dijkstra's algorithm is, that the path is not found explicitly, but the movement direction is computed during the time line. An advantage of this method is that it can be used for both static and dynamic environments with many moving objects.

This method assumes that every object in the game significant for planning a path

generates a potential field around itself. The field can be attractive for objects that have to be reached by path-finding, or repulsive for objects that need to be avoided. The attractive field can be modelled by positive potential values around the object, the repulsive field can be modelled by negative values. A total force acting on an object is given by a superposition of the fields of all objects in the game.

The basic principle of object navigation is to move the object to the direction with the highest increase of potential. However, there are more problems to be solved. One of the difficulties is the possibility of ending up in a local minimum of the potential. Other problem could be the narrow passages between the cliffs having too big potential, so the units tend to go another way instead of the passages.

An example of designing potential fields for the strategic combat scenario in ORTS (Section 4.3.2) can be found in [8]. The impassable cliffs, moving obstacles and own units generate a repulsive field to avoid collisions when moving. The opponent's objects generate an attractive field with its maximum in a ring with radius equal to maximum shooting distance, so the own units tend to stay at the distance of the shooting range from the opponent's units. Narrow passages are cleared by setting their potential to zero. Tiles to be cleared are identified by having negative potential and having adjacent tiles even lower potential. To avoid an unit getting stuck in a local extrema, it is moved to a random direction after it stays idle for some time.

## 3.2 Decision-Making Algorithms

Decision making algorithms are used to make decisions which group of units to move to what position and when, or which enemy unit to attack. In case the game requires building an economy, decision-making is used to control unit and structure construction.

Monte-Carlo methods are based on playing a number of games using random moves, after which the turn with the best results is selected. In contrast, the rule-based system test validity of specific conditions and take actions if the conditions were met.

### 3.2.1 Monte-Carlo Methods

Monte-Carlo methods are based on playing random turns and evaluating the results, and can be used also for planning in real-time strategy games[5]. Monte Carlo method just generates a list of possible actions for the player, and then it simulates number of games using random turns for the opponent and player based on the selected turn from the list. UCT planning[2] generates a tree of possible actions for both player and the opponent, and simulates number of games using actions given by different branches of the tree selected on the basis of a weight of the tree nodes. The weight of the nodes is computed on the basis of success of actions given by the leaves reachable from the node.

Big advantage of Monte-Carlo methods is that it is possible to stop the simulation any time and its results are still acceptable. This is an advantage for the RTS games because the time available for computation is usually very short.

#### 3.2.1.1 Monte-Carlo Planning

The Monte-Carlo algorithm contains a generator of possible turns, and an evaluation function able to evaluate the success of a game played by applying a list of turns.

Every time a decision must be made in the game, the generator makes the list of turns. After that, hundreds or thousands of games are played starting with a selected turn from a list followed by random other turns. The turn with the best results based on the evaluation function is chosen to be applied.

This approach is very useful for systems with significant uncertainty in inputs and several degrees of freedom. For such system is it often impossible or very difficult to find an exact result with some deterministic algorithm. The bigger number of inputs and input data range is, the more simulations have to be performed.

Typical Monte-Carlo simulation proceed as follows:

1. Generate a list of possible actions
2. Select the random turn from the list
3. Simulate a game by applying random moves after the previously selected turn
4. Evaluate the result by the evaluation function

5. Repeat the stages 2–4 above as much as possible upon the given computing resources
6. Select the action from the list of actions having the best results

Creating an evaluation function needs some kind of expert knowledge and usually is set by a human. It may take in count the number of units survived weighted by their number of hit points, attack potential or any other usability during the game. The advantage of the Monte-Carlo planning is that it is needed only to find a heuristic function to evaluate the plan results.

There are several parameters that can be modified in the Monte-Carlo method. At first, it is the maximum number of moves to look ahead when performing the simulation. At second, it is the total number of plans simulated, however this value is usually limited by the given computation resources.

### 3.2.1.2 UCT Tactical Planning

Upper bound for confidence tree (UCT) is an approach inspired by the Monte-Carlo method which can be likely used in real-time strategy games. In contrast to Monte-Carlo method, UCT does not play only random turns, but generates a tree of several first possible turns, where the better evaluated ones are tried more times. The simulation starts every time at the root of the tree and approaches to the leaves. At each node, probability of selecting the branch to continue depends its previous success. When a leaf is reached, the algorithm plays the game beginning with the actions reached and continuing with random actions. After that, an estimation function is computed and statistics of all leaves passed are then updated, so next time the algorithm tends to end in the leaves with better estimation value.

Each node of the tree holds the number of times  $n(s)$  it was previously visited, the number of times each action  $a$  available in node  $s$  was visited in previous runs  $n(s, a)$  and an estimation value for each action of the node  $Q(s, a)$ . If the node contains any actions which have not been explored yet, some of them is selected and explored. Otherwise, the algorithm selects an action which maximizes the upper confidence bound given by

$$Q^+(s, a) = Q(s, a) + c \times \sqrt{\frac{\log n(s)}{n(s, a)}}, \quad (3.5)$$

where  $c$  is a domain dependent constant. In fact, this constant has a significant impact on performance and its optimal value vary across different game scenarios. In order to balance exploration of the tree, and exploitation given by number of random searches, the constant  $c$  is set to  $c = Q(s, a)$ . This value of  $c$  was evaluated using practical experiments.

When the exploration path reaches the tree leaf, the reward value describing the plan result  $R$  is computed. Every previously passed leaf is then updated as follows

$$\begin{aligned} n(s, a) &\leftarrow n(s, a) + 1 \\ n(s) &\leftarrow n(s) + 1 \\ Q(s, a) &\leftarrow Q(s, a) + \frac{1}{n(s, a)} [R - Q(s, a)] \end{aligned} \tag{3.6}$$

### 3.2.2 Rule-Based Systems

To illustrate the rule-based systems, two methods were chosen. Scripting methods are used by majority of commercial RTS games[5], because they are easy to be implemented, but the disadvantage may be their unusability in different domains. On the other hand, a finite-state machine (FSM) is an easy way to describe a simple behavior of an object, using set of states and conditions to step from one state to another.

#### 3.2.2.1 Declarative Scripting

Declarative scripting method[6] works with a set of rules where each rule contains a set of conditions and a set of actions. If all the conditions are met, the actions are executed. The advantage of declarative approach is the separation of the implementation from the logic of the behavior. The behavior can be easily changed without changing the implementation, and can be tested separately from the application.

An example of the rule declaration is shown in Fig. 3.4. The first three lines after the `defrule` command describe the conditions that must be all met to perform the actions declared in the second half of the example. By using a big amount of such rules, artificial intelligence can be able to solve many situations that occur through the game-play.



---

```

(defrule
  (wood-amount > 1200)
  (or (food-amount < 1600) (or (gold-amount < 1200) (stone-amount < 650)))
  (can-sell-commodity wood)
=>
  (chat-local-to-self "excess wood")
  (release-escrow wood)
  (sell-commodity wood)
)

```

---

Figure 3.4: Example of the rule definition for selling excess resources (Age of Empires II, Ensemble Studios)

### 3.2.2.2 Finite-State Machine

A finite-state machine is a behavior model described by a finite number of states and transitions between the states. The state is changed according to the current machine state and an input condition. A simple finite-state machine transition table is shown on the Fig. 3.1. The behavior is described by actions performed when entering or leaving the states.

The finite-state machine can be used for different levels of abstractions. It can describe behavior of one single unit or a whole group of units moving together and having same targets. It can be also used for controlling the growth of whole economics, including building different types of building and researching new technologies.

Condition \ State	Move	Attack	Stay
Enemy unit in range	Attack	Attack	Attack
No more enemies on the map	Stay	Stay	Stay
Enemies out of range	Move	Move	Move

Table 3.1: A simple finite-state machine



# Chapter 4

## Open RTS Game Environment

The Open Real-Time Strategy environment[3] was built as an alternative to commercial RTS game engines that are usually closed for any changes and improvements, which makes them unusable for artificial intelligence research. In contrast, the ORTS is licenced as open-source, so any changes and improvements of the engine are possible. All computations of visibility and unit interactions are performed on a server side and the clients get only the information they are supposed to, so they have no possibility to obey the game rules.

