NTNU
Innovation and Creativity

Artificial Intelligence Techniques in Real-Time Strategy Games - Architecture and Combat Behavior

Sindre Berg Stene

Master of Science in Informatics
Submission date: September 2006
Supervisor: Keith Downing, IDI
Co-supervisor: Sule Yildirim, Høgskolen i Hedmark

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The Problem Description:

Look into the use of artificial intelligence techniques in a Real-Time Strategy (RTS) game and make an implementation that will advance the state of the art in displaying humanlike intelligence. First, this requires a literature survey in order to figure out the strengths and weaknesses of existing techniques as far as RTS AI is concerned. Next, suggest the approaches that can solve the weaknesses. Finally, incorporate these approaches within an architectural framework and do the implementation.

Abstract

The general purpose of this research is to investigate the possibilities offered for the use of Artificial Intelligence theory and methods in advanced game environments. The real-time strategy (RTS) game genre is investigated in detail, and an architecture and solutions to some common issues are presented. An RTS AI controlled opponent named "KAI" is implemented for the "TA Spring" game engine in order to advance the state of the art in usin AI techniques in games and to gain some insight into the strengths and weaknesses of AI Controlled Player (AI CP) architectures.

A goal was to create an AI with behavior that gave the impression of intelligence to the human player, by taking on certain aspects of the style in which human players play the game. Another goal for the benefit of the TA Spring development community was to create an AI which played with sufficient skill to provide experienced players with resistance, without using obvious means of cheating such as getting free resources or military assets.

Several common techniques were used, among others Rule-based decision making, path planning and path replanning, influence maps, and a variant of the A* search algorithm was used for searches of various kinds. The AI also has an approach to micromanagement of units that are fighting in combination with influence maps. The AI CP program was repeatedly tested against human players and other AI CP programs in various settings throughout development. The availability of testing by the community but the sometimes sketchy feedback lead to the production of consistent behavior for tester and developer alike in order to progress. One obstacle that was met was that the rule-based approach to combat behavior resulted in high complexity.

The architecture of the RTS AI CP is designed to emerge a strategy from separate agents that were situation aware. Both the actions of the enemy and the properties of the environment are taken into account. The overall approach is to strengthen the AI CP through better economic and military decisions. Micromanagement and frequent updates for moving units is an important part of improving military decisions in this architecture.

This thesis goes into the topics of RTS strategies, tactics, economic decisions and military decisions and how they may be made by AI in an informed way. Direct attempts at calculation and prediction rather than having the AI learn from experience resulted in behavior that was superior to most AI CPs and many human players without a learning period. However, having support for all of the game types for TA Spring resulted in extra development time.

Keywords: computer science information technology RTS real time strategy game artificial intelligence architecture emergent strategy emergence humanlike behavior situation situational aware awareness combat behavior micro micromanagement pathfinder pathfinding path planning replanning influence maps threat DPS iterative algorithm algorithms defense placement terrain analysis attack defense military control artificial intelligence controlled player computer opponent game games gaming environmental awareness autonomous action actions agent hierarchy KAI TA Spring Total Annihilation

Acknowledgements

First, some big credits go to my partners in the implementation of the AI controlled player:
Leon Palm, at University of Cambridge, UK <leonpalm(at)gmail.com>
Rune E. Jensen, at NTNU <runeerle(at)idi.ntnu.no>

I would also like to extend thanks to the people in the TA Spring online community for their help and assistance, including but not limited to:
Alexander "submarine" Seizinger, at University of Tübingen, Germany <alexander.seizinger(at)gmx.net>
Tom "AF" Nowell, at Liverpool John Moores University, UK <tarendai(at)gmail.com>
The rest of the people in the TA Spring community, both developers and players, you know who you are. Thanks for participating!

I would also like to thank my thesis advisor:
Sule Yildirim, at Høyskolen i Hedmark <suley(at)osir.hihm.no>

Finally I would like to thank my friends and family for their support and advice while doing this thesis.

Table of Contents

List of Figures

Terminology

Game Environment - A game environment consists of units, buildings, resources, and the map. The map is the playing field - the background for the units, buildings, and resources.

RTS or Real-Time Strategy. It is a genre of wargames where the player focuses on managing resources, building an army and attacking, often all at the same time. [7]

Unit - (For construction or combat) A controllable game entity that can move, such as a tank or boat. Many of them are armed with a weapon, but some are for the purposes of construction or scouting. Every RTS game has several types of units that are used in combat.

Combat Unit - An armed unit for the purpose of warfare. Combat units are essentially military units and can also be referred to as troops, forces or army.

Construction Unit - A unit that can construct other buildings and sometimes units.

Base or Buildings - Buildings in RTS resource buildings, storage, factories for unit production or they are armed defensive structures for protecting the other buildings against enemy attacks.

Factory - A building that can produce units, using resources to do so.

Resource Building - A building that produces or extracts a resource or energy.

Infrastructure Building - Building usually without a weapon which contributes in terms of economy, such as energy, storage, or even research buildings in some games. A research building can provide means of improving your units or improving the efficiency of other buildings.

Defensive Structures - Buildings that have weapons, and are for defense.

Map/Level - The file describing the properties and terrain where a game can be played in. “Map” is comparable to a geographical map, but refers to the actual virtual playing field. The map includes starting locations, and sometimes also include pre-made bases for the players.

Resource - Items of value located on the map, or that are produced. The type of resource depends on the game, but it is commonly energy, or valuable minerals, or other materials. Resources that are located in various regions on a map are usually fought for because they provide an advantage. Resources are needed in the construction of bases and units

Commander - The starting unit in TA Spring, which can construct some buildings and can also attack other units. Commander can also refer to the human player in some games. Not all RTS games have a commander unit.

Raid or Attack Raid - An attack with the purpose of harming the infrastructure buildings of the enemy.
Siege - Attacking with long range weaponry from a long distance sometimes to limit the movement of the opponent.

Unit Type or Unit Movement Type - The characteristic of a unit relating to how it moves. Units can be land-based, sea-based, underwater (submarines), air-based, all-terrain, and there are a various more rare types also. Unit type can also refer to the type of unit in terms of how heavy armor it has, or in terms of its purpose.

Build power - A term relating to construction abilities; how quickly we are able to build. More construction units and factories means we can build faster, this is what more build power means.

Military Power or Might - Relating to combat abilities; how much damage we can inflict and sustain.

Damage Per Second or DPS - A general way of referring to how much damage a combat unit can inflict over time. For example, 10 damage every 2 seconds is equal to 5 DPS.

Area of Effect or AOE - A general type of attack that inflicts damage over an area rather than only on a specific spot. A real world example of AOE damage would be a bomb, as opposed to a bullet which damages a single point.

AI Controlled Player or AI CP - A computer controlled opponent in the role of a player of the game. [3]

Emergence - "The idea of robust high-level phenomena emerging from simple foundations"[4]. Designing for emergence means creating groundwork and architecture such that high-level behaviors can emerge.

Combat Tactics - Combat tactics are methods for deploying combat units in specific situations.

AI Combat Behavior - How the AI controls his combat units in combat situations. This includes both how to select enemy targets to attack and the detailed control while fighting.

Mod - Short for "modification", and has some special connotations in the gaming context. Mods are extensions or replacements for original parts of a game, developed by a third party or individual. In the TA Spring context, a mod is a complete set of definitions of buildings, units, and their properties, making a complete game playable in the engine.

Player - The entity which makes the decisions and controls the actions of the units in a game through an interface. A player can either be a human player playing via the user interface of the game, or an artificial intelligence controlled player playing through the artificial intelligence interface of the game.

Line of Sight or LOS - From the perspective of a single unit, it is how far it is able to see in all directions. LOS is normally limited by a maximum range, and can also be blocked by terrain height differences and other obstacles. It is typical for RTS games to include the combined LOS of each of the players units to determine which parts of the map are visible.

Teching - A part of a strategy which involves focusing on the research aspects of the RTS game from the start of the game, in order to be able to build more advanced units earlier in the game. Typically this part of a strategy results in the player being weaker at the start of game, but stronger as soon as the more advanced units are completed.

Expanding and Expansion - This refers to the act of taking control of new areas of the map. Generally it also involves constructing buildings on these new areas, and these buildings are usually resources buildings. In this sense, expansion also means expanding resource production.

1 Introduction

1.1 Artificial Intelligence and Computer Games

The field of artificial intelligence is a wide science encompassing conventional AI (GOFAI) and computational intelligence, and there is a long tradition of research on analysis abilities, reasoning, learning and many other related topics. One way of describing the general research goal of the field is mechanizing intelligence[1]. If our goal is to be creating a working AI controlled player (AI CP) for a Real-Time Strategy (RTS) game, then the perspective is slightly different. In any given computer game you have a specific environment with its own obstacles and problems that must be solved in order to implement a computer opponent. The problems to be solved are all the practical considerations like how to gather and analyze data about the environment and what kind of actions are possible. What actions are possible are is determined by game settings, the actions of opponents, and the progression of time in the game. From the perspective of the game AI developer, while there is indeed the desire to impress the player with the intelligence of the AI, in practice a lot of time will be spent on applying algorithms to solve practical problems, such as terrain analysis techniques, path planning and pathfinding, and following the rules of the game in terms of what can be built at a given time and where it can be built. This aspect of creating game AI takes place both in the infrastructure required to produce the information needed for mechanized intelligence, and in the various algorithms intending to produce intelligent behavior.

1.2 The opportunities in TA Spring

New and advanced game environments provide ample opportunity to employ methods and architectures from the artificial intelligence field[15]. Computer games is a modern frontier for innovation in fields ranging from technology to art. The games sector is larger than ever, and there is a desire to bring techniques that have been used in commercial game AI out from their closed sources into scientific literature. Many games are released with AIs that are not written about in literature. RTS games often remain closed source and documentation for their AI usually is not publicly released. The demand for research and breakthroughs in the field of artificial intelligence for use in computer games is certainly there, and game AI also presents a environment for testing theories about the nature of intelligence. While closed source development presents an obstacle against testing theories in games made by the big game industry, some tools are becoming available. One of the communities providing such tools is the TA Spring community. In short, TA Spring is an open source real-time game engine with several groups of people and individuals cooperating and competing to develop parts that compose complete games. TA Spring is a RTS; a complex wargame in real-time with 3D graphics. The TA Spring game engine supports different sets of units and buildings known as “mods” that are almost entirely different games. The TA Spring engine allows AI controlled players that are compiled as separate “dll” files, that players can add to their matches. This thesis makes an effort to explain the context of RTS games, and the context and role of the use of AI in RTS games, as well as present an agent architecture and some algorithms solving common RTS problems.

1.3 Other AIs for TA Spring

At the time I started contributing to the effort of creating an AI controlled player for TA Spring,

there were a few individuals in the TA Spring open source community already working to create AI controlled players. Some of the other AIs used lists with building names for determining which buildings to construct and in which order. These lists were created by players who manually select what works best at the start then make a general plan that should work decently in various situation. While this is done so that it works decently for a large amount of game settings, it gets predictable, and fails to take into account the differences in availability of resources among other things. Determining construction orders by following a list also required a new and different list for each mod, and there is also some trouble regarding different versions of the mods. Another weakness of the AIs for TA Spring that were available at the time was the simple way in which they performed attacks against enemy players. They would send units one by one the second they were ready with no regard for the situation, or they would pile up a large army and send it all at once, often straight into heavy defense. Situational awareness is required to give an impression of intelligence, since attacking without regards for the particular situation is not something a human player would do. However, these previous AIs had their strengths too. One such strength was the learning aspect that some of them had in terms of what kind of army to construct, but this learning had the negative aspect of requiring a training period that the player would be responsible for doing[2]. [17] explains how learning can have this negative effect. The same AI is also able to line up buildings in a systematic fashion like some human players do. It was chosen not to develop these two features, i.e. learning unit strength and making neat bases, since it had already been done. These other AIs for TA Spring have also improved and the development of at least one new AI has been started since recently.

1.4 Motivation

There is much progress to be made in the area of RTS AI. When looking at the AIs previously made for TA Spring, there are a number of features and behavior an AI CP could have that were desirable which had not been implemented. The chosen goals can be summarized in two points:

1. An RTS AI CP capable of defeating experienced human players.

2. An RTS AI CP with combat behavior and detailed unit control similar to a human player.

This thesis presents KAI, an AI CP capable of playing the TA Spring RTS game, and the thesis seeks to defend how these points have been accomplished through emerging a strategy through situational awareness and micromanagement of units in combat, and solving practical problems with various algorithms.

1.5 Contributions to the field of RTS AI:

This is not the first case of designing for emergence in games[4], and it is certainly not the first RTS AI. However, I think it is within the scope of a master thesis to take a known concept in literature and apply it in a new way. The research field of artificial intelligence will certainly benefit from more focus on opportunities given by and progress made in new advanced computer games. What is presented in this thesis can be summarized like this:

1. Creating an AI architecture so that the strategy in its totality may emerge, and the infrastructure needed for this to happen.
2. Creating the infrastructure necessary for advanced detailed human-like control of units in combat, and implementing such behavior.

Infrastructure means the algorithms and maintained data structures at the lower levels of code,

such as analyzing possibilities and costs for constructing buildings and units in the game, predicting unit combat strengths, path planning and replanning, pathfinding and influence mapping[3]. Infrastructure is mentioned because achieving advanced behavior with the higher levels of the code is only possible when this infrastructure is in place.

1.6 Method of Investigation

This thesis is centered around the development of an advanced working RTS game AI, and the context and problems involved in doing so. The improvement was made iteratively, with testing starting as soon as the basic behavior was in place. The emergence approach and the goal of achieving human-like behavior by the RTS AI prompted an empirical method and involved a lot of test matches against human players and other AIs. Testing identified the problem areas where the behavior of our AI needed to be improved (as well as fixing bugs). Testing was done continuously during the development both against human players and other AIs, in various settings.

The working process done for this thesis is:
* Get first hand experience of the specific game and actions that need to be taken.
* Identify problem areas where the AI needs better behavior; more effective or more human-like.
* Read and research RTS AI literature and relevant theory from related fields.
* Map selected problem areas into the AI architecture.
* Find and implement a solution for the problem area and present the algorithm used.
* Test solutions in practice against a human or another AI.
* Evaluate resulting behavior and possibly make improvements.