Commercial RTS usually use a closed communication protocol to protect the intellectual property of the developers. This fact blocks any possibility to incorporate new AI algorithms by scientific researchers. The first goal of the ORTS engine is to design a free and open communication protocol allowing connection of any client software through a TCP socket. This architecture allows using any programming language and environment capable of socket communication. Because the ORTS has been published under the GNU Public License, users can change client and server software according to their needs.

These features make ORTS an ideal test bed for new AI techniques development. An open-source architecture is ideal for using functionality developed in different programming languages, such as C++ or Java, and the ORTS source code can be compiled on both Windows and Linux operating system environments. These capabilities make ORTS very popular along the AI researchers and RTS game competitions take place at University of Alberta every year from 2006, more information can be found in [4].

## 4.1 ORTS Client-Server Architecture

Most commercial RTS games use one computer marked as a server only to negotiate the game scenario settings. After that, all clients communicate between each other and transfer only orders for different units. Simulation is performed on each client machine. This kind of architecture has advantage of having very low transfer rate between client stations, client software hides map information according to player's field of view. However, this can be exploited by hackers who may reveal the information which was meant to be hidden. An example described in[3] can be a commercial poker server which sends all hidden cards information to all clients. Hacking this kind of game would have caused losing part of business profit.

The ORTS programming environment uses a different architecture based on a server side game simulation. A game scenario is created or loaded by the ORTS server and it waits for all client connections. When all clients are connected, they get information about game map and unit positions, and they are able to send orders for the units to move or to attack. Results of the orders and a visibility of the map is computed on the server side, so the players have no possibility to cheat and they receive only their own visible information. This architecture takes all hacking possibilities out of question, but requires a higher transfer rate between the client and the server, because it has to send complete game state after simulating every frame.

A time-line of the game is divided into short time frames. Every time frame consist of sending game state views to all players, waiting for all object orders and executing the received orders including the collision detection. Every game object is defined by number of parameters. Basic object properties are its position, destination and movement speed. During every time frame all new positions of moving objects are computed. In case any two object are going to collide during the time frame, they stop moving before their shapes overlap.

## 4.2 Client-Server Communication

To minimize the data transfer rate between the server and the clients, two different methods are used. Each object in ORTS is determined by following array of values:

(Object ID, Owner, Radius, Sight Range, Minimum Attack Range, Maximum Attack Range, Speed, Attack Value, Replicating, Hit Point, Moving, Position)

Most of these values remain constant for a long period of time. Because the time frames used to compute actions and visibility are very short, non-constant values change very slightly. Computing only difference vectors heavily reduces the entropy of the transferred data, which can be therefore easily compressed by LZ77 [11] algorithm. Compressing and decompressing the data increases the client and server CPU load a bit, but has much bigger advantage in low data bandwidth.

## 4.3 ORTS Competition Scenarios

There were several game types at the first AIIDE RTS Game Competition [4] in 2006. The list below refers to the updated game scenarios at last AIIDE competition in 2009 [1].

### 4.3.1 Game 1: Cooperative Path-finding

Goal of the cooperative path-finding scenario game is to gather as many resources as possible within a given amount of time. It's only a single player game, so no competitors are taken in count. However, there are moving indestructible obstacles called as "sheep" in the game map which makes the game more dynamic.

The game takes place in smaller game field, having 32 tiles at each x and y size. The player starts with a control center and 20 workers nearby. The workers have to find ways to the nearest resource patches and take resources back as soon as possible. There is a limitation of maximum four workers gathering from one single resource patch. The game map contains randomly generated static obstacles the workers must get around. Player gets full information about position of all resource patches and obstacles, so there is no need for map exploring.

The task is to effectively control the motion of all workers to avoid collisions between each other and randomly moving sheep through the map. The algorithm cannot spend

too much time on finding optimal routes, because the game goes still on and the situation could change before the computation is done.

### 4.3.2 Game 2: Strategic Combat

The strategic combat is a tank battle of two players having the full information about the map and opponent's units. The objective is to destroy as many opponent's bases as possible within the given amount of time. The combat takes place in a bigger map having 64 tiles in size. Each player starts with 5 randomly, but symmetrically located control centers surrounded by 10 tanks each. Terrain contains several static and several moving obstacles to force players to implement dynamic path-finding. In the case all bases of one player are destroyed, the game ends immediately after last building destruction.

This scenario focuses on small scale combat, where tanks should fire at weak units first or concentrate power to attack only one target to eliminate it faster. Units should be grouped into larger formations to have bigger attack power. There must be some algorithm for target selection and cooperative path-finding to ensure all units are able to get to the target.

### 4.3.3 Game 3: Real RTS Game

Real RTS game scenario combines previous two game types and adds a need to build two more types of military buildings to enable unit production. Player implementation must face the incomplete information caused by "fog of war" hiding everything which is out of the unit fields of view.

Each player is starting with a control center surrounded by 6 workers and resource cluster nearby. Position of other four resource clusters and the enemy base is unknown, so there is a need for map exploration. There is a simple technology tree consisting of three building types depending on each other and two types of units with different abilities trained in two different building types.

This type of game needs to solve several problems natural to RTS games. At the first stage, it is a worker production and resource gathering needed for building the base.

When there are the first military units created, some of them should be ordered to explore the map to find new resource clusters. They can also possibly locate the position of the enemy base. At the second stage of the game, different offensive units must be created and grouped into armies attacking the enemy base. This phase combines two approaches, the main strategy describing when to send armies and how to possibly split them to attack the base. When the combat begins, there must be some tactical mechanism which controls unit attacks and movement in smaller scale.

#### **4.3.4 Game 4: Tactical Combat**

Tactical combat extends the strategic combat described in Game 2 by adding new type of attack force. Two unit types, tanks and marines, have different moving speeds, attack forces and number of hit points. Goal of this scenario is to handle heterogeneous groups of units and destroy as many enemy forces as possible.

Solving all resource gathering, strategic combat and tactical combat problems are necessary prerequisites to build artificial intelligence able to solve all problems given in RTS game 3.





# Chapter 5

## Simple AI Player Implementation

This chapter describes the design of a simple player for strategic combat scenario working as an ORTS client. The player uses the functionality and source code fragments only taken from ORTS sample AI project which implements only simple unit movement and attack control. To make the client work with objects of different layers of abstraction, it is build of several layers:

- an integration layer separating the AI development from ORTS libraries
- an individual layer used for path-finding and individual objects manipulation
- an abstraction layer used to combine units into groups and modeling the opponent units into same kind of structures
- a sophisticated graphical output capable of visualizing unit group abstractions and routes found by path-finding algorithms. The graphical output could be easily extended to produce not only bitmaps rendered on the screen, but also a vector graphics in Adobe PostScript format.

The structure of the application is shown on the Fig. 5.1. The integration layer completely separates higher layers from the ORTS environment. All higher layers could run without ORTS using a different integration layer. The individual layer uses objects and definitions from the integration layer, the abstraction layer joins the objects from individual layer into groups. The graphical interface visualizes objects from all integration, individual and abstraction layers.

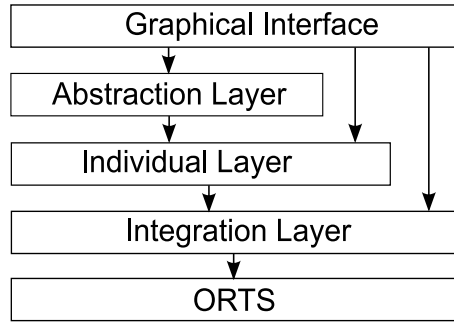


Figure 5.1: Client application structure

## 5.1 Strategic Combat Scenario

This section describes the world where the strategic combat takes place. The world includes a map divided into tiles and containing terrain structures. Players operate with moving units and try to protect their buildings situated on various places on the map.