When an algorithm designed to solve something in a particular problem area was in a working (implemented) condition, it was tested to see if it was capable of accomplishing the desired behavior. The program was then run with settings to best illustrate the solution of the problem area, and information about the resulting behavior was collected through several methods: Drawing lines in the game illustrating the decisions made by the AI, was one of the methods. Another method was storing various data generated about the environment as 2D images and looking at them outside the game. Various information was also gained through this testing by log files and by looking at how the AI performed in the game. The last but not least important method of testing was done by playing the game against the AI, and discussing with other players who did the same. This thesis does not go into detail on algorithms that were replaced, but focuses on the approach taken in the improvement and the resulting algorithms.

As previously mentioned, the strategy of the AI playing the game was emerged through situational awareness, and this “emergence” approach influenced how the research was performed. New behavior had to be implemented in such a way that it did not interfere much with the actions of other parts of the program, unless it was desired to do so. For example, the code doing micromanagement of a group of combat units while fighting should in some cases interfere with the planned attack path given to the attack group, while sometimes it would just be dealing with a minor obstacle while continuing on the way to the planned target. In cases like this, part of the method was to determine which behavior was proper in different situations, and determining the priority of actions. For instance, the act of a small attack group avoiding a sudden immediate high threat

takes priority over moving along the planned path to an enemy target, if such a threat appeared on the path. Detailed behavior related to these kinds of situations were often not obvious until the situation arose while testing the AI, therefore testing was a vital part of the research.

1.7 Thesis Structure

After the initial terminology pages and this introduction, the thesis presents the background information of real-time strategy games and the context of computer opponents in them. Effort is then made to explain the common concepts of strategy in real-time strategy games. The next topic is the specific environment of the TA Spring engine and its opportunities and challenges for RTS AIs. Then the AI Architecture is presented and the approach of emerging a strategy through situational awareness and various algorithms is explained. We then move on to discussing the details of novel algorithms in KAI and details of their use. The results and contributions are discussed in the final chapter.

2 Real-Time Strategy Games

2.1 The Real-Time Strategy Game Genre

Real-time strategy (RTS) is the genre of computer games that encompasses war games that happen in real-time. "Real-time" means that there is a continuous flow of time in the game world, so that "thinking on your feet" and responding quickly to arising situations is important. "In a typical RTS, the goal is for the player to collect resources, build an army, and control his units to attack the enemy."[7] Since such games have both an economy aspect and a combat aspect, the attention of the player is divided, and it is considered a hard game genre to master. In order to describe the qualities of this game genre, the origin of the genre and some sample games follow.

2.2 Early History of the RTS Genre

One of the first real-time strategy games was "Dune 2: The Battle for Arrakis", also known as "Dune II: The Building of a Dynasty", released in 1992 by Westwood Studios[9]. Figure 1 shows the game with its 2D graphics and a point-and-click graphical user interface. The game is based on the Dune universe from a series of science fiction novels[10], and was released with better graphics as Dune 2000 in year 2000. It is hard to pinpoint exactly where the real-time strategy genre started, but Dune 2 is a good representative of early RTS games. Since then the genre has developed and is continuing to develop. There are series of games with many sequels and long storylines, as well as new and independent games.

Figure 1: A screenshot from the original Dune 2 RTS game from 1992.

2.3 Examples of RTS Games

Many games have been released in the genre, among the more famous ones are "StarCraft", the "Warcraft" series of games, both by Blizzard Entertainment. The "Command & Conquer" series started by Westwood Studios and "The Age of Empires" series by Ensemble Studios are also prominent games in the RTS genre, and there are many more. Some of these games carry characteristics inspired by other genres. For instance, while in general being a RTS, Warcraft III has special "hero" units with individual special attacks that improve as the heroes are being used actively in combat, which is inspired by the role-playing game genre. Ground Control II by Vivendi Universal Games has no base or resource management, but instead is focused around combat tactics. Another variation can be found in SimCity by EA Games can be played as a real-time game but has no military combat but focuses instead on city planning and economy. A closely related genre is the turn-based strategy game genre, categorized differently from RTS games because the progression of time in the game is stopped while the player contemplates strategies and moves. There are also games that differ from the norm in other ways, such as Homeworld by Sierra Entertainment. In Homeworld and its sequel Homeworld 2, players create and control an army of spaceships and buildings in space, and the game environment is truely 3D. An attack can come from above or below or any other direction, making the game more complex than normal RTS games[14]. The personal computer is the most popular platform for strategy games, but some of them are also released for console platforms i.e. video game machines connected to a TV such as Sony Playstation. A survey made for The Entertainment Software Association in USA approximates that 30.8% of sales of games for personal computers are strategy games[8].

2.4 The Original Total Annihilation

Total Annihilation by Chris Taylor and Cavedog Entertainment was made in 1997, and was one of the first RTS to feature 3D units and terrain to some extent[9]. It became popular at that time and stayed popular for a long time, partially also due to the lack of restrictions on production of military units and weapons. Most strategy games have restrictions on number of units and some sort of costly upkeep for having units. For example, Dune 2 had a 25 as the maximum number of player controlled units, and Starcraft and the Warcraft series has various kinds of upkeep related game entities such as farms required to feed soldiers. Although the original Total Annihilation (TA) did in fact have a restriction at 250 units, this combined with the lack of other upkeep requirements was enough to contribute to the game being popular. A screenshot of TA is shown in figure 2.

Figure 2: A screenshot from the original Total Annihilation RTS game from 1997.

2.5 A Typical Sequence of Events in a RTS Game

At the start of a game that is being played, the player typically has few resources and units, but has the possibility of gaining more resources. The immediate tasks at hand are constructing or expanding a base, for the purposes of gathering resources and spending those resources for producing an army. The player also should make sure he will be able to survive the beginning by producing units or defensive structures, because attacking as early as possible is common. Some games have a storyline with specific objectives that are to be met, but in general it is all about control of resources, putting the units to good use and achieving superior military power. The game is usually ended when a player has destroyed all his enemies, or when a player achieves some predefined objectives. In a single player game these objectives are usually defined in a mission briefing following a storyline and they are usually accessible while playing the game in a list of "mission objectives". The mission objectives follow the general storyline of the game, and can be some detailed action that has to be taken, or just something general like "destroy all enemies". It is often possible to know who will win the game by looking at who has the advantage in resource income and control of the most areas, and this is often possible to see some time before the game ends to an observer. There are several guides online explaining how to play the most common RTS games, and games that are released commercially usually come with a pamphlet explaining the basics of playing the game.

3 Strategies in RTS Games

3.1 Strategies

The meaning of the word strategy in the context of an RTS game is the high-level decisions about how to play (and possibly win) the game. For instance, a high-level strategy for a player could be what is known as the "rush" strategy. Rush refers to rushing/hurrying into one or more early attacks. Most if not all RTS games have properties enabling this strategy, and variations of this are by far the most commonly used strategies. Rush implies hurrying to produce an army as early as possible in the game and sending it immediately in an attack, and it is usually composed of basic land-based combat units. While the construction of resource buildings and other infrastructure buildings might suffer due to the priority of constructing armies early, the advantages that can be gained with an early attack are more important. Such advantages can include intercepting the enemy's attack, crippling the enemy's economy, or establishing control over an area of the map, or blocking the enemy from leaving his corner of the map. While having combat units early in the game is normally advantageous, a player is only rushing if he uses them to attack the enemy directly. If an attack fails and the defending party ends up having a major military advantage, then a counter-attack is in order. Performing such immediate counterattacks are often referred to as "countering".

Another approach is to stay defensive while producing the forces necessary to gain an advantage such as air superiority, with the goal of succeeding with a surprise attack. Keeping the enemy from discovering your strategy might be advantageous so that the enemy is unable to counter the strategy, in our case by building defenses against air units. If successful, the player should be able to surprise the enemy and cripple the enemys economy with an air raid. There are various ways of performing surprise attacks and otherwise avoiding enemy defenses.

"Siege" is another general strategy related idea and it involves playing the entire game defensively utilizing long range heavy weaponry. The weapons may be defensive structures, or slowly moveable heavy combat units. The idea is that the enemy can be suppressed by heavy fire at long range without heavy losses on the players side. If a siege attack is being led against the enemy, only alternatives for the enemy is either to attack the siegeing forces, or to suffer a slow death from the long range weaponry. When siege or another way of attacking is not used as a high level strategy for the entire game, but instead is used simply as a method of deploying units in specific situations, it should be considered a tactic. Tactics is military terminology, referring to the way that a single battle is fought. A real world example of siege would be artillery bombardment or long range missiles.

A player who gets into a defensive position but is unable to expand or perform important attacks, is commonly referred to a as a player who is "turtling". There are many improvements and additions that are possible to add to an attack before sending it off to fight, but making improvements takes time and resources. While defensive structures are generally cheaper and more powerful than attacking forces, waiting too long is a losing strategy in the long run because you are then allowing the enemy reign free in the meantime. Being able to harass the enemy while increasing your own

might is advantageous. Another strategy which involves a play style which is easily mistaken for turtling, is "teching". Teching can involve prioritizing research from the very beginning, in games that have research. In general, teching involves being passive and spending resources on enabling the player to construct more advanced combat units. Early in the game it is a disadvantage, since resources are being spent on creating more advanced buildings than are common. The advantage of the teching strategy becomes apparent as soon as the player has produced the few first advanced units, since these are generally stronger than the less advanced units. The most major weakness of the strategy is that resources are not being spent on units early in the game, which leaves the player vunerable to early attacks.

3.2 Resource Control

Having control of resources is an important aspect of strategy in RTS games. While there is almost always a basic amount of resources at the starting location for the player, as time passes it becomes more and more advantageous to have control of additional resources which may be located in separate locations of the map. In order to safely extract resources in a given area, the area should be defended or otherwise safe from enemy forces. Controlling a larger area requires more forces and so it is a risk. But controlling more resources increases the players construction capabilities, since all construction requires an amount of resources. Covering a larger area is referred to as "expanding", and this is an important aspect of RTS games in cases where the game is not directly won by the initial rush attacks.

3.3 Considerations of the Game Environment

There are also considerations relating to the properties of the map, for instance holding the high ground or a hill gives a combat advantage in most RTS games. What strategies are possible, depends on various RTS games and game settings such as availability of resources and the terrain of the map. For example if a lake or river blocks passage between the player and his enemy, then having only land-based units will not enable the player to attack. In such cases where the land-based path is blocked, the player has to either construct a means of transport, or he has to build units capable of moving through the difficult terrain. This is an example of restrictions on the gameplay resulting from game settings. The game developers usually effort to balance the units in the game so that no specific strategy is superior to all others. Balancing can mean adjusting the speed, firepower, robustness or cost of the units. Typically, units that are more maneuverable are weaker and vice versa.

3.4 Micromanagement

The term "micromanagement" or "microing" in the RTS game context, is commonly used to describe the task of controlling combat units in detail while they are in combat. Normally, combat units will automatically shoot at anything hostile that is in range, but an advantage can be gained by moving them to another position or making them attack certain targets before others. One example of this is moving your units on top of a hill where they can gain benefits both in terms of firepower and having longer line of sight, and in addition if the enemy wants to get closer he will have to climb the hill while being fired upon. Another example is when there is a difference in maximum attack range, like using archers to kill swordsmen. In this example, the archers would

benefit from being at range while the swordsmen would benefit from being closer, so the player which is able to position his units better will be able to utilize them more efficiently. In most RTS games the units have basic behavior, for example a unit that is fired upon will attempt to move close enough to the enemy unit to be able to attack it back. Human players learn over time how to control their different types of units in the most efficient way, but traditionally RTS game AI will rely on the basic behavior and not control units beyond telling them to where to attack and when to retreat.

The concepts explained in this chapter about strategies in RTS and related topics are mostly from first hand experience in online gaming communities such as Battle Net[11] and the TA Spring community[12], and from discussions with other gamers.

4 Artificial Intelligence in RTS Games

4.1 The Role of AI in RTS Games

A "bot" is an intelligent artificial player emulating a human player in the game environment[19], but this terminology is used to describe AIs in First Person Shoot-em-up (FPS) games. The directly corresponding entity in an RTS game would be something like the individual combat units or groups of them, since a player takes the role of the commander of many units in an RTS game, and not just one unit. However, the terms "AI" and "bot" are sometimes used interchangeably. While RTS games have more units and unit types, the actions that can be done by each individual unit are more limited. For instance, a bot in an FPS game can strafe, jump, sometimes dodge projectiles, pick up various items and interact with the environment and items in other complicated ways. A normal combat unit in an RTS game can not do much more than move around and shoot in that respect, though there are a few games where units can activate buildings or storyline related items. The detailed control in terms of moving and attacking is more limited than that in an FPS game. Most games, whether they are FPS games or RTS games, have computer controlled opponents that the human player plays against, in one form or another. One important property of RTS AI controlled players though, is the large amount of units and economic decisions that are required. Designing an RTS AI is no easy task. [16] concludes that "[RTS] is one of the hardest genres to develop AI for and will test your abilities to the limit."

4.2 Computer Controlled Opponents in Single Player Mode

Most, if not all RTS games can be played in a single player mode where the player faces a computer opponent. Traditionally the behavior of the computer oppoent was simple, such as having the computer opponent attack at intervals in small waves of units, putting a sense of action into the game. In most single player game situations in RTS, the human player starts with considerably less resources than the opponent. The task for the player is to expand his/her military power and wipe out the enemy. So the purpose of the computer opponent is to provide enough resistance for the player to feel there is a purpose. Often the enemy has a predesigned base and a continuous production of units. A common way to implement the computer opponent is to have scripts associated with locations and points in time. These scripts define events in the game and are often narrated making a contribution to the storyline. A typical thing done by such a script is to place enemy units at a location or to taunt or warn the player. The game StarCraft even included a "Campaign Editor" that allowed players to create their own maps/levels that included scripted triggers[13]. But in StarCraft and many other RTS games, there are also algorithms controlling the computer opponent behavior that go beyond simple triggers and response mechanisms. This behavior might include forming attacks and sending them to a poorly defended part of the players base.

4.3 Single Player Mode and Artificial Intelligence

Referring to the RTS computer opponent as just "the AI" is common, and there is a reason for this. One of the ways of making the player immersed/engaged in the game play is to have the computer opponent respond to his actions in a human-like fashion, creating the impression of intelligence. A goal in the development of an AI is to make the human player identify the computer opponent

as an opposing player and not just a set of algorithms. One should remember that the most pressing requirement as far as the game developer is concerned is subjective user satisfaction. Scripted and well narrated events following the storyline of the game universe contribute to the gaming experience, and behavior consistent to the storyline on behalf of the computer controlled units is desired. The development of a computer opponent for a game requires that most of the game engine has been completed, so AI development is often hurried towards the end of the process. This is one of the reasons why traditionally RTS AI has been predictable using scripted behavior, since this is low cost and reliable. But there is also the desire to create computer opponents that show intelligence while providing challenging resistance.