### 5.1.1 Map Tiles and Unit Positions

The strategic combat scenario (4.3.2) map is divided into  $64 \times 64$  map tiles (Fig. 5.3). Each tile has four sides (Fig. 5.2 a), where each side may have different height. In the strategic combat scenario, the tile heights have no meaning because all units are moving in the same height.

Map tiles can be also surrounded or crossed by terrain boundaries, which cannot be crossed by any ground unit. The six possible positions of boundaries around a map tile are shown in (Fig. 5.2 b).

In ORTS, each map tile is divided into  $16 \times 16$  elementary positions (Fig. 5.2 c). Every game object can be situated on one of these  $16 \times 16$  positions of a tile, and can be ordered to move onto another elementary position of any tile, so total number of possible game object positions on the whole map is given by

$$n = n_t \cdot n_d = 64^2 \cdot 16^2 = 2^{20} \sim 10^6, \quad (5.1)$$

where  $n_t$  is total number of tiles,  $n_d$  is number of possible positions inside a map tile.

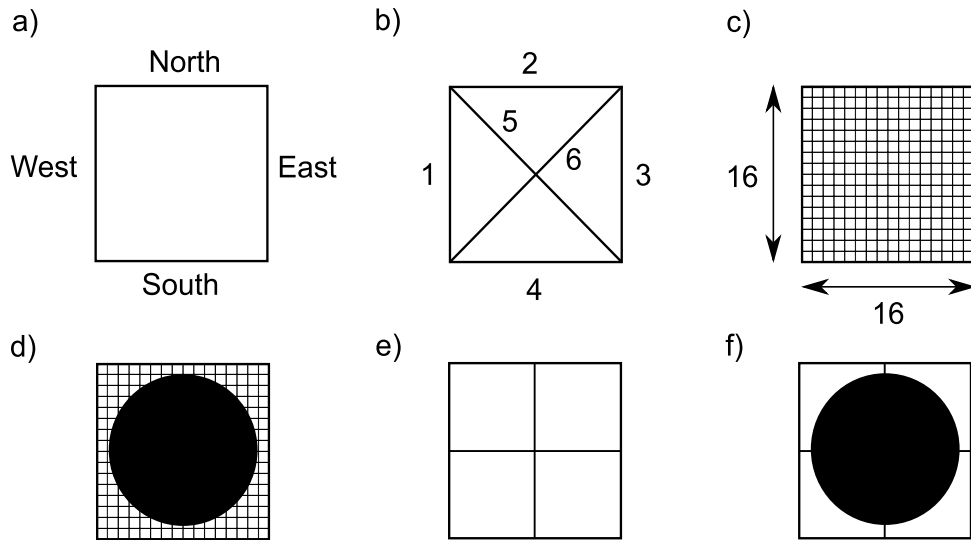


Figure 5.2: (a) Map tile sides (b) Possible terrain boundaries (c) Map tile division (d) Map tile occupied by an unit, which can be situated on one out of  $16 \times 16$  different positions (e) State space discretization with one node per tile and four edges to neighbour tiles (f) Discretized map tile occupied by an unit and possible directions for movement

### 5.1.2 Scenario Objectives

A typical strategic combat scenario is shown on Fig. 5.4. A map having  $64 \times 64$  tiles contains randomly generated terrain boundaries, which are impassable for any ground units. There are only two types of player's objects in the scenario: each player starts with five symmetrically positioned control centers incapable of any attack, surrounded by ten siege tanks each. The tanks are able to move to any non-occupied place on the map, or they can be ordered to shoot an enemy object situated inside their weapon's range radius.

A player who manages to destroy all opponent's control centers, wins the game. To make the game more dynamic, moving obstacles called 'the sheep' are added by the server. The sheep are moving in random directions and cannot be destroyed in any way. If there is no winner after 15 minutes of play, the game is a tie.

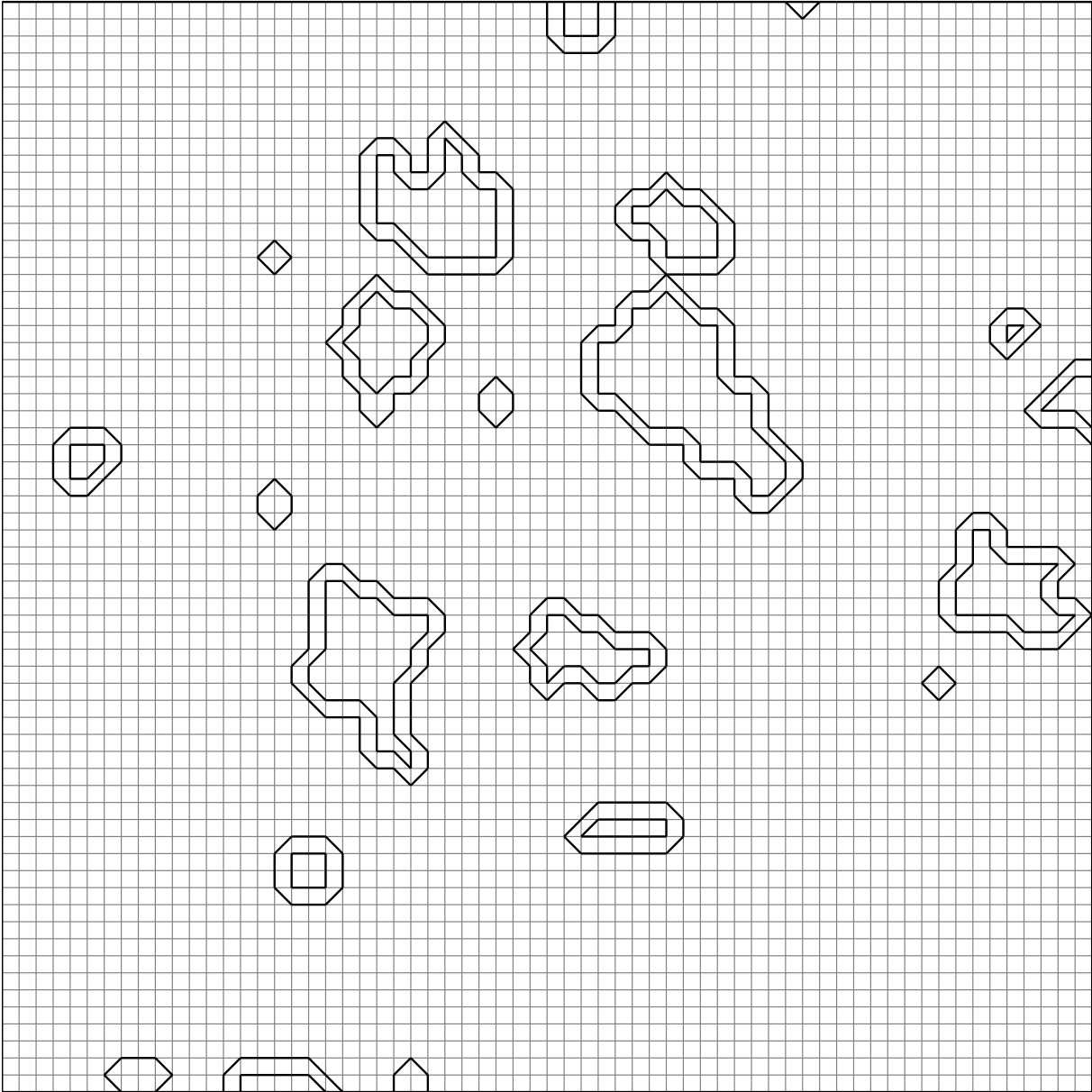


Figure 5.3: A Strategic combat scenario map divided into  $64 \times 64$  map tiles, tile boundaries are shown.

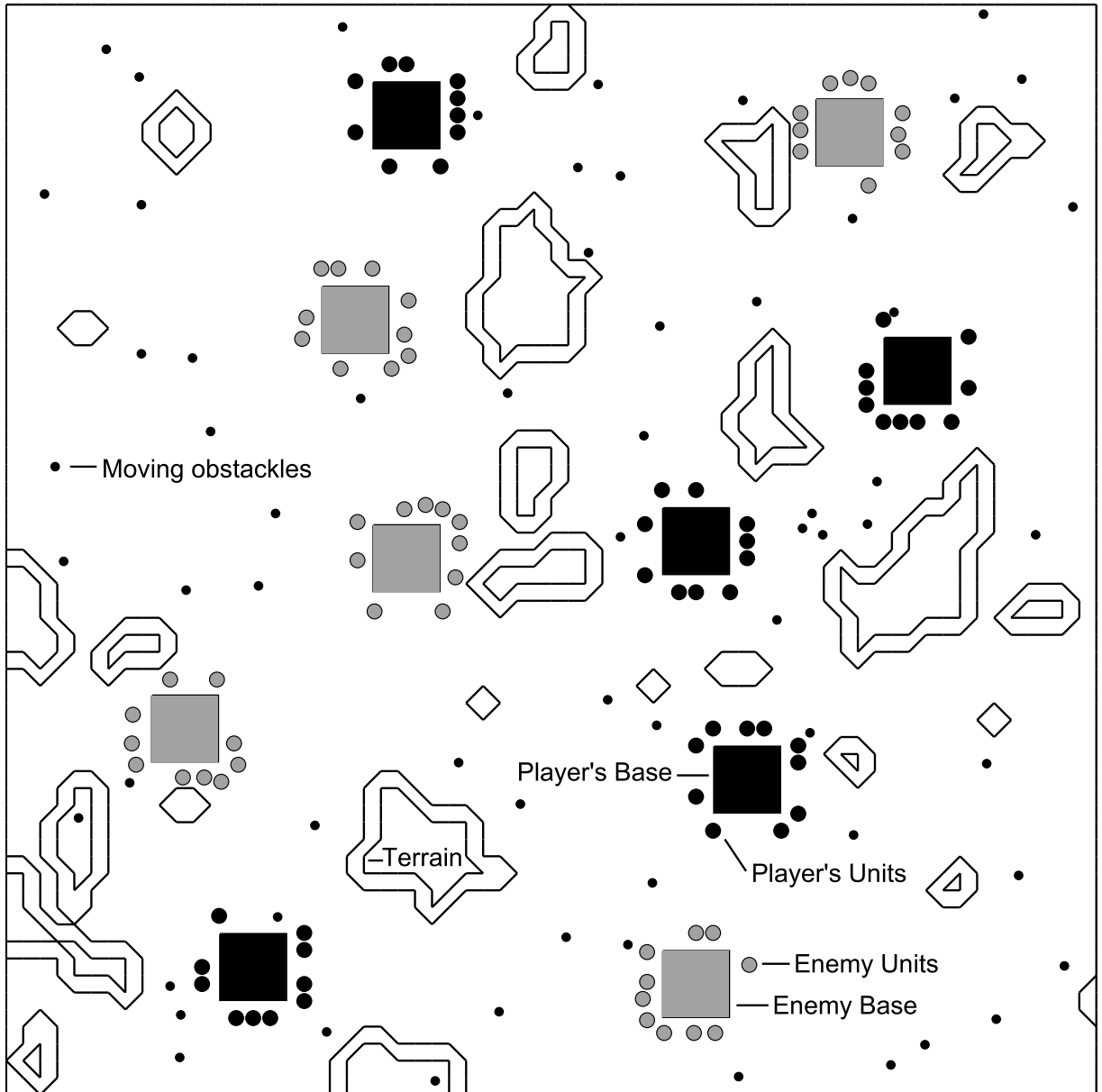


Figure 5.4: Example of strategic combat game scenario containing control centers (squares), siege tanks (bigger circles) and moving obstacles (small circles).

## 5.2 AI Player Design

The algorithm must deal with various problems that appear in different implementation layers. This section contains problem descriptions and solutions used for simple player development.

### 5.2.1 Graph Discretization

Path-finding is a functionality that must be solved by any RTS game client. Units must be able to find a free path in a very short time, because the game time is still moving ahead and never stops. The first problem is the total number of searched graph nodes, which is approximately  $10^6$  (equation 5.1). The second problem is, that one unit blocks big number of graph nodes (Fig. 5.2 d), so the path found may not be wide enough for a unit to pass through.

One of the solution is to lower the resolution of the graph, and allow unit movement only in the orthogonal directions (Fig. 5.2 e, f). In this case, one map tile correspond to exactly one graph node. An unit blocks only one node when holding a position, or two nodes when moving from one node to another. Every path is wide enough to enable unit movement without blocking other unit. In this case, maximum graph for searching contains only

$$n_{dis} = 64^2 = 2^{12} \sim 4 \cdot 10^3. \quad (5.2)$$

The routes found using this solution are not the shortest ones possible, but can easily be searched in real-time. A discretized graph with all routes possible is shown in the Fig. 5.5. Any point of the map occupied by a terrain boundary, building, unit or a moving obstacle is marked as impassable and is not able to be reached by path-finding.

### 5.2.2 Creating Abstractions

The units acting in the RTS games are much more powerful when moving in compact groups instead of acting as individuals. It is very useful to create such groups according to the unit positions, so the near units make a group.

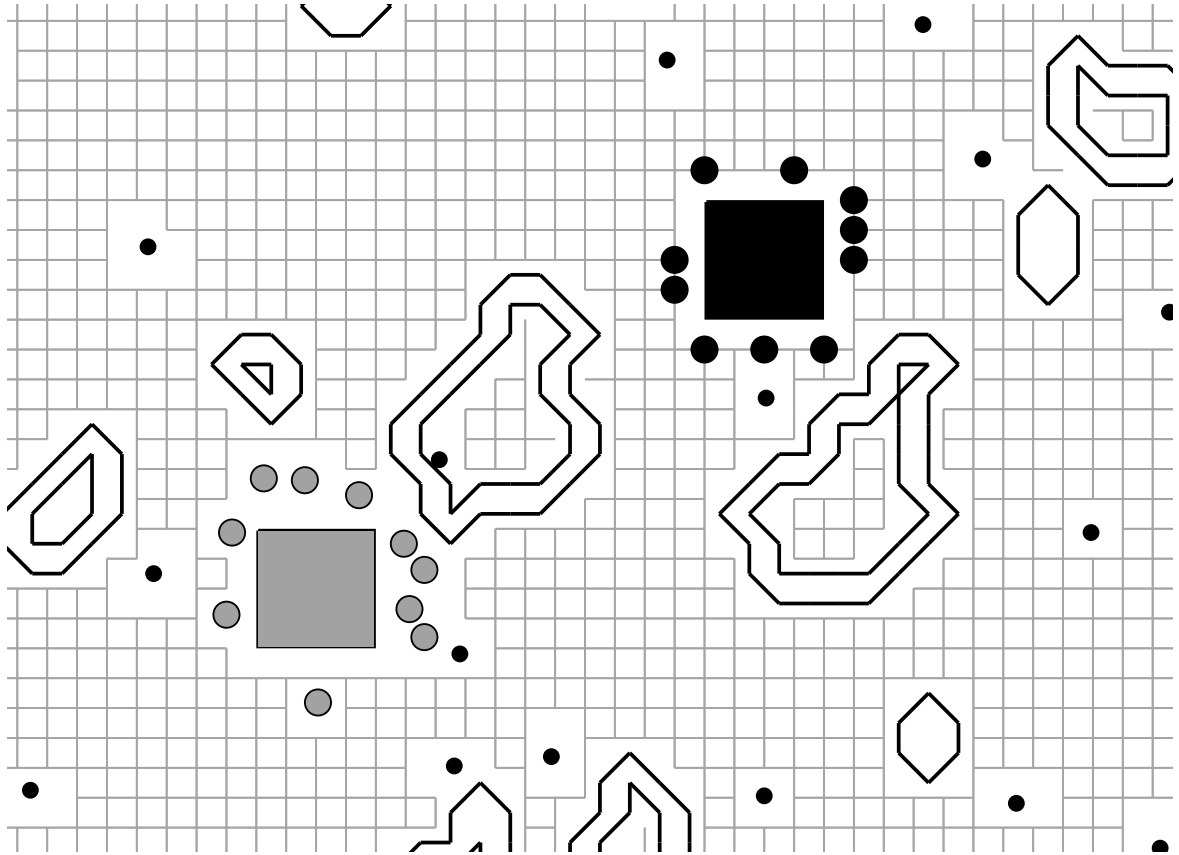


Figure 5.5: Discretization of a graph used for searching. Instead of tile boundaries, connections of tile centers are shown. The connections represent all possible paths of player's units (black).