4.4 AI Controlled Opponents in Multi Player Mode

Most RTS games offer some sort of multi player mode intended for fights against other human players in a network of computers, but most also offer AI controlled opponents in the role of a player so that a single human can play in multi player mode, or a team of humans can play against AI controlled players. This kind of match is sometimes referred to as a "skirmish", meaning a small battle. Multi player games are generally for the purpose of competing and seeing who is the better player. The setting of the match is generally on even terms, so that the skill of the player is compared on a "fair" basis to that of a AI controlled opponent. Creating a computer controlled opponent which has sufficient skill to put up a good fight against a skilled player is difficult, so various means of tipping the scales are often used. For instance, an AI opponent in WarCraft III multiplayer mode set to "hard" difficulty will receive twice as many resources as human players from all sources. Another trick benefitting computer opponents is having information about the players movements and units available at all times without having to travel to discover his position. Creating an RTS AI that works in a match on even terms is a more complex task than just defining triggers and basic attack behavior as can suffice for a single player campaign, but upon completion it can also be used in single player mode. After all, it is easier to adjust it to make it play worse than it is to make it play better, and any necessary storyline events can be accomplished as separate script, or even as instructions to the AI.

4.5 Other AI Related Features in Games

Techniques from the AI research field are not necessarily used only in AI opponents in the role of a player. For instance, an implementation of flocking can reside in the RTS game engine so that it is in effect when a human player is moving a group of units, such as in Age of Empires[3]. Another example is the animal behavior of the animals in the persistent virtual world in the online game Ryzom[20]. The animal behavior includes moving as a herd, grassing, flocking, and there are also predator animals hunting other animals. There are also other simple algorithms that are AI related that may be implemented in an RTS game engine, such as dead reckoning for combat units that are shooting. Dead reckoning is a basic way of approximating where an object moving at constant speed will be after a certain period of time, and can be used for approximating where to aim ballistic weapons. Approaches to keeping units moving in formation can also be considered to be AI related, if the means for doing so imitates simple animal behavior or otherwise shows intelligence. Game AIs have attempted to take advantage of most techniques relevant to the general AI research field in one form or another[17].

4.6 Pathfinding in Real-Time Strategy Games

Most games require pathfinding, which in simple terms is to find the shortest path between two points. The most common usage in RTS games is moving a unit from one area to another. Real-time strategy games are games where large amounts of units require pathfinding, and these actions are initiated by the human player as well as the computer opponent. The units may be placed in positions in the game environment. As the player gives the order for one unit to move from its current position to a new position, the unit must move along a path in order to arrive at the destination. This pathfinding is usually implemented in the game engine and available in the client that the human player plays though, and a path is generated when a player selects a target or destination for a unit or a group of units. Figure 3 shows a section of the screen in the game Civilization IV as the player gives the order to move a Warrior unit to the forest-covered area in the bottom-left corner. The white encircled numbers are indications of the time it will take to go along that path. The environment in the strategy game Civilization IV has tiles which determine the space that units may stay in, and since this results in relatively few choices of movement the execution of the pathfinder search is cheap. In comparison, if the resolution of the tiles were at lot higher, for example the same resolution as the pixels shown in this image, pathfinding would be more computationally expensive. It is common to have a separate low resolution virtual representation of the map in this manner so that the computation can be more efficient. Reducing the resolution of the search area for a pathfinder results in faster execution but lower accuracy. Please note that Civilization IV is not a real-time strategy game but a turn-based strategy game.

Figure 3: Example of a generated path to a destination in Civilization IV.

There are many approaches to pathfinding, often determined by the virtual environment. A pathfinding algorithm could for instance give a straight line to the destination, and have the algorithm controlling unit behavior take care of avoiding obstacles along that path. A simple approach like this might work in simple game environments, but most other game environments are more complex regarding pathfinding, often with a 2D or 3D structure of walls and obstacles, and terrains with different travel cost (for example muddy terrain). In fact, many advanced methods are used to determine a path that an agent might move through when a player provides it with a destination.

The pathfinding algorithm has to be efficient since it will be used frequently, and it often runs on demand so that it can not be precalculated. If it is not efficient, the person playing the game might experience choppy performance if he moves a large amount of units at once, or when the AI controlled player might require several pathfinder calls in a short time interval. Certain tricks can be employed to reduce this, for instance spreading the calculation of the individual paths over a time frame, or sending several units through the same approximate path. But the fact that the algorithm should be efficient remains.

By placing path nodes all over the area that units may move through, you enable use of an algorithm for finding a path from its current location to a destination. In the FPS game type, path nodes are somewhat harder to calculate automatically, due to the 3D game environment. The path nodes in FPS games are commonly referred to as waypoints, and are so used in the FPS game type because the geometry of the game environment in FPS games is often quite complex (having multiple platforms in the vertical direction). In RTS games the game environment is usually 2D, and have units moving mostly on a 2D surface with height differences, giving the impression of a 3D environment. A 2D array with lower resolution than the virtual map can be used as the data for a pathfinder, or it can be represented as a graph of nodes generated from the map.

In a 2D environment or a 2D surface in a 3D environment generating waypoints is easy; they can be placed evenly all over the surface, and they can be stored in a low resolution 2D array. It might be necessary to analyze the type of terrain within the passable areas, such as water, lava or other obstacles. Also, it might be necessary to take into account the cost of moving through a certain area which has a rocky or muddy terrain, or other features that make traveling there less feasible. Furthermore there are cases where some areas may be too steep to pass, and have to be avoided when planning the path. Taking into account properties of the terrain is called Terrain Analysis, and may also include analyzing the changing situation in terms of deformable terrain and actions of the players.

The full 3D environment is another environment setting, with multiple paths even vertically. This is mainly related to other game types than RTS, but there are examples of RTS games which require 3D pathfinding, such as HomeWorld[14]. While the waypoint-based method can also work well here, there's another method which utilizes the geometry. By flooding the map from the inside with boxes or other simple shapes that fill the areas, one can create a simpler model of the 3D area that contains all the parts where one is free to move around. It is possible to make all these area parts convex, and in doing so you enable certain more efficient method of pathfinding. [19] presents an efficient way for pathfinding and bot behavior in fully 3D environments using portals between convex open areas within the geometry. This relates to the Binary Space Partitioning method of storing 3D geometry, which is not a topic of this thesis because it is generally not applicable to pathfinding in RTS.

Another method for generating the data structure used in pathfinding is creating Voronoi diagrams[21]. Voronoi diagrams are generated with a spatial partitioning algorithm that essentially creates exactly one path around each of many obstacles in a diagram. This is useful because the resulting paths are efficient paths for navigating the terrain, or they can be used to generate paths

with a pathfinder algorithm.

All the pathfinder related methods mentioned so far in this chapter can be seen as methods for generating waypoints. A waypoint is a point that is passed through on a path to a destination. Each waypoint is a node in a graph that is searched through with an algorithm. In the graph search, the nodes represent the waypoints, the edges are the travel cost between waypoints, and the goal node(s) is the destination. The goal in pathfinding is to find the best path through a subset of the waypoints in order to get to the destination. There are many ways of determining what paths are the best, but the most common criteria are distance travelled, and the speed of travel between waypoints. There can be additional bonuses and costs in terms of the elevation at the positions passed through, or enemies in the area can affect the cost of travel.

There are several algorithms for finding the best path. The most common algorithm is the A* algorithm. A* is a graph search algorithm which employs the best-first strategy. At each node in the graph, it evaluates each node at the next level using a heuristic estimate. A* then selects the one which seems have the shortest distance-cost to the goal node. The result is the best path from the origin point to the destination, through waypoints. There is a certain condition that have to be met for the A* algorithm to be applicable. The condition is that in order for A* to be efficient it requires a working heuristic function returning an approximation of the cost of travelling from the intermediary waypoints to the destination, and the computational efficiency of the best-first search depends on the accuracy of the heuristic function. A* with no heuristics function for the best-first guessing is called the 'A' algorithm (similar to Dijkstras algorithm). This algorithm is also able to find the best solution, but uses a breadth-first approach since it lacks a guessing mechanism for selecting the best node first.

5 TA Spring

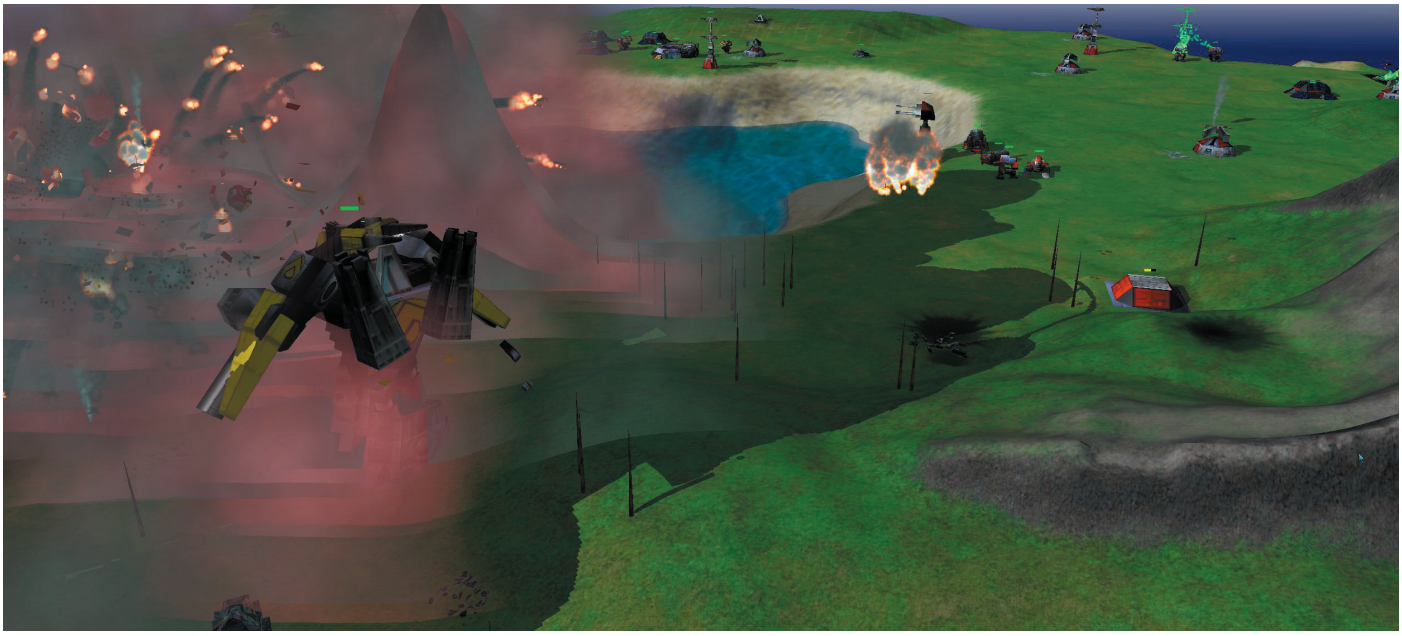


Figure 4: A screenshot from TA Spring running the Absolute Annihilation mod.

5.1 Total Annihilation Spring

Total Annihilation Spring, also known under the names TA Spring, SpringRTS, or even just Spring, is an open-source project developing a real-time strategy game engine[12]. The engine is heavily based on the original Total Annihilation game and is made mostly in C++, but supports hardware accelerated 3D graphics and is multi-platform. A screenshot from the game while running the Absolute Annihilation mod is shown in figure 4. There are several aspects of the TA Spring engine that make it unique. For one, the maximum number of units that a player can control at any given time is currently 5000, and is being increased to 10000. Having many mods playable in it, the physics simulations of the environment, and the deformable terrain are all rare features in modern RTS games, contributing to making it popular compared to other open source games. At the time of writing, the TA Spring project is in beta version 0.72b. There is a large community of developers and players involved online in the project. The engine and games playable in the engine are inspired by the original Total Annihilation, although the community are continuing to develop new things and replacements for old things so that the similarities with the original Total Annihilation have faded.

5.2 TA Spring Game Development

There are four main categories for development of TA Spring:

* Developing the TA Spring game engine (and lobby client, GUI, et cetera).
* Designing "mods" for TA Spring, and the units the mods consist of.
* Designing maps for TA Spring.
* Developing an AI controlled player for TA Spring.

These categories and their relevance to this work will be explained in the following chapters.

5.3 The TA Spring Engine

The main component of the TA Spring game architecture is the TA Spring engine, which is the program that runs the game. The engine source code consists of thousands of lines of code and hundreds of classes that are available on an online code repository[22]. The engine runs as a networked server even in single player mode, where the player(s) connect through the client part of the program. The most common way to play the game is through the online multiplayer lobby program that is distributed with the default game package. The lobby client program enables a player to host a game, and other players can join the game that is being hosted. Players can choose whether to play or to be a spectator. Each player is able to add AI controlled players to the game, these AI controlled players will then run as a program on the computer of the human player that added it. A player can play the game while running multiple AIs. There are settings for putting players on teams, and players can also share control over the same units. The host of the game is able to choose which map to play on, which mod to play, and various other options and settings for the game to be played.

5.4 Maps in TA Spring

In games, The term "map" - similar to in the real world - describes a certain geographical area. In TA Spring, the map determines the terrain height, texture, water, locations with amounts of metal, and various other minor properties of the playing field such as the strength of gravity and wind.

There are hundreds of maps available to be played on in TA Spring. Some of these maps are inspired by real world places, either created from maps of countries or generated from satellite imagery. Other maps are created from scratch in a map editor program. Figure 5 shows a screenshot of a map called "DeltaSiege" made by "IceXuick".

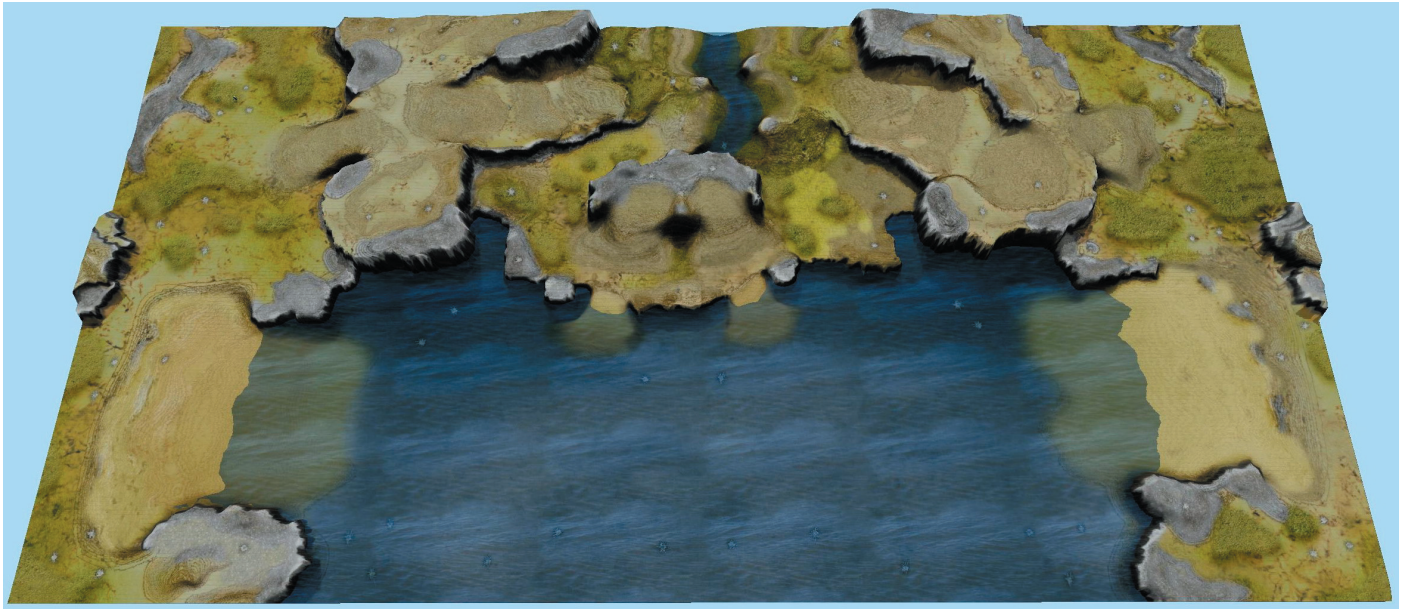


Figure 5: A rendering of the DeltaSiege map with textures.

Actual map files for TA Spring are in specific formats that are compiled from a collection of smaller files. A text file describes names and general properties like predefined starting locations, as well as multipliers determining movement speeds for various unit movement types and forces of nature

such as wind strength, tidal force strength, sunlight and gravity. Bitmap images (two dimensional matrices of integer values) describe the details of the terrain. One of these bitmap files is the "height map", where the data at each pixel represents the height of the terrain at the same coordinate on the playing field. Another bitmap is the map describing where the metal resources are located. The third bitmap is a graphical image with the textures that cover the ground in the game. These files are compiled and compressed in a single file which can be downloaded for the game.

The map selected when playing a game of TA Spring has major implications on how the game will be played. The main considerations are the shape and properties of the terrain that affect possibilities for building structures and moving units, as well as the placement and concentration of resources on the map.

5.5 Mods for TA Spring

A "mod", normally short for modification, is in the TA Spring context a set of unit and building definitions. A single mod defines all the buildings and structures by a list of unit definitions, and it is essentially a separate game playable in the TA Spring engine. A unit definition describes every aspect of a single unit, including its size, power, weapon type, movement type, cost, build time, graphics and many other things. These properties are stored as tags, and here is the complete tag list describing a single unit:

```
acceleration ActivateWhenBuilt AutoHeal badTargetCategory BrakeRate BuildCostEnergy
BuildCostMetal Builddistance Builder BuildPic BuildTime canCapture candgun CanDropFlare canfly
canhover canmove canResurrect Category CloakCost CloakCostMoving commander Corpse cruisealt
DamageModifier Description Drag EnergyMake EnergyStorage EnergyUse ExplodeAs ExtractsMetal
FlareDelay FlareDropVector FlareEfficiency FlareReload FlareSalvoDelay FlareSalvoSize FlareTime
floater FootprintX FootprintZ frontToSpeed HideDamage HighTrajectory hoverattack IdleAutoHeal
IdleTime init_cloaked isAirBase IsFeature istargetingupgrade kamikaze KamikazeDistance
LeaveTracks MakesMetal Mass maxAcc maxAileron MaxAngleDif maxBank MaxDamage maxElevator maxPitch
maxRudder MaxSlope MaxVelocity MaxWaterDepth MetalMake MetalStorage MetalUse mincloakdistance
MinWaterDepth MovementClass myGravity Name NoChaseCategory Objectname onlyTargetCategory
onoffable RadarDistance RadarDistanceJam SelfDestructAs selfdestructcountdown SightDistance
SmoothAnim SonarDistance SonarDistanceJam SoundCategory speedToFront Stealth TEDClass
TidalGenerator TrackOffset TrackStrength TrackStretch TrackType TrackWidth transportcapacity
TransportMass transportsize turnRadius TurnRate Unitname Upright Waterline weapon WeaponSlaveTo
WindGenerator WingAngle WingDrag WorkerTime wpri_badTargetCategory wsec_badTargetCategory wspe_
badTargetCategory YardMap
```

Some tags are necessary, while others are not, depending on the kind of unit being described. Most of the tags have an effect, to varying degree, on how that unit may be used and how valuable it is. [18] describes the details of each tag.

There are many of mods created and in development for TA Spring, in various stages of completion. Some are inspired by the original Total Annihilation, like Absolute Annihilation which has been rebalanced and has many new units, and the space ship based Final Frontier. The OTA mod seeks to emulate the original TA, and the XTA mod is another variation of the original TA game which is distributed in the default software package users get when downloading TA Spring. There are also mods inspired by other games and media, like Gundam Universe inspired by the Japanese animated movie series Gundam, and Star Wars TA inspired by the Star Wars movies. Furthermore there are mods that have themes completely original to TA Spring like Expand & Exterminate, and Xect vs. Mynn. There are many other mods also.

Two mods may be radically different in terms of which strategies are possible. The AI presented

in this thesis supports most major mods for TA Spring and is designed to support future mods without requiring an update. This is intended to benefit those contributing to the TA Spring community by developing and testing new mods, and it is also in an effort to make parts of the AI general and possibly reusable for other game engines.

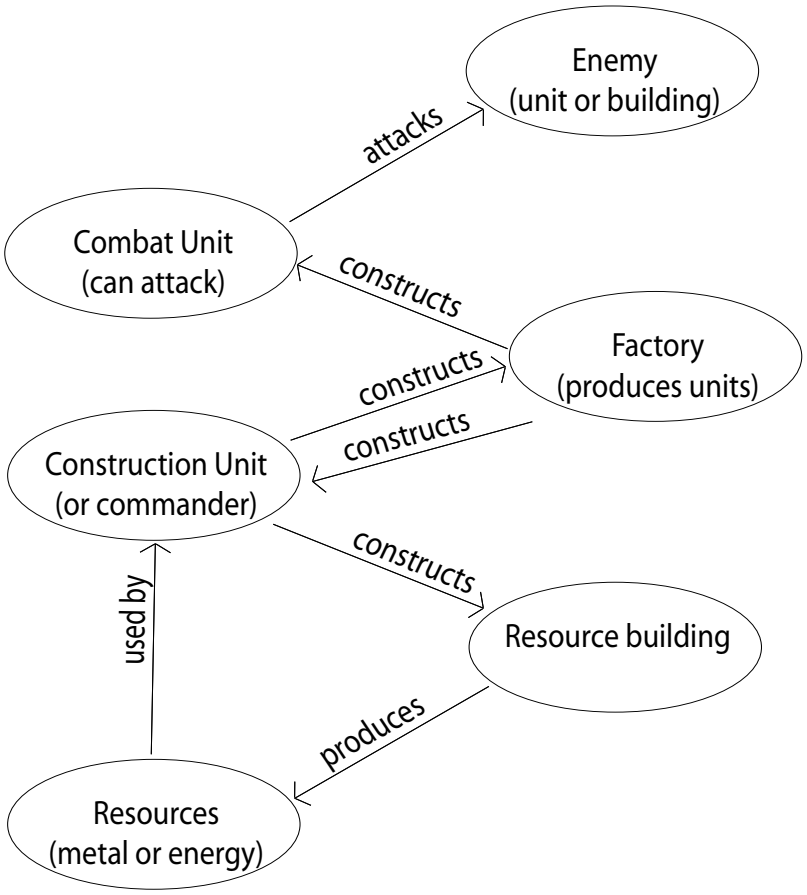


Figure 6. Ontology for construction relationships in TA Spring.

5.6 Construction Relationships

Figure 6 shows the ontology for construction relationships in TA Spring. Construction buildings (factories) can construct units, and construction units can construct buildings and sometimes units. Construction means the capability to produce additional game entities, and in TA Spring all construction expends resources. Resources are in turn produced by a type of buildings, and there is a low starting amount of resources to get the process of construction started. If a player has a construction unit, it is usually possible to build all units and buildings from that point, by constructing factories which in turn can make new and different construction units.

5.7 The Goal of Playing the Game

A goal is the underlying reason for performing any action. The main goal that players have when playing TA Spring is to win, just like in any other game. Winning is ultimately done by destroying the enemy. In order to achieve victory, a player has spend resources on military related construction, so that he is able to attack. More resource income enables more construction, so while the player has the goal of increasing his military force, he should simultaneously increase resource income, which is a sub-goal. The metal resource in TA Spring is normally spread out over the map,

and being able to extract the metal requires presence and control over the immediate area around that metal spot. Because of this, achieving control of more regions is another sub-goal.

5.8 Example: A Game of TA Spring

At the start of a game the player typically has only the Commander unit and a small amount of resources. The player then gives the orders for the commander to produce a handful of resource producing buildings and a factory. This factory can produce additional units, which can be both combat and construction units. The resource buildings are different from factories in the sense that they produce the resources that are required by the factory to construct additional units. Resources are also required for construction units to produce buildings. The game progresses by increasing the amount of resource buildings, construction units and combat units. The combat units are used for attacking the enemy or defending against enemy attacks. Perhaps some armed defensive structures that can be built, but these can not be moved after being built so they are for defensive purposes.

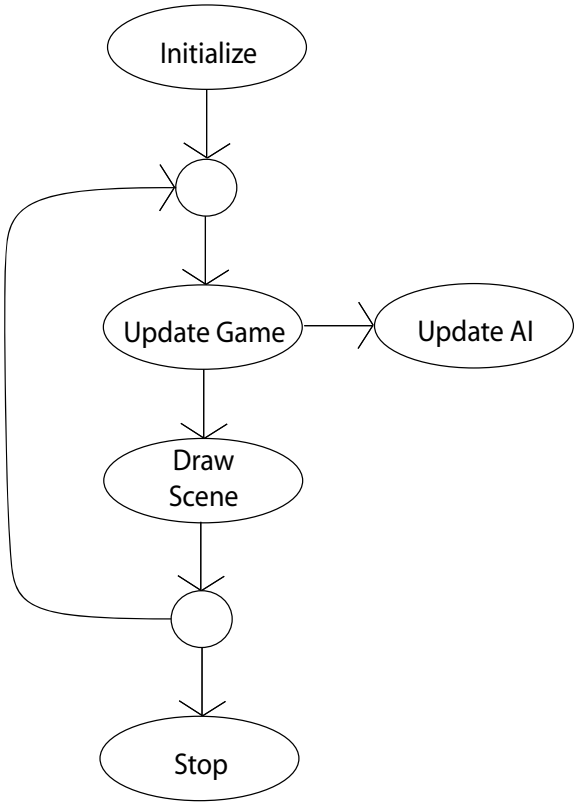


Figure 7: The main events in the main game loop.

5.9 The Main Game Loop

Games are programs that are constantly running and updating and TA Spring is no exception. The part of the code encompassing doing the frequent updating of the game state is called the main game loop. By "game state" is meant all the data that pertains to the world one point in time, including all resources, buildings, and all the properties of the units such as health, position, speed, heading, current orders, and even debris from explosions. The time between two updates is called a frame[7], and in TA Spring there are 30 such frames per second. This means that the game state is updated 30 times per second. The main game loop executes essentially two main tasks: Updating the game logic, and rendering and drawing the graphics[23]. All the information relating to

the shape of the world and resources and units is called the game state. In short, the main game loop first executes all code affecting the game state, and this information is used when drawing the graphics. Figure 7 shows the context where the main game loop runs. Please note however, that the graphics can update more than once per game frame and there are algorithms in place making unit movements look smooth in these additional updates of the graphics. Updating the game includes taking into account the actions of human players and AIs. For the player this means processing user interface input, and for the AI this means running the AIs main update function. After receiving these actions the engine executes the unit scripts, and then all the physics simulations and unit position updates. Unit scripts are automated behavior of the units taking into account the definitions from the mod, such as how they turn, animate, move, et cetera. Physics simulations include weapons damage to units, deformations to terrain, explosion fragments, ballistics calculation for projectiles, and every other update that affects the virtual world. The engine calls the update function of each AI every frame, i.e. 30 times per second.

5.10 Updating the AI

As previously mentioned, the main game loop calls the main update function of the AI. All the AI code execution is done from this AI update function, with minor exceptions of event driven messages that are sent from other parts of the main game loop. These event driven messages are messages of events that happened that specific frame. Such messages include units that died this frame, messages of damage taken to units, units that have newly been constructed, units that are stuck in the terrain, and so on. The update of the AI is 30 times per second just like the engine, but time consuming operations that need to be done with regular intervals can be done less frequently, such as every 15, 30 or 60 frames, or on demand. TA Spring offers two separate interfaces that developers may connect an AI through. The one used in the work in this thesis is the “GlobalAI“ interface, which consists of previously mentioned events, the update function, and a set of callback functions for giving unit orders and retrieving all information about the game state and settings. All communication from and to the AI goes through this interface. There is also a “GroupAI” interface, which is for a different purpose. A group AI in TA Spring is a program that a human player may utilize and apply to a handful of units or buildings to automate some of their behavior, but if the contribution that the group AI makes gives an unfair advantage then it is considered cheating. Programs communicating through either the global AI interface or group AI interface are compiled as separate dll files and loaded into the game.