An algorithm used to create the group is described on the Fig.5.6. This algorithm produces a group where for each unit in the group exists another unit of the group not farther than the distance  $d$ . The smaller the  $d$  is, the bigger number of smaller groups are found. The parameter  $d$  can be set manually according to results of previous experiments. Lonely units having no other units within the range  $d$  are treated as groups having only one member.

This abstraction makes the decision-making much simpler, because it separates the decision making from the low-level operations, like navigating single units or attacking opponent units. The groups are simply ordered to reach a specific destination, or to attack an enemy unit group, and all path-finding and attacking operations are encapsulated inside the unit group object.

---

```

function MAKE_GROUPS(unit_list, distance) sorts units in unit_list into groups
group_list ← empty
while NOT EMPTY(unit_list) do
  new_group ← REMOVE_FIRST(unit_list)
  for each unit in unit_list
    if (CLOSE_TO_ANY_UNIT_IN_GROUP(unit, new_group, distance))
      new_group ← INSERT(unit)
      unit_list ← REMOVE(unit)
  group_list ← INSERT(new_group)
return group_list

```

---

Figure 5.6: Making unit groups according to their positions. The function `CLOSE_TO_ANY_UNIT_IN_GROUP(unit, group, distance)` decides if the distance of a unit to the closest object of a group `group` is lower than the parameter `distance`. The function `INSERT` inserts an object into a list, the function `REMOVE` removes an object from a list.

An example of this method is shown on the Fig. 5.7 a, e). Units on the figure were sorted into groups, where center of each group is marked by a black cross.

### 5.2.3 Opponent Modeling

To describe the opponent's units distribution over the game map, the same method like in 5.2.2 can be used. Describing opponent's units like unit groups having a specific size and a position simplifies the decision making, which can be implemented in a higher application layer.

The decision making algorithm works with the unit groups, so it can order a group to attack an another group, and does not need to care about the individual units inside the group. A single unit with no other units nearby is treated as a group having the only member. The recognized enemy groups and their positions marked with the gray crosses are shown on the Fig. 5.7 d) and on example Fig. 5.10.

The enemy unit grouping algorithm should be applied periodically to have the best description of the game state. The enemy units of the previously detected group can move in very different ways, so the new group abstractions have to be found every  $t$  number of



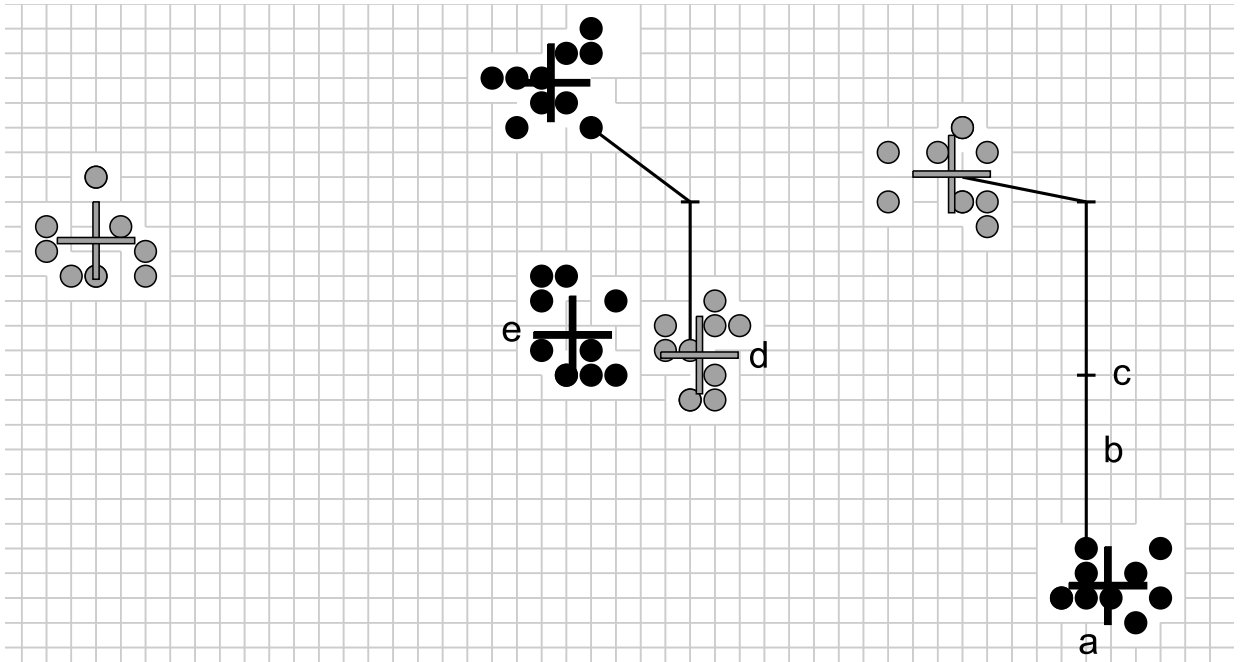


Figure 5.7: A testing scenario to explain unit grouping and group path-finding: Player's group of units with the cross indicating the group center (a), a path of the group leading to the nearest target (b), one of the path way point (c), an enemy group abstraction and its center (d), a group having enemy units nearby has no path, it directly attacks enemy units (e)

game windows. The value of the parameter  $t$  can be chosen according to an experiment.

#### 5.2.4 Navigating Units and Groups

When a group is ordered to reach a destination, it has to navigate each unit to the destination and has to avoid any collisions of the units between each other and with other obstacles in the game. Searching for the path for every unit over the entire game map consumes too much computation time and cannot be done in the real-time. Moreover, searching for the whole path does not have any sense because the game world is changing through the time and the paths found could have to be changed a short time after the computation. To solve these problems, the path-finding is divided into two levels. The first level finds a path for a group, the second level looks for the paths for the individual units inside the group.

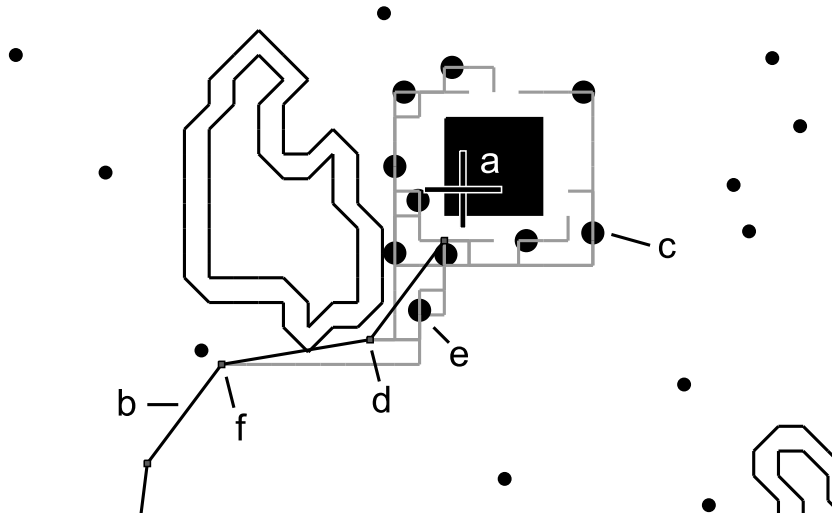


Figure 5.8: An example of a path-finding method for the individual units. A group with the center in (a) has an assigned major path going to the destination (b). The units (c) are moving along their routes (gray lines) to the way points of the major path (d). The units which have reached a way point (e) are assigned a new route to the next waypoint (f).