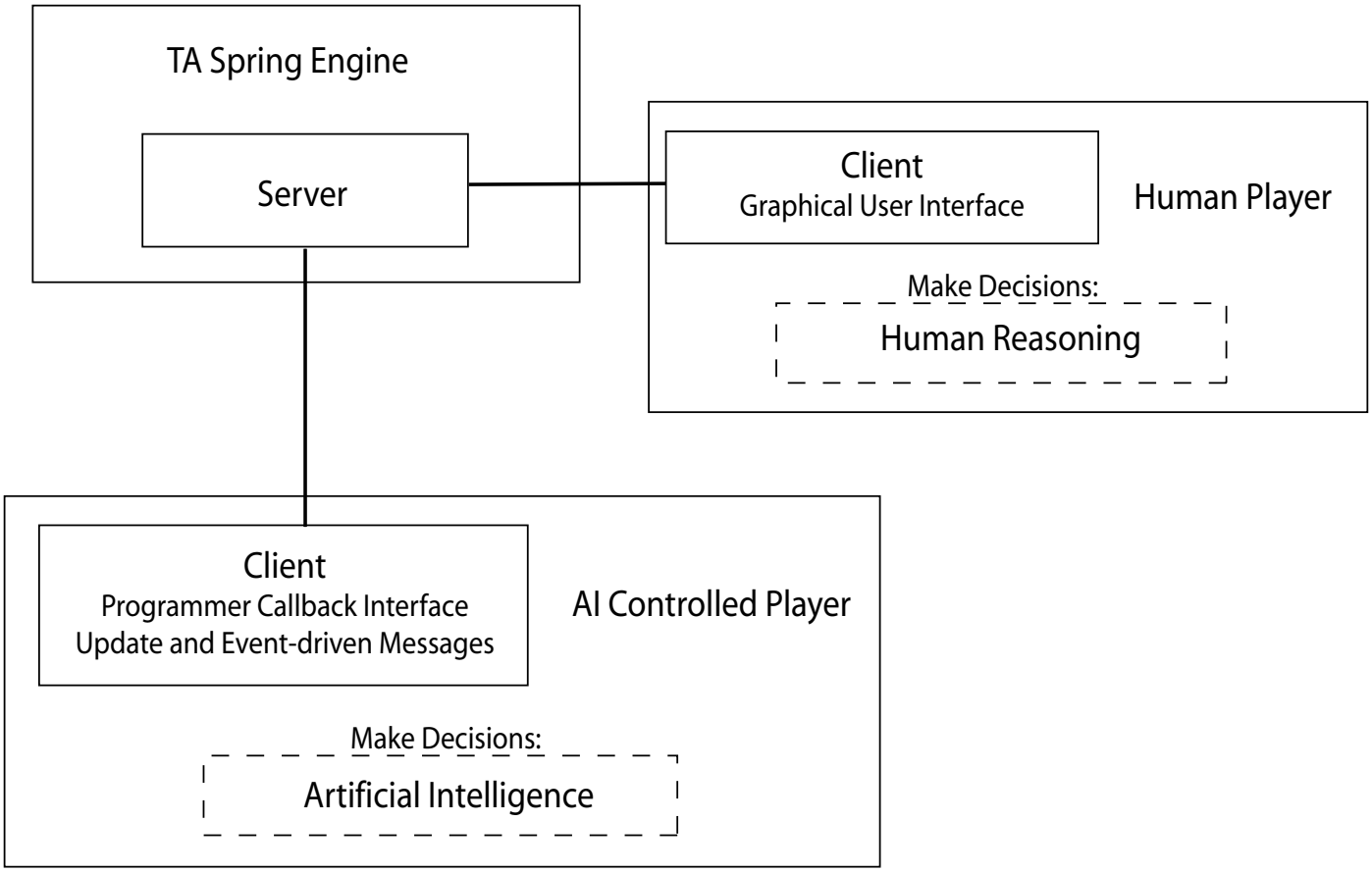


Figure 8: The human player and the AI controlled player playing the TA Spring game.

5.11 The Context and Purpose of the TA Spring AI

The interface that the AI controlled player plays the TA Spring game through is different from the one that a human player plays though, and a comparison is shown in figure 8. Still, the AI takes the role of a player from start to end, and it has goals and possible actions that are the same as a human player has. The left part of Figure 9 shows a game started for two players (AI or human) where neither player takes any action. The unit in the top left corner is the Commander unit for player 1, and the unit in the bottom right corner is the Commander unit for player 2. As explained in the ontology for construction relationships in TA Spring, the Commander unit can build factories which build additional units and so on. If no action is taken by any player, time will pass but nothing at all will happen.

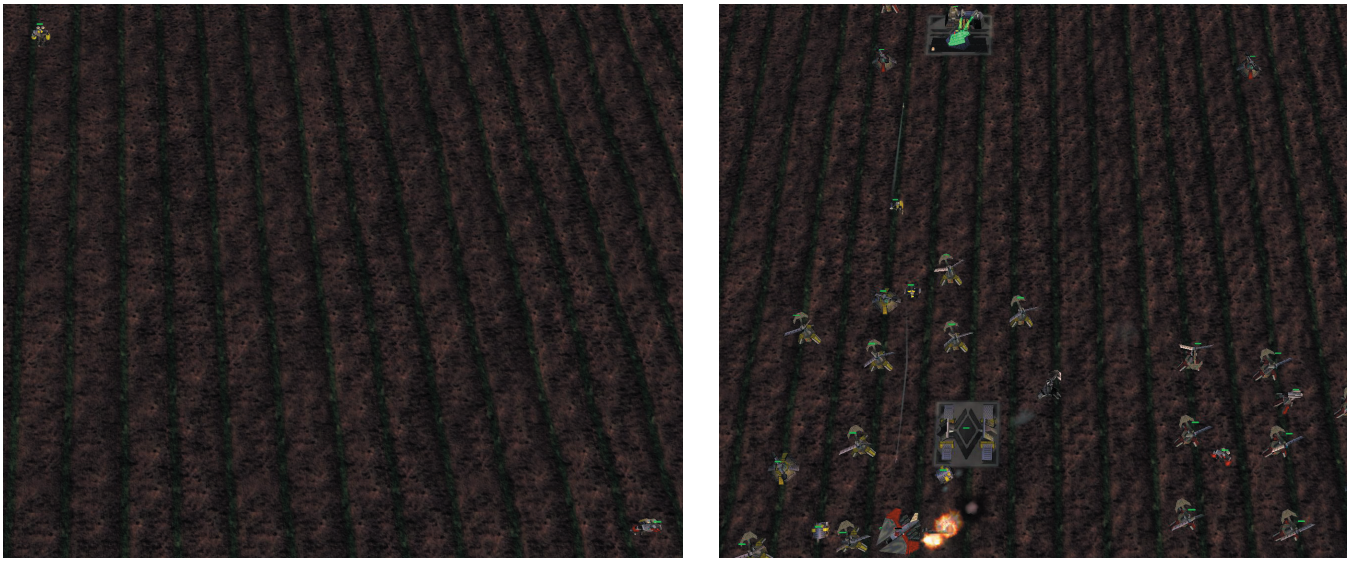


Figure 9. Two screenshots showing the difference between when players take no action (left) and where both players take action to destroy the other (right).

A human player is easily able to beat an AI which takes no action. This is the context and purpose of the TA Spring AI; providing a human player with resistance and entertainment (games are about fun, after all). In order to fulfill the role of a player of the game, the artificial intelligence controlled player has to take action. Furthermore, it is desirable to have an AI that plays intelligently by using a strategy that can enable it to win against a human player, and this strategy should be put into action in a way that takes the environment properties and changing situation due to enemy actions into account.

5.12 Human Players

As shown in figure 8, there are differences between the interface that human and AI controlled players communicate through. The human player gets information through a 3D display of the virtual world, with a 2D overlay user interface with additional written information and illustrations. Figure 10 shows a part of the TA Spring user interface, where the human player is using a number of construction units to build a few buildings. All actions that a human player takes are goes through the same interface. Human players learn how to play the game with experience, and are able to make intuitive guesses about what the enemy is doing based on what they have done previously. Human players are also able to view the game through the interface and use their cognitive abilities to analyze and predict such things as speed of units, approximate range of attack, logical routes to travel, and other aspects of the game that might influence their decisions.

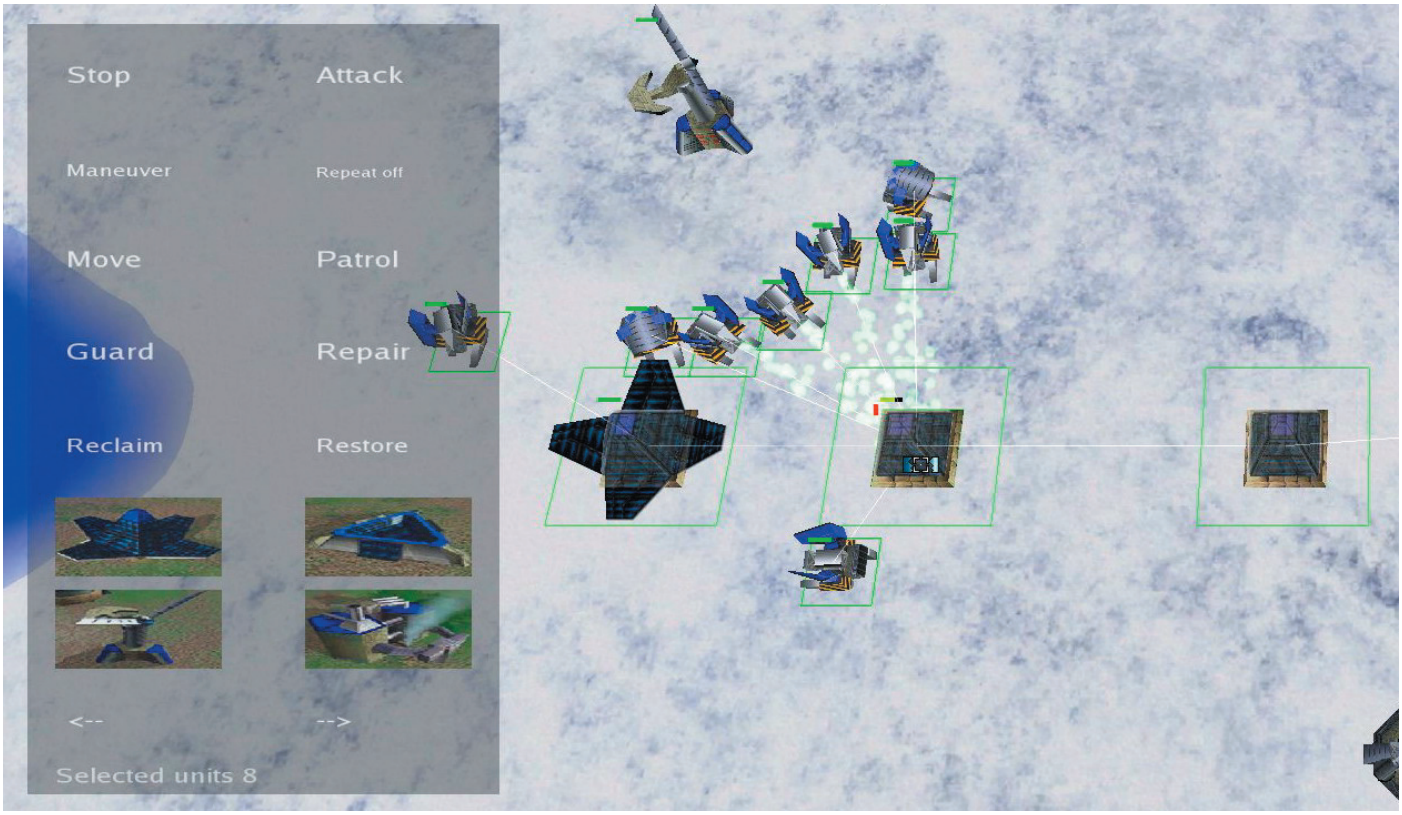


Figure 10. A human player playing the game through the graphical user interface, ordering construction units to construct some buildings.

5.13 Differences Between Human Players and AI Controlled Players

Human players playing against the same AI opponent repeatedly will be able to learn and gain some understanding of how the AI is able to play in terms of strategy, and also how it plays in detail. Human players tend to specialize in particular ways of playing, and so the AI is likely to play similar from game to game against this player. Therefore, if there is consistency to the behavior of the AI, a player would certainly be able to figure out any weaknesses of that behavior and adapt to best defeat it, in the long run. The AI has to generally be robust in the ways of being able to beat different strategies, because if there are strategies it is unable to beat then it will certainly lose most games in the long run. The AI has to have a way of countering all imaginable strategies that the player can choose, because a player will experiment and try to find strategies that the AI can not counter, if there are any.

A human player is able to scout in the game, by sending units to places he suspects there might be enemies and possibly discovering their location and what they are doing. He also knows the settings for the game which determined the possible starting locations, and so he can scout those locations to be reasonably sure to find enemy units. It is also common to be able to guess which route the enemy is most likely to attack via. As a human players plays the game, he gains experience about what strategies are common under given game settings, and so he will effort to make safeguards against those strategies. It is possible for an AI to achieve some of these qualities through learning, which requires a training period for the AI to gather and analyze data. The AI will however have to relearn strategies if the player chooses to play a different mod, since strategies can vary from one mod to another.

Another major difference between AI controlled players and human players are the way in which they reason about units in combat, their ranges, their positions and the places they can travel. They both may be able to assess the terrain, the human by looking at the 3D representation, and the AI by retrieving data about the terrain through the AI callback interface. The positions of units, buildings and resources are also available to both in the same manner as the terrain. It is definitely a challenge to take all these factors into account in an AI algorithm in a way that will result in optimal performance, both in planning attacks and in micromanagement during combat.

AI controlled players have some benefits over human players. Computational power for numbers is an obvious advantage, especially if the AI CP has a means for calculating which units and buildings are the most efficient at any given time. Another important AI CP advantage is the ability to easily repeat the same procedure to large amounts of instances of the same problem in very short time. Once an AI has the ability to solve a particular problem in an efficient way, it will have no difficulties doing this often or several places at the same time. Since RTS games have both economic aspects and military control aspects, human players have to prioritize actions. While a human player focuses on micromanaging units in combat, he is likely to be utilizing his resources less efficiently. The AI does not have this problem as long as none of the algorithms take all available CPU time. Therefore, once a good and efficient solution to a given problem area exists for the AI, it will be able to solve all instances belonging in that problem area with more ease than a human player, since the person would have to focus on one instance at a time. The AI CP also obviously has a benefit if there are problems that can be solved with straight forward computing of numbers. However, it should not be forgotten that the point of games are entertainment, and if the goal is to entertain the human player then overusing advantages such as computational power to make decisions that are hard for humans might result in an AI CP which is not entertaining.

5.14 Situational Awareness in TA Spring

Situational awareness is definitely an aspect of playing the TA Spring game, in several ways. The best example would be the actions of other players of the game, which have serious effects on how a player should utilize his forces. The strategies in RTS chapter already describes how actions of one player affect the merits of certain actions on the behalf of other players. There is also the aspect of resource storage and income, which are constantly fluctuating and have to be taken into account when the player makes economic decisions.

6 AI Architecture

6.1 Architectural Framework

This chapter presents an architectural framework of a real-time strategy game, which has been used to implement the "KAI" AI computer opponent for the TA Spring real-time strategy game. First the architecture is given in figure 11, and then the explanation of the modules and components and the interactions between them within the architecture are explained. This chapter will make heavy use of terminology, concepts and ideas explained in the previous chapters of this thesis. The strategy aspect of RTS AI is central to this architecture and Figure 11 shows a diagram of the AI architecture.

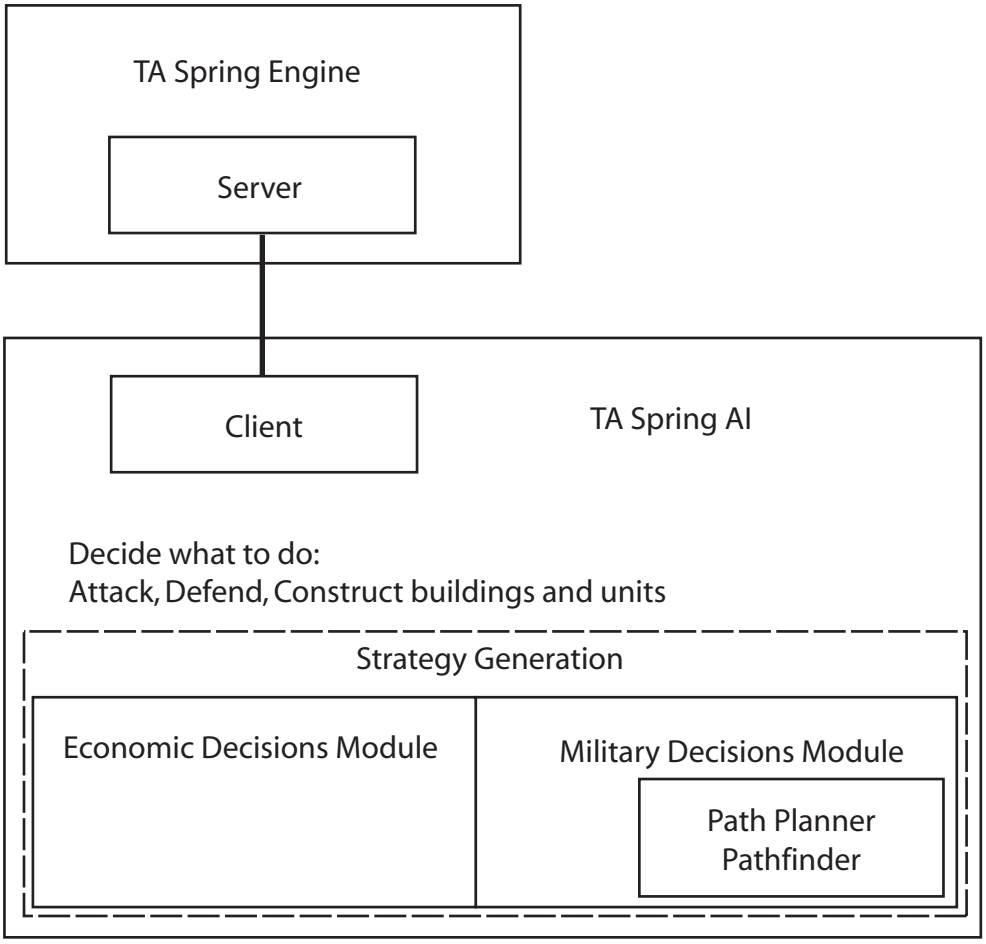


Figure 11. The AI architecture, showing strategy generation by the two main modules.

Strategy generation has two aspects: the economic aspect and the military aspect. The two modules shown in Figure 11 handle these two aspects respectively.

6.2 The Economic Decisions Module

The economic decisions module makes all decisions related to the economy of the game. Economical decisions are decisions both for expanding your economical situation and for constructing buildings and units. It is centered around selecting what to construct and the details of the task of construction, such as assigning construction orders to specific construction units. This includes construction of resource buildings and other infrastructure buildings, defense buildings, construction units, and combat units. The question of selecting which combat units to construct is particularly important, since it determines which options for attack the military decisions module has. Selecting units to be constructed is done by calculating the strengths of the different units that can be built using mathematical calculation approximating how efficient the units will be in combat. Constructing the combat units is the end goal of the economic decisions module, but this requires resources and that is the reason why expanding the economy is a goal.

Once a combat unit has been constructed, control of it is handed over to the military decisions module. Economic decisions also include the production of defense buildings. Since defense buildings do not require further control after they are built, control of them is not given to the military decisions module.

6.3 The Military Decisions Module

Military decisions are about utilizing military assets. After a combat unit is constructed, control of it is given to the military decisions module, which then makes all decisions related to the combat unit for the rest of its lifetime. Military decisions are made towards the goal of destroying as many as possible of the enemy's units and buildings, without losing buildings or too many units in doing so.

The first task is organizing units into groups, because a controlled group of units is more likely to be able to destroy a target than unorganized units. The amount of groups is determined by the existence of easy enemy targets, hence if all targets are heavily defended then more units will be added into an existing group. Human players also tend to organize their units into groups when attacking, so this in combination with the details of micromanagement of units are vital parts of achieving the goal of making a humanlike play style visible to players and spectator.

Having units in groups creates stronger entities than individual units that the military control module can utilize. This makes it possible to include information about the groups strength when planning paths for attacks. The military decisions module attempts to find targets on the enemy side which are poorly defended, and to attack these with a sufficiently powerful group of attack units.

Path planning is the art of deciding which route to take[6]. KAIs military decisions module utilizes a combination of unit strength assessment and path planning for locating targets and assigning attack groups to attacking the area that the targets are in. Replanning happens in cases where

the enemy situation changes while a group of combat units is on the way to their target so that enemies get in the way. In this case, a new decision on where to utilize the attack group is made. The details of the algorithms involved are included in the AI Algorithms chapter.

Micromanagement of units is also the responsibility of the military decisions module. Micromanagement is done by a set of rules defining the behavior of units while they are fighting, such as placing them in a formation or moving them closer or further away from enemies, or selecting which enemy in a group to kill first. Micromanagement of units is implemented in the AI CP in order to use combat units more efficiently and to display more humanlike behavior. Details about the algorithms governing micromanagement is explained in the AI Algorithms chapter.

In literature on RTS AIs, generating or selecting strategies are often mentioned, typically from set of possible choices. In order to use learning efficiently, the number of choices (the size of the search space) should be reduced[27]. However, playing optimally in the environment of TA Spring requires a staggering amount and frequency of decisions. This makes it a very hard task to create a strategy in advance that would suffice more than a few minutes into the game. The AI presented in this thesis has a dynamic approach to strategy "generation", and emergence and situational awareness are important concepts in understanding this approach.

6.4 Emergent Strategy

The strategy that the AI is using emerges from the actions of the parts. The actions done by the AI CP are determined by higher and lower levels of decision making components, in a structure similar to a network of agents. Higher level agents do not command lower level agents, but they activate or trigger lower level agents[24]. Higher level agents consider available information, retrieved by lower level agents, to make high level decisions. There is however no main decision about the totality of the strategy to be selected. Instead, the strategy emerges through the interaction of agents solving their part of the problem, by using the common infrastructure for analyzing the environmental situation and the actions of the enemy (situational awareness). The architecture illustrating the relationship between different agents is shown in figure 12.

While the economic decisions regarding increasing income and increasing build power are determined largely by a script of rules, decisions in combat unit production are made on basis of the situation and the actions of the enemy. The AI tries to build those units which best are able to counter the offensive units and defensive structures that the enemy constructs. The same is true for defensive buildings, they are selected in an effort to best counter the mobile forces that the enemy builds. The high level strategy of the AI emerges through these mentioned actions. Following is an example illustrating this point:
A human player is playing TA Spring against our AI. The game is done in space-based setting where all combat units are spacecraft of various types. The player decides to construct a great number of small and cheap attack units in an effort to overwhelm the AI. The AI will then effort to produce the units which are best suited for destroying the human players small units, which in this case are medium-sized units with weapons designed to be efficient against small units. The military decisions module will then utilize these units to cost-effectively destroy the enemy small units, which contributes to winning the game.
So in this fashion the game would be won by using a strategy with medium sized units to destroy

the enemy, made manifest by constructing the appropriate "medium" factory and producing the unit most suitable to counter the enemy's forces, and using them to easily destroy the small ships. The relationship between strengths of units in RTS games can be simplified and compared to that of the children's game "rock-paper-scissors", where each selection has strengths and weaknesses against other selections[27]. In TA Spring, different units with the similar cost of production might be more efficient depending on what units it is going to face in combat, and selecting which combat units to construct is part of the strategy. Furthermore, the actions of the enemy player in terms of combat unit movements and defense placements greatly influence how the AI moves its combat units. The military module will effort to organize units and send them off in such a fashion that they follow a path with low risk up to a position with enemy units or buildings that can be destroyed without heavy losses for the AI. In this way, the actions of the player are taken into account on the more immediate level by the AI when it moves combat units. Controlling units in this manner requires frequent checks and updates in unit orders. Circumstances are likely to change in the course of a few seconds, therefore it is not optimal to determine all movement in advance. Instead the actions of the units are adjusted dynamically during the fight depending on the actions of the enemy.

6.5 Situational Awareness

The point that the TA Spring game requires situational awareness to be played efficiently is an important point because it has major implications for how the AI CP should work, particularly in terms of executing a strategy, and it also points to the necessity of having dynamic real-time algorithms managing several aspects of the game. For a given problem area, the developer must find an appropriate solution. If symbolic information can be found in the problem area to the extent that a top-down decision is optimal, then this is likely the easiest and best approach. On the other hand if the solution is dependent on perception in the immediate situation then perhaps a "bottom-up" decision is best made by local perceptions, internal states, and communication, in a manner similar to in an agent based architecture. An agent may perceive the virtual environment, analyze the information, and commit an action, which may require storing a plan for a sequence of actions to happen over time. [26] explains how the main advantage of such a "bottom-up" approach is that it enables some autonomous adaption through generating actions that can be considered to be emergent behaviors.

The KAI architecture uses concepts of situational awareness. The most important changing factor that is being taken into account, is the actions of the enemy related to military assets. What kind of combat units and defensive structures the enemy constructs is used to determine which units and defensive structures KAI builds. This is done by using calculation to simulate which units are most cost-efficient for destroying the forces that the enemy currently has.

Many factors affect the availability of resources to the AI CP, such as variations in wind, resource buildings being destroyed, and resources being used by actions that the AI has taken. The availability of resources and resource cost of units and buildings is also taken into account when deciding which buildings and units to construct. Lower availability of resources will result in the construction of cheaper units. Furthermore, what factories and construction units that the AI CP controls determine the speed at which it can construct, which is also a factor in the decision.

There is also the differences between different mods and maps that can be played. A different map might have a very low amount of resources, and the terrain might also make travel with one type of units better than another unit, and all these things are taken into account when selecting what to build. KAI adjusts resource expenditures by the amount of available resources at any given time. Having more resources available results in KAI allowing for the construction of more expensive buildings and heavier units when enough resources for it are available.

Path planning and micromanagement are also problem areas where the AI CP must take into account the specifics of the situation. These decisions are influenced by many factors that change, the most important being the movements of the enemy troops and the placement of the enemy's defenses. Pathfinding using threat maps solves the problem of taking enemy movements into account in detail, in an efficient way. The details of the threat map is explained in the AI Algorithms chapter.

6.6 Higher and Lower Level Decisions

As explained in chapter 6.3, the actions done by the AI CP are determined by higher and lower levels of decision making components, in a structure similar to a network of agents. Playing the game with an emergent general strategy is at the highest level, and below that level are the two modules implementing economic control and military control. These two modules each has several goals guiding their decisions. Their knowledge of the environment is processed by lower level agents, and their decisions are executed by lower level agents. A more detailed explanation is presented in figure 12.