When a group is ordered to reach a destination, only one path is searched over the entire game map. The path to the destination starts at the position of the nearest unit of the group to the destination. When a path is found, every  $n$ -th point of the path found is marked as a way point. The example of the group route with the way points is shown on the Fig. 5.7 b, c)

The single units have to find a route only to the first way point, or from the one way point to the next one, respectively. The waypoint does not need to be reached exactly, the unit only needs to get closer than the distance  $d_w$ . When the unit gets close to the waypoint, a path to the next waypoint is planned. This technique allows to reduce the state space for the searching algorithm, because the path fragments between way points are very short. Unit routes can be easily re-planned every time they recognize the route is blocked. An example of this path-finding method is shown on the Fig. 5.8.

To navigate the units, two methods were implemented. The first method uses a 3-D reservation table (3.1.2.1). This method is able to avoid collisions between units until there are any unexpected collisions, but when any unit gets blocked or starts shooting instead of moving, it usually blocks other units and all paths have to be re-planned.

Because such interruptions are very common in the game, re-planning of the paths of all units occurs very often and slows the game. When there are two groups of units moving together, this method would not work unless they are merged into one bigger group. To speed up the game, the second method plans the paths of units independently on each other. When a unit is going to move to a next map tile, it reserves that only one map tile to avoid movement of more units to the same place. In case the next map tile is reserved or occupied, the path of only one affected unit is re-planned. The second method does not try to minimize path collisions, so re-planning of the paths occurs more often, but re-planning of the single path takes very short time compared to re-planning paths for all units in a group using the first method.

### 5.2.5 Decision-Making

Each unit group is controlled by a by a simple mechanism based on a finite-state machine (FSM). The behavior of the FSM can be described as:

- If there is an enemy group nearby  $\rightarrow$  attack
- If there is no enemy group nearby, but some group exists  $\rightarrow$  find a path to nearest enemy group
- If there are less than  $c$  enemy groups in the game, enemy buildings are added to a enemy group list, so they are searched by the group. The parameter  $c$  is set to a small value determined by practical experiments.

The last rule has been set according to the real experiments in the game. At the beginning of a scenario, it is better to attack the enemy groups of units using all possible power. When a number of enemy groups drops to  $c$  at the final stage of a game, the buildings are searched and destroyed.

When a group attacks an enemy group, all units try to get close to the nearest enemy unit. The units attack any time they have an enemy object in range, so they do not miss any possibility to hit the enemy. When there are more enemy units in range, the target is selected according to the following priorities:

- The highest priority have the units which have already been attacked by other player's unit

Game	1	2	3	4	5	6	7	8	9	10
Player (P) buildings left	5	4	3	3	5	4	4	4	5	4
Player (P) units left	24	23	27	24	36	23	28	19	19	11
Samle AI (S) buildings left	0	0	0	0	0	0	0	0	0	0
Sample AI (S) units left	0	0	0	0	0	0	2	0	0	0
Winner	P	P	P	P	P	P	P	P	P	P
Game	11	12	13	14	15	16	17	18	19	20
Player (P) buildings left	5	5	4	5	5	3	5	3	3	3
Player (P) units left	17	28	19	34	23	27	31	21	21	23
Samle AI (S) buildings left	0	0	0	0	0	0	0	0	0	0
Sample AI (S) units left	0	0	0	0	1	0	1	0	0	2
Winner	P	P	P	P	P	P	P	P	P	P

Table 5.1: Designed player (P) compared to sample AI player making only random moves (S)

- The second highest priority have other enemy units
- The lowest priority have the buildings

### 5.3 Practical Experiments

In order to discuss the quality of the designed simple player, some practical experiments were made. At first, the simple player has faced to sample AI algorithm moving units in random directions and attacking when having an enemy in range. At second, the simple player has been compared to the player made by the Blekinge team, a winner of 2008 and 2009 AIIDE competitions, which uses the potential fields method to navigate the units.

The parameters used for creating groups and path-finding were set as follows:

- The maximum distance  $d$  of two units to be considered as one group was set to  $d = 64$  (4 map tiles).
- Enemy units were modelled as groups every  $t = 8$  frames
- Distance between two waypoints of the path of a group was set to  $n = 7$
- The maximum distance to the waypoint to continue to the next one was set to  $d_w = 64$  (4 map tiles).

- The maximum number of enemy groups to start attacking buildings is set to  $c = 2$

Comparison with the randomly moving player is shown in the Table 5.1. The randomly moving player does not form units into groups and is not able to concentrate fire of several units onto one enemy unit together. The simple player is able to overplay the sample random player with the average of 23.9 units out of 50 and 4.1 bases out of 5 saved.

Comparison of simple player to the Bleckinge player is shown in the Table 5.2. The Bleckinge player is able to balance its unit positions exactly at the weapon range of the opponent player's units, so it is quite complicated to hit its units while they have a free way to keep stepping back. The Bleckinge player manages to win and save average number of 41.4 units out of 50 and 4.4 buildings out of 5.

The example of the game including the group and unit planned paths is on the Figure 5.9.

Game	1	2	3	4	5	6	7	8	9	10
Player (P) buildings left	0	0	0	0	0	0	0	0	0	32
Player (P) units left	0	0	0	0	0	0	0	1	0	5
Bleckinge player (B) buildings left	5	4	4	4	5	5	5	4	4	0
Bleckinge player (B) units left	38	41	42	37	41	49	42	42	39	0
Winner	B	B	B	B	B	B	B	-*	B	P**
Game	11	12	13	14	15	16	17	18	19	20
Player (P) buildings left	0	0	0	0	0	0	0	0	0	0
Player (P) units left	0	0	0	0	0	0	0	0	0	0
Bleckinge player (B) buildings left	5	5	4	4	5	4	4	5	4	4
Bleckinge player (B) units left	41	47	39	35	47	38	49	38	44	37
Winner	B	B	B	B	B	B	B	B	B	B

Table 5.2: Designed player (P) compared to Bleckinge team player (B). (\*) In this case, the Bleckinge player has not found a path to the last base in time, so the game was a tie (\*\*) The Bleckinge player crashed, so the rest of its units were not able to defend themselves. The case(\*\*) was not used when computing average values.