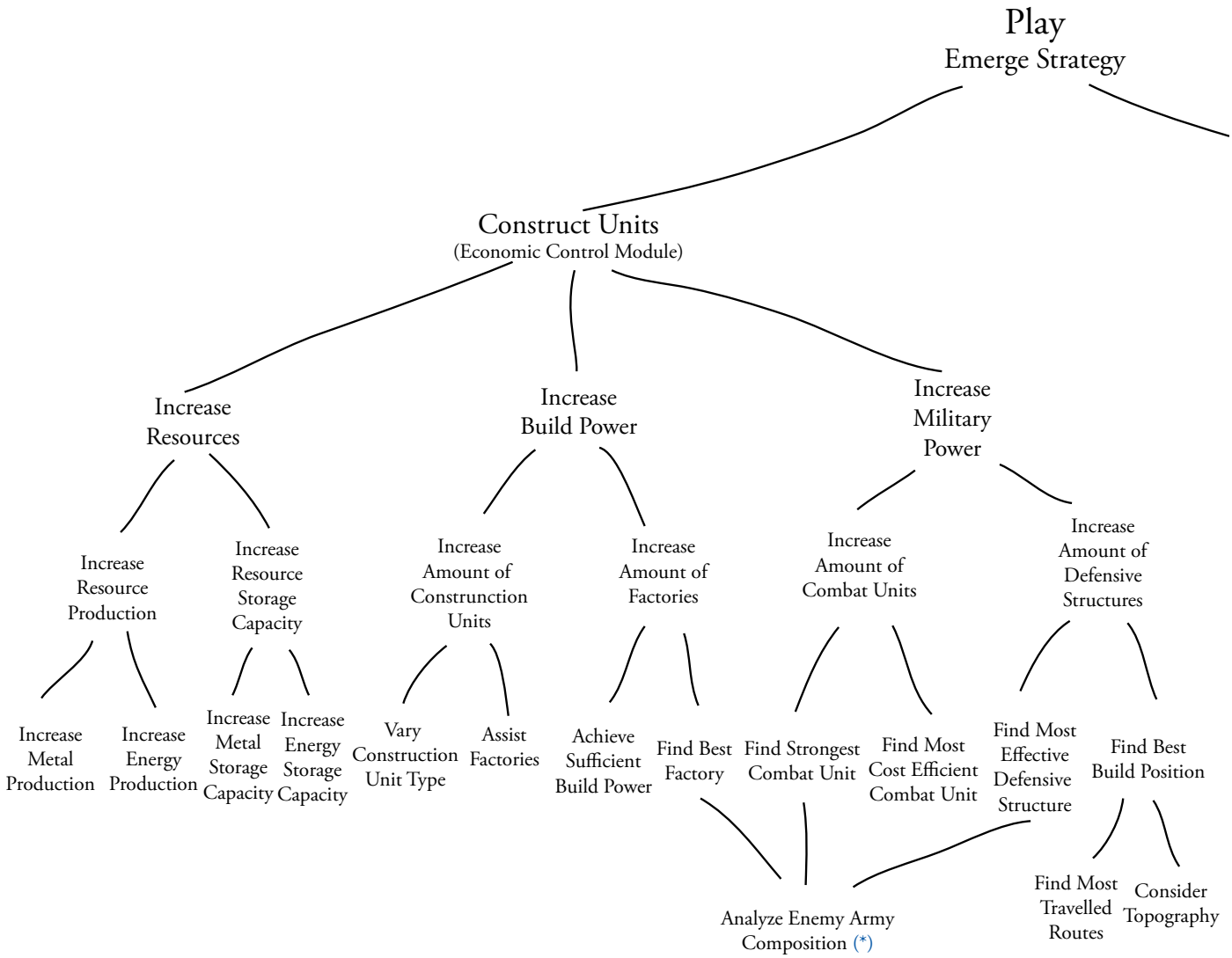
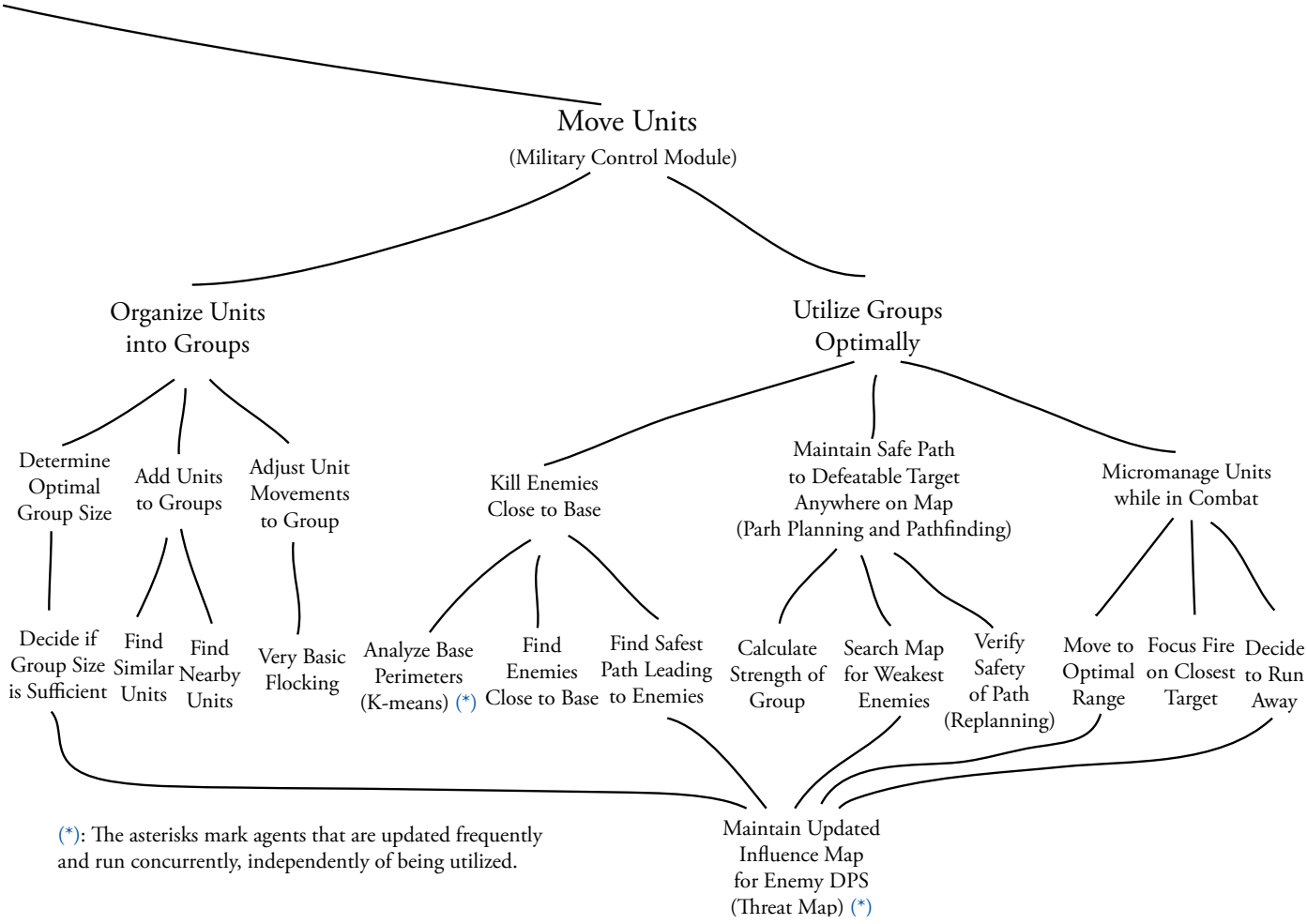


Figure 12: A two-page diagram modeling KAI as a hierarchy of agents.


Move Units
(Military Control Module)
Organize Units into Groups
Utilize Groups Optimally
Determine Optimal Group Size
Add Units to Groups
Adjust Unit Movements to Group
Kill Enemies Close to Base
Maintain Safe Path to Defeatable Target Anywhere on Map (Parh Planning and Pathfinding)
Micromanage Units while in Combat
Decide if Group Size is Sufficient
Find Similar Units
Find Nearby Units
Very Basic Flocking
Analyze Base Perimeters (K-means) (*)
Find Enemies Close to Base
Find Safest Path Leading to Enemies
Calculate Strength of Group
Search Map for Weakest Enemies
Verify Safety of Path (Replanning)
Move to Optimal Range
Focus Fire on Closest Target
Decide to Run Away
(*): The asterisks mark agents that are updated frequently and run concurrently, independently of being utilized.
Maintain Updated Influence Map for Enemy DPS (Threat Map) (*)

An AI CP for an RTS game needs to handle the most basic aspects of the game in order to play at all, in terms of retrieving information about its units and sending unit orders to the engine. These basics point to the need for the developer to separate parts of playing the game into problem areas, and creating modules and components with different responsibilities, and then connecting these modules and components in an optimal manner. Figure 12 models these relationships in KAI as a network of agents in a hierarchy. The highest level goals are solved by the higher level agents, the highest being the agent playing the game with an emergent strategy. The figure is not complete down to the code level, but it includes most important decisions and makes the actions of all important parts of the architecture visible.

6.7 Substrategies

The substrategies that compose the high level strategy are selected dynamically based on these important factors:

1. The amount of resources in storage and resource income are important. The amount of available resources is a factor in determining which units the AI attempts to build at any given time. More available resources means the AI CP can construct more expensive structures and units.
2. Our construction capabilities are important. The AIs speed of construction affects the decision about which units to build, little construction power means smaller units and buildings are built. Having lots of resources and low construction power means that the AI CP should attempt to increase its construction capabilities. Increasing construction capabilities (also known as build power) in TA Spring is done through producing additional construction units and constructing more factories.
3. Unit vs. unit efficiency is established based on the attacking units range, weapon, firing rate, projectile speed, firing accuracy, gravity, target unit health, target unit armor, and other special abilities. This calculation is used to select which units are effective vs. other units, and this is used in order to create the units which best are able to destroy the enemys current army.
4. The composition of the army that the enemy has is important. Each potential unit that can be built is tested against the composition of the enemy army and scored accordingly.

Some of the substrategies are the responsibility of the military decisions module. These substrategies include the following decisions:

1. Which units to group together in an attack or a defensive measure.
2. Which enemy units or buildings to attack with a given group of units, and how to get there.
3. How to micromanage units while fighting to best be able to destroy the units that the enemy has.

Tactics in the context of TA Spring is the art of utilizing combat units in a small battle, and in the AI CP this is implemented mainly in the micromanagement algorithm, which is explained in the AI Algorithms chapter.

6.8 The AI "Loop"

As explained in the TA Spring chapter, execution of the update of the TA Spring AI is done 30 times per second from the TA Spring engines main game loop.

The sequence of events in the AI main update function is:
- Adjust and verify amounts of resources in storage and predict future income and expenses, for use when deciding how much resources to spend on future construction.

- Calculate how much combat power we have and how much combat power the enemy has, in order to decide how to prioritize construction.

- Calculate how much need there is for defense, which is influenced by the above calculation.

- Iterate all factories and see if any of them are idle and we can start the construction of a new unit.

- Iterate list of idle construction units and give appropriate orders for construction, assisting in construction that another construction units is doing, or reclaiming wreckages on the map in order to receive bonus metal resources.

- Check for newly constructed combat units, and put any new combat units under the control of an attack group.

- Check if it is necessary to reorganize the attack groups, by adding or removing units.

- Verify that orders given to the attack groups are appropriate and still valid, or give new attack orders. Cancel an attack and give defense orders if enemies are close to base.

- Update each of the groups and verify/update the orders given to each combat unit.

The low amount of time available for a single update puts a constraint on the types of reasoning which may be used in real-time games, and TA Spring is no exception. Details of various algorithms are explained in the AI Algorithms chapter.

7 AI Algorithms

This section gives a selection of the algorithms, and relates these algorithms to the AI Architecture. All the algorithms explained in this AI Algorithms chapter were implemented in KAI to improve behavior in specific problems areas. When reading about the algorithms, keep in mind the time constraints to the calculation. An AI module of the TA Spring game is updated every game frame, i.e. the CGlobalAI:Update() function is called 30 times per second, and frequently using more than a fraction of a second to run an algorithm is not acceptable in a running game AI.

7.1 K-Means

K-Means is a partitional clustering algorithm is from the Information Retrieval field, usually used for data mining[5]. In our TA Spring AI it has been simplified and used to create an abstraction (higher level knowledge) of where the bases are. A partitional clustering algorithm is an algorithm for taking large amounts of data in the form of vectors and organizing them into clusters. Which cluster each vector ends up in is determined by the data, similar data will end up in the same cluster. Each cluster will have a cluster representative, which is the vector which is the average of all vectors in the cluster. Each of the cluster representatives are then used as abstractions representing different parts of the base. In our case the vectors to be organized into clusters are three dimensional (three-space) and the measurement of similarity is their distance measured in two dimensions. Two dimensional distance is sufficient since buildings are placed on the surface of a two dimensional plane, it is impossible to build in the air or below ground. K-means is an iterative algorithm and has been distributed over time so that a single iteration is only done every 150 frames (every 5 seconds). This is sufficient for our limited data set, which are just three dimensional vectors [x,y,z] representing the positions of buildings in our game. There are two separate data sets, one is the list of positions of the buildings that the AI controls, and the other is the list of positions of buildings that are controlled by players hostile to the AI. The number of cluster "centroids" (the average vector representative of a cluster) increases with the number of buildings, by the formula k = min(1 + sqrt(n), 32) where n is the number of buildings and k is the number of clusters. How to determine k is not included in the K-means algorithm definition. Figure 13 shows how the number of clusters increase as the number of buildings increase:

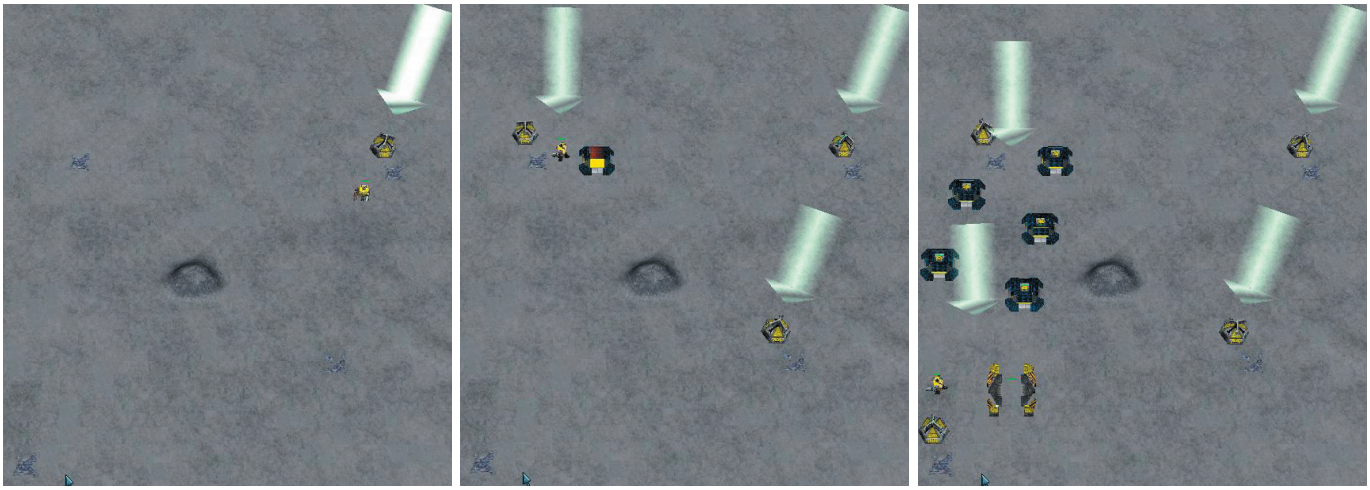


Figure 13. The light green arrows represent cluster centroids generated by using the K-means algorithm on a data set consisting of friendly building positions.

The information generated by the k-means algorithm is used by the AI for calculating approximate base perimeters, and the AI also uses the technique to calculate enemy base perimeters. Figure 99 shows where the means are at 3 different points in time, illustrating how the number of means and their positions change over time, starting with just one (left). The list of "means" serve as an abstraction or simplification of which parts of the map are under control of which player. The friendly base perimeters are then analyzed with regards to distance to enemy base perimeters, and sorted by how safe the spots are. When this has been done, the AI knows which "means" are the safest and which are the most risky, and the risky ones are the ones at the perifery of the base. It becomes more useful as the number of buildings increase, later in the game, since the amount of resulting "mean" position always stays a low number. This is part of the AI infrastructure and has various uses, some of which utilize the fact that it is sorted while others do not.