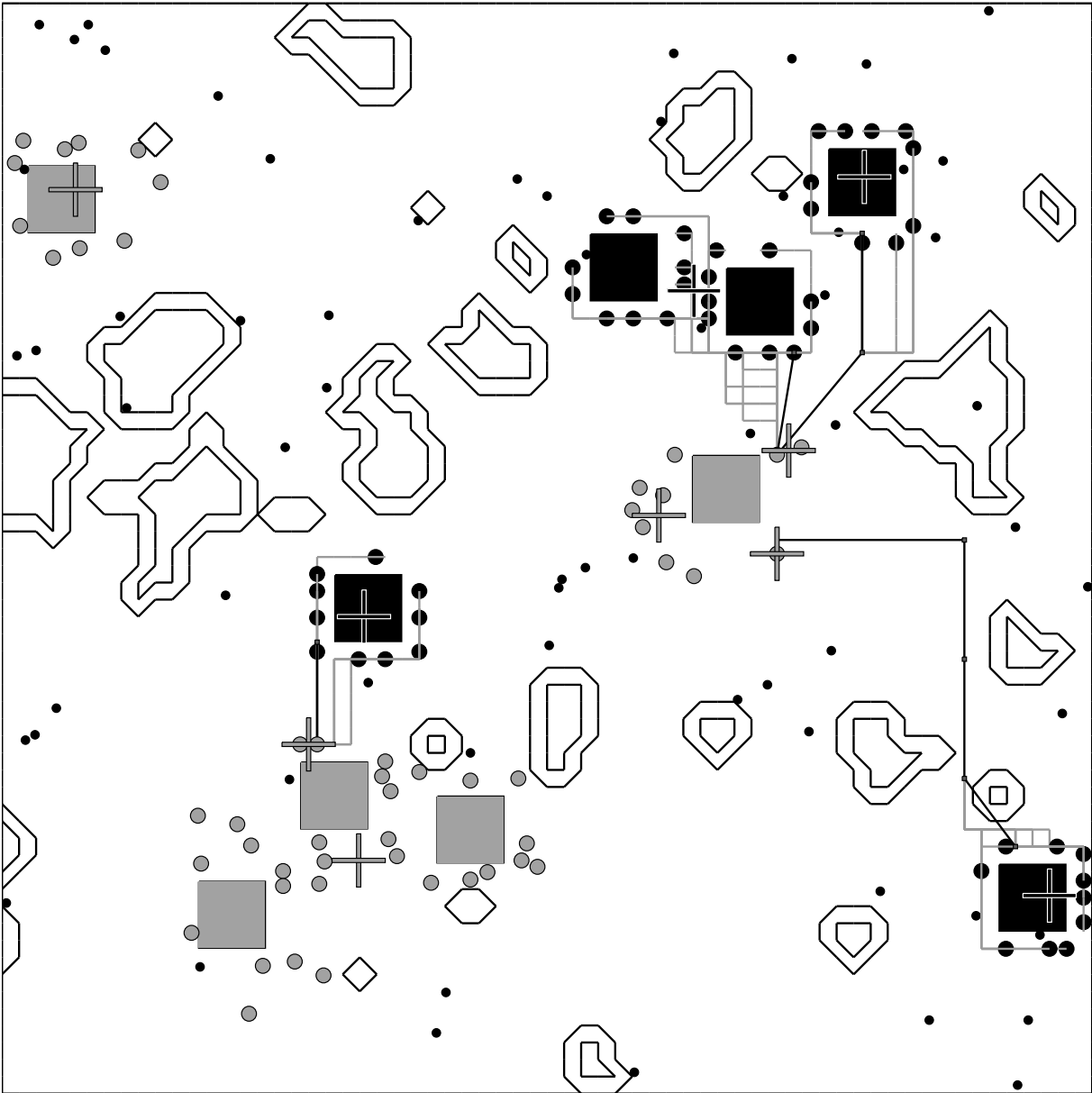


Figure 5.9: Example of a beginning of the scenario: Simple AI player (black) connects its and enemy units into groups and finds the routes to the nearest enemy groups. The centers of the groups are marked as crosses.

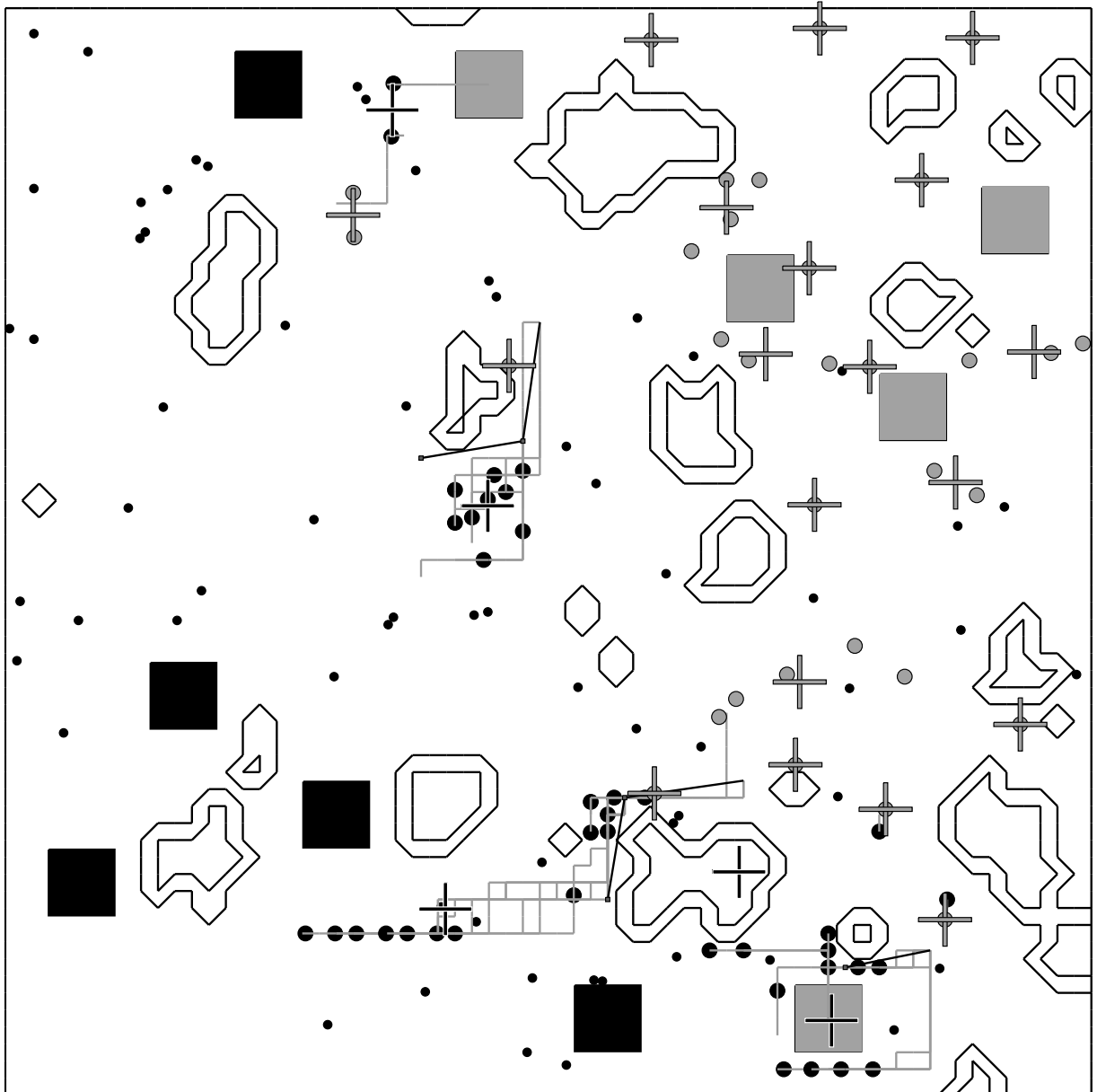


Figure 5.10: Example of a scenario: Simple AI player (black) moves several groups of units towards the enemy (gray).





# Chapter 6

## Conclusion

Strategy games cover large scale of problems, including resource gathering and management, decision-making under uncertainty, creating reasoning abstractions and planning. All these problems appear not only in games, but also in real military simulations, automatized systems for resource gathering or robotic armies. New methods and solutions found with help of computer simulations can be used also in real world systems.

The Open Real-Time Strategy engine is an ideal test bed for new algorithm development and comparison because it is an open-source environment allowing the developers to change the application according to their needs. All computations are performed on the server side, so there is no possibility for the players to obey the game rules and get some additional information they should not have. There is a competition taking place at University of Alberta every year allowing the comparison of solutions made by different researchers.

A simple artificial intelligence player client for the Open Real-Time Strategy environment was developed. It uses a fast implementation of Dijkstra's algorithm for finding paths for the player units. The player combines units into groups to be more powerful and try to concentrate the firepower of more units onto one enemy. The behavior of each group is controlled by a simple finite-state machine.

The ORTS engine uses many objects and structures making the understanding to the code quite complicated. To separate the development the the artificial intelligence from the ORTS, an integration layer encapsulating the specifics the the ORTS has been built.

The developed player was compared to two other AI players. The first one was only the ORTS sample player application, moving units in random directions and attacking units in its range of fire. The sample AI has been overplayed in all 20 of 20 simulations made, where the player managed to save approximately 48% of its units while destroying all opponent's buildings. The second comparison was made with the Bleckinge player, a winner of AIIDE competition 2008 and 2009. The developed player was not able to win the game, but still managed to destroy approximately 17% of its units.

The developed player has been also used to test the functionality of the integration layer which simplifies the AI development and can be used for development of more sophisticated methods. The future development can be focused on the methods like navigating units using the potential fields or Monte-Carlo planning.

# Bibliography

- [1] 2009 ORTS RTS Game AI Competition. Web Page, January 2010. [online] <http://web.cs.ualberta.ca/~mburo/orts/AIIDE09/index.html>.
- [2] R.K. Balla and A. Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of the International Joint Conference on Artificial Intelligence. Pasadena, California: Morgan Kaufmann*, pages 40–45, 2009.
- [3] M. Buro. ORTS: A hack-free RTS game environment. *Lecture notes in computer science*, pages 280–291, 2003.
- [4] Michael Buro, James Bergsma, David Deutscher, Timothy Furtak, Frantisek Sailer, David Tom, and Nick Wiebe. AI System Designs for the First RTS-Game AI Competition. 2008.
- [5] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte Carlo Planning in RTS Games. In *CIG. IEEE*, 2005.
- [6] Nathan Combs and Jean louis Ardoint. Declarative versus imperative paradigms in games ai 1, 2005.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [8] Johan Hagelbäck and Stefan J. Johansson. Using multi-agent potential fields in real-time strategy games. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 631–638, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.
- [9] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

- [10] Nathan R. Sturtevant and Michael Buro. Improving collaborative pathfinding using map abstraction. In *AIIDE*, pages 80–85, 2006.
- [11] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

# Appendix A

## Implementation Notes

The simple player application consists of several implementation layers to separate solutions to different types of problems. An integration layer separates the application from the ORTS engine, so it could be possibly ran also on a different engine after implementing just a new integration layer. An individual layer adds a functionality independent from the integration layer, such as unit path-finding and control. An abstraction layer provides functionality to create and control groups of units.

The application contains about 130 source code files containing 73 classes and some extra data types. To help to understand the code, this appendix describes the most important objects of the application.

### A.1 Integration Layer

The integration layer has to provide information about terrain boundaries, map tiles, number of player and game objects, such as control centers, units and moving obstacles. To provide these information, several objects based on the abstract factory design pattern[7] were created. All objects mentioned below are only abstract classes to define the interface used by higher levels of application, and encapsulate all data specific for a lower implementation layer. The integration layer provides only data acces and control, but does not implement any functionality.

The `GamePlayer` object defines an interface for game player implementation. It defines the `Equals` function used to test if two `GamePlayer` objects refer to the same player in the game; and provides the `Type` function to determine if the player stands for enemy, moving obstacle or client's player.

The `GamePlayerFactory` is responsible for providing information about number of players in the game and creating `GamePlayer` objects.

The `MapBoundary` and `MapBoundaryFactory` provide information about number and positions of terrain obstacles in the game.

The `MapTile` and `MapTileFactory` provide information about map size and all map tile heights and types. The `GROUND` tile type can be crossed by all movable units, the `WATER` and `CLIFF` types can be passed only by air forces (air forces are not a part of the strategic combat scenario).

The `GameObject` encapsulates the information about specific object of the game, and refers to an instance of `GamePlayer` object to identify the owner of the object. It provides functionality for reading its parameters like position, attack range, maximum speed, size etc. and simplifies controlling its movement and attack actions. The `GameObjectFactory` is responsible for monitoring new objects in the game and creating the `GameObject` instances referring to valid `GamePlayer` objects.

## A.2 Individual Layer

The individual is built on the top of the integration layer and adds the functionality for unit control and pathfinding.

The `StrategyObject` is a proxy class for `GameObject` and adds the possibility to control unit movement along its planned path.

The `StrategyMap` is a two-dimensional map of `StrategyMapTile` objects which stores information about passability of the map tiles. The `StrategyMap` is used for game map discretization and rendering the game objects onto the map. The `TimeStrategyMap` is a 3-dimensional map storing information about tile occupation through the time.

Starting state	Event	Next State	Action
Idle	GroupInRange	Attacking	FindAttackPaths
Idle	GroupExists	Approaching	FindPathToNearestGroup
Idle	NoGroupInRange	Idle	NoAction
Idle	NoGroupExists	Idle	NoAction
Approaching	GroupInRange	Attacking	FindAttackPaths
Approaching	GroupExists	Approaching	NoAction
Approaching	NoGroupInRange	Approaching	NoAction
Approaching	NoGroupExists	Idle	EnterIdleMode
Attacking	GroupInRange	Attacking	NoAction
Attacking	GroupExists	Attacking	NoAction
Attacking	NoGroupInRange	Idle	EnterIdleMode
Attacking	NoGroupExists	Idle	EnterIdleMode

Table A.1: Finite-state machine definition used for decision-making

The `DijkstraPathFinder` object is used to find paths between two points on `StrategyMap`. The `DijkstraTimePathFinder` implements the 3-dimensional reservation table searching method. Both objects used for path-finding return the `StrategyPath` object storing the information about the path found.

### A.3 Abstraction Layer

The abstraction layer provides objects for grouping the units into groups. The `UnitGroup` object implements functionality to find a path for a group and controls the units inside the group. The groups are created by the `BasicGroupManager` object.

The `UnitGroup` is given orders by a finite state machine which can be in one of the three states. In the `Idle` state, the group does no actions; when the group is in the `Approaching` state, its units follow the main path with way points; in the `Attacking` state, the units of the group shoot the enemy units, or they try to get enemy units into their attack range.

The finite-state machine of an unit group receives four types of events. When there are some enemy unit groups in the game, the `GroupExists` event is received; otherwise, the `NoGroupExists` event is received. When there is an enemy group nearby, the

`GroupInRange` event is received, otherwise, the `NoGroupInRange` event is received.

When entering a new state, specific action is performed. The `FindPathToNearestGroup` action finds a major path to the nearest groups of enemies. The `FindAttackPaths` action finds a path for every unit leading it to the nearest enemy. The `NoAction` action means there is no change of the state, the `EnterIdleMode` action stops all unit group movement. The definition of the finite-state machine used in the game is in the Table A.1.

When there is a small number of enemy groups remaining, the buildings are added to the enemy group list by the `BasicGroupManager`. The buildings are then treated as enemy groups by the finite-state machine.

## A.4 Graphical Interface

The graphical interface is built on the top of all implementation layers. The `StrategyWindow` draws the unit groups, buildings, terrain and unit paths using only a few primitives – lines, circles and squares. The `MapWindow` objects is able to draw primitives into the bitmap on the screen, the `PsWindow` generates the Adobe `PostScript` output.



# Appendix B

## ORTS Installation

A version of ORTS including the developed AI player and other player applications used for testing can be found on the CD attached to this work. To install and run the ORTS on a linux system, follow the instructions below (SUSE 10.1 and SUSE 11.2 distributions were tested)

- In order to run ORTS, the following prerequisites must be installed first:
  - Install the `boost` library
  - Install the `boost-jam` library
  - Install the `SDL` library
  - Install the `SDL_devel` library
  - Install the `SDL_net` library
  - Install the `SDL_net_devel` library
  - Install the `freeglut` library
  - Install the `glew` library
- To compile the ORTS follow these steps:
  - Unpack the `orts.tar.bz2` archive into some directory
  - Enter the `orts` subdirectory
  - Run “`set OSTYPE=LINUX && make`” command

- Run “`make orts`” to compile the orts server
- Run “`make simpleai`” to compile the developed player
- Run “`make sampleai`” to compile the sample random AI
- Run “`make tankbattle5`” to compile the Bleckinge player
- To run the tournament, run one of the following:
  - Run “`game2_orts`” to start the strategic combat scenario without the ORTS graphical output
  - Run “`game2_orts_disp`” to start the strategic combat scenario with the graphical output

In case of problems with the ORTS installation or running it on a different operating system, more information can be found on the ORTS homepage:

<http://webdocs.cs.ualberta.ca/~mburo/orts/>.

# Appendix C

## Attached CD Content

The CD attached to this work includes:

- Electronic version of this document
  - `/doc/bp.pdf`
- Archive containing source codes of the ORTS engine and the implemented player
  - `/src/orts.tar.bz2`
    - \* The orts server is located in `/orts/apps/orts`
    - \* The implemented player can be found in `/orts/apps/simpleai`
    - \* The sample random AI client is in `/orts/apps/sampleai`
    - \* The Bleckinge player is in `/orts/apps/tankbattle5`