There are uses for this list of centroid vectors representing the base area, most of which are path planning related:
1) The vectors can serve as starting points when determining if the enemy is attacking the AI's base or if any enemy units are close. Normally, the safest position is used as the origin for the search, so that the enemy which has advanced the furthest into the base will be found first. For example, if the AI base is in a corner, the safest position is likely to be near the edge of that corner. Then, whichever enemy is closest to that corner is the enemy most likely to be the highest threat to the base, and this enemy will be found first since the distance from the corner to his position is the shortest.
2) When groups of combat units are retreating to base, the cluster representative vectors are used as goal nodes for the path planner search creating the retreat path. In this case, the sorting of the vectors by their safety is not used, instead their proximity to the retreating units is more important.
3) A subset of the least safe positions can be used as a patrol path for defending units. Since they are the least safe parts of the base, they are the places where the enemy is most likely to attack.
4) When running the AI in graphical debug mode, the cluster representative vectors are shown on the map. This serves as an illustration of data that the AI is basing some of its choices on.
It is also worth mentioning that this iterative algorithm is iterated only once every few seconds, and is updated regardless of how often the knowledge it generates is accessed. It is not technically a separate process, but it does run independently, taking the changing environment into account and updating its model as time passes. The "environment" in this case is the building positions, and they change when a new one is built and when they are destroyed.

7.2 Influence Maps and the Threat Map

Influence mapping is a tool used for terrain analysis. It is essentially a 2D array representing the playing field, or parts of the playing field. In general, values in the 2D array are increased or decreased based on various influences[3]. In the case of our AI, the influences are enemy military might, in terms of combat units and defensive structures. Specifically, in the TA spring AI, the value at any given coordinate is the damage that the enemy immediately is able to inflict there, updated two times per second. This is done by the same calculation method as the one used for calculating unit strengths, i.e. in terms of DPS (damage per second). We refer to this specific use of influence mapping as "threat mapping", and "threat maps".

Updating the threat map is done though iterating the list of known enemies, and for each enemy it adds its DPS to the value of each field in the influence map which is in range of the unit. An example situation and differences in threat values is shown in figure 14.

Figure 14: An illustration of an excerpt from a threat map with various threat values determined by nearby enemy units. The damage that the units are able to deal decides the level of threat, and their maximum attack range determines which fields the damage value is added to. Note the "High Threat" areas within reach of both units.

This threat map has several uses in the TA Spring AI. One of those uses is as a basis for path costs in the pathfinder. In this way, a combination of distance and threat to safety determines which

paths are chosen. By using this method it is possible to avoid unnecessary conflict while travelling towards a destination. A typical example would be combat units bypassing enemy defenses in order to attack their infrastructure buildings, by moving outside the range of the defensive structures whenever possible. It can also be used to assess the chances of the success of an attack, by comparing the DPS at a position with the DPS of a AI CP controlled group of units.

Updating a threat map accurately and often can be a time consuming operation. In order for micromanagement procedures to be able to get accurate data in the middle of a fight, data that is one second old can make the difference between life and death for combat units. The complexity of the update is high because for each enemy unit that can deal damage, a large set of values in the threat map has to be updated. Increasing the resolution of the threat map means that the amount of sectors within range of a given unit increases a lot, while too low resolution results in inaccurate data which can lead to bad decisions about where to move. However, within normal parameters for the size of the map and the number of units and their range, the threat map can be updated within a frame without resulting in choppy game performance.

7.3 Path Planning and Replanning

Path-Planning and replanning is part of the Military Decisions Module in the AI architecture. [6] proposes the following definitions: “Path planning is the art of deciding which route to take, based on and expressed in terms of the current internal representation of the terrain. Path finding is the execution of this theoretical route, by translating the plan from the internal representation in terms of physical movement in the environment.” Pathfinding is used to determine the detailed path which the attack group is to follow on their way to their target. Path planning is used to determine which target to attack, which is an unusual but effective way of solving the problem of allocation of military resources. The strength of the group is calculated by estimating their combined DPS. The DPS of each enemy unit is added into a threat map at positions within their firing range, and this threat map is used as the cost for the pathfinder search when travelling between nodes in the search space. Pathfinder searching normally includes having a single point of origin and a single destination point. The pathfinder algorithm in KAI has expanded this method to using multiple goal nodes. When the path planner tries to find a target and a path to the target, it compiles a list of known enemy positions that can be destroyed by the attack group and adds them all as goal nodes for the pathfinder search. The search space is then a number of nodes representing positions on the map, where travel costs are the threat of enemy fire at each point along the path, and the end nodes are potential enemy targets. The search will then find the path which has the lowest cost, and this path will be the path to the target which offers the least resistance.

The enemy targets are filtered by their value as targets, and when a path has been assigned to a group of units, that group reserves the area in the immediate vicinity of the target which the path leads to. The targets within this area are then not accessible as targets to the other attack groups that the path planner assigns targets to as long as one of the groups has them already assigned. Enemy units which resided in the area but moved out of the area will become available as soon as they leave the area.

When the path planner selects enemy targets to attack, a maximum threat value is sent to the

pathfinder to be used as a cutoff value. Cutoff means that the pathfinder will not traverse nodes that have a higher threat than the specified value. The cutoff value is calculated from the combined DPS of the group. The result is that an attack group will not attempt to pass through enemy fire which is more powerful than they can destroy.

The path planners way of locating targets in combination with adjusting the size of groups enables military units to be assigned to attack enemy targets in an optimal way. The target is optimal both in terms of assigning targets of appropriate difficulty for a group, and in having low cost of travel. This method also offers the possibility of filtering potential targets down to important targets, such as expensive enemy base buildings.

If the path planner can not find a target for a given attack group, the group is sent to the periphery of the friendly base. Reasoning about where the periphery of the friendly base is done with the help of the K-means algorithm, and one of the least safe "means" of building positions are used in this case. The least safe "means" are the ones on the outer border of the friendly base. Placing idle combat units at the outer border of the base enables quicker response when defending, since enemy attacks are most likely to come in those areas. Micromanagement behavior might maneuver units a short distance away from the buildings, but units will never receive a path to a destination which is further back than the outermost buildings unless the enemy has advanced that far.

Defending the base against enemy attacks is a higher priority than attacking. The procedure for planning defense is more complex than attacks, with regards to finding paths. However, defense targets are not reserved as soon as a group has been sent to handle them. This is because defense is a much more rare occurrence than attack, and because it takes priority. The procedure for planning defense paths first looks at a potential group of friendly units, then makes a pathfinder search to see if they are able to travel to any base locations, and locates the most safe (determined by k-means) base location. This base position is then used as an origin for a search for nearby enemies, supplying to the pathfinder a cutoff value for the maximum distance away from the base. There is however no use of the threat map in this search, since its better to risk death while defending than not to defend at all. Enemy units which are far away from the base are not considered targets to defend against since they are not a direct threat to the base. If this second path to an enemy is found, it means that there is an enemy unit near the base. In that case, a third path is made for the defending group, from their current position to a position resulting from the second search. Specifically, the goal node of this third search is near the found enemy but closer to the base, at a point along the second path from the base to the enemy. Using these three searches, the defending group is sometimes able to intercept enemy units before they reach the base. Performing pathfinder searches locating nearby base buildings is also made necessary by the fact that TA Spring games can happen on multiple continents at the same time, or units can be otherwise located in positions from which they can not reach every part of the base.

Using the various pathfinder searches can become a computationally heavy operation, so they are spread over several frames. A cheaper operation to do is to verify currently active paths. There is a procedure making this check in the code controlling group behavior, and the military control module calls this procedure with short intervals so that it is able to retrieve information about the

status of the planned path. The test case involves two checks. The first check consists of simply retrieving the value in the threat map at each waypoint along the planned path, and comparing them to the group's tolerance level for threat. The second check verifies that targets have not moved out of the destination area. If either of these checks fail, the planned path is discarded.

Replanning is the concept of discarding a previous plan for the benefit of a new plan from the current state to the goal state when the current state is not the expected state. The "plan" in our case is the route to travel to an assigned target. Figure 15 and figure 16 show an example of path replanning. In figure 15, our unit has received a path to a destination which avoids enemy fire. Seconds later in figure 16, the threat in the planned path for our units has increased considerably because an enemy unit has moved closer. This causes the path to be invalid, and the path planner assigns them a new path which has been adjusted for the change in the situation.

Path replanning is also done with long intervals regardless of whether verification of the previous path failed. This is because the efficient algorithm for checking if the current path is valid is not sufficient for judging the changes in the general game state that might make another route more optimal. Replanning is done in order to optimize the path with regards to changes in the general game state, such as another attack route opening.

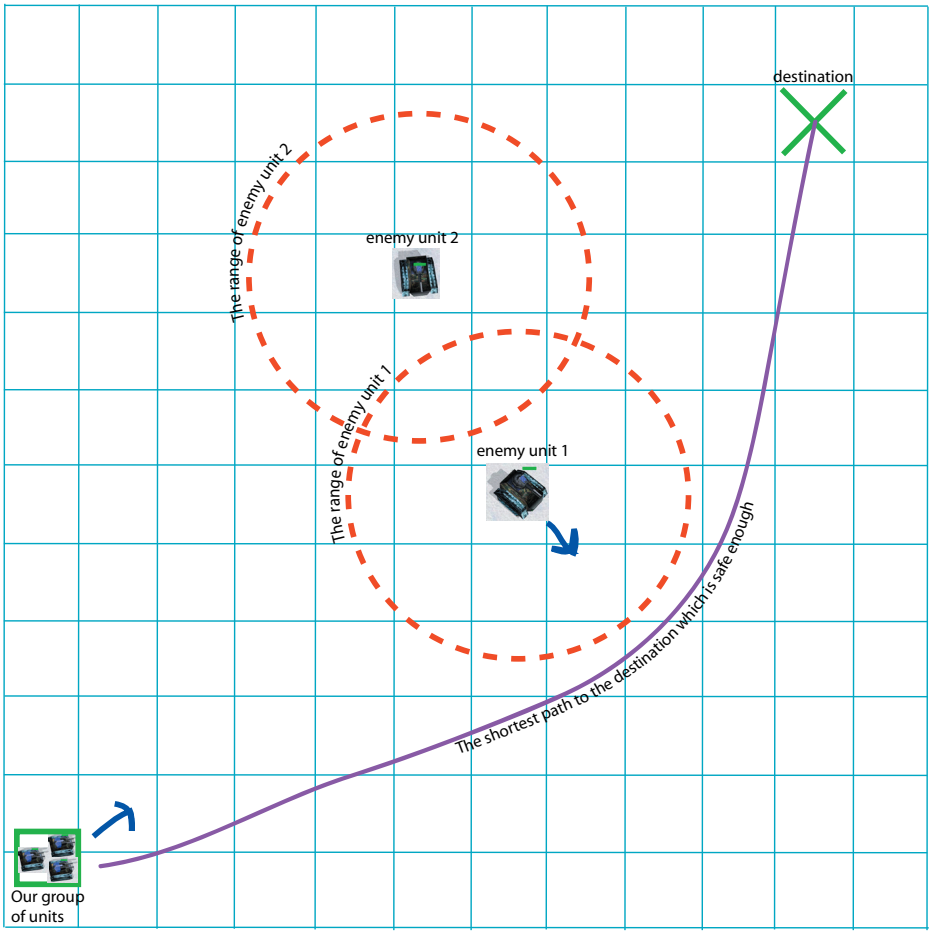


Figure 15: Path planning to a predefined target using enemy threat as travel cost.

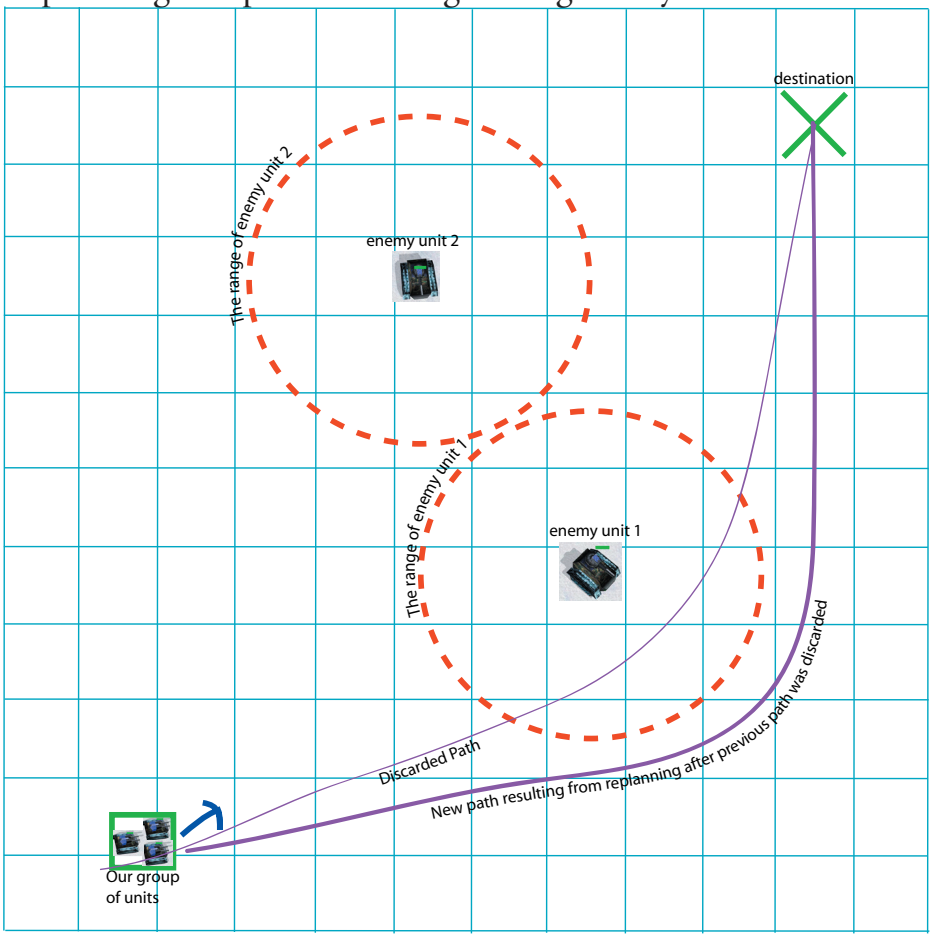


Figure 16: Path replanning after the first path was discarded due to too high enemy threat.

7.4 Micromanagement of Units in Combat

Achieving realistic unit micromanagement goes towards the goal of designing an AI opponent that acts intelligently. Micromanagement means the second-to-second low level decisions about where to move, and which enemies to attempt to destroy first. Although major considerations have to be made towards the higher level strategy in terms of planned paths and targets, this issue can be described as a separate topic with a scope encompassing only the short-term decisions while firing at enemies. The TA Spring AI presented in thesis implements micromanagement for combat units in an effort to play more realistically and be a more skilled and entertaining opponent for human players. Micromanagement is done in cases where groups of units are in range of enemy units and are engaging them in combat.

The micromanagement behavior achieved can be described as two actions:
* Maneuvering each unit to more beneficial positions when possible.
* Giving attack orders making each unit focus fire on the same target.
* Staying in range of as few enemy units as possible (if beneficial), but at least one.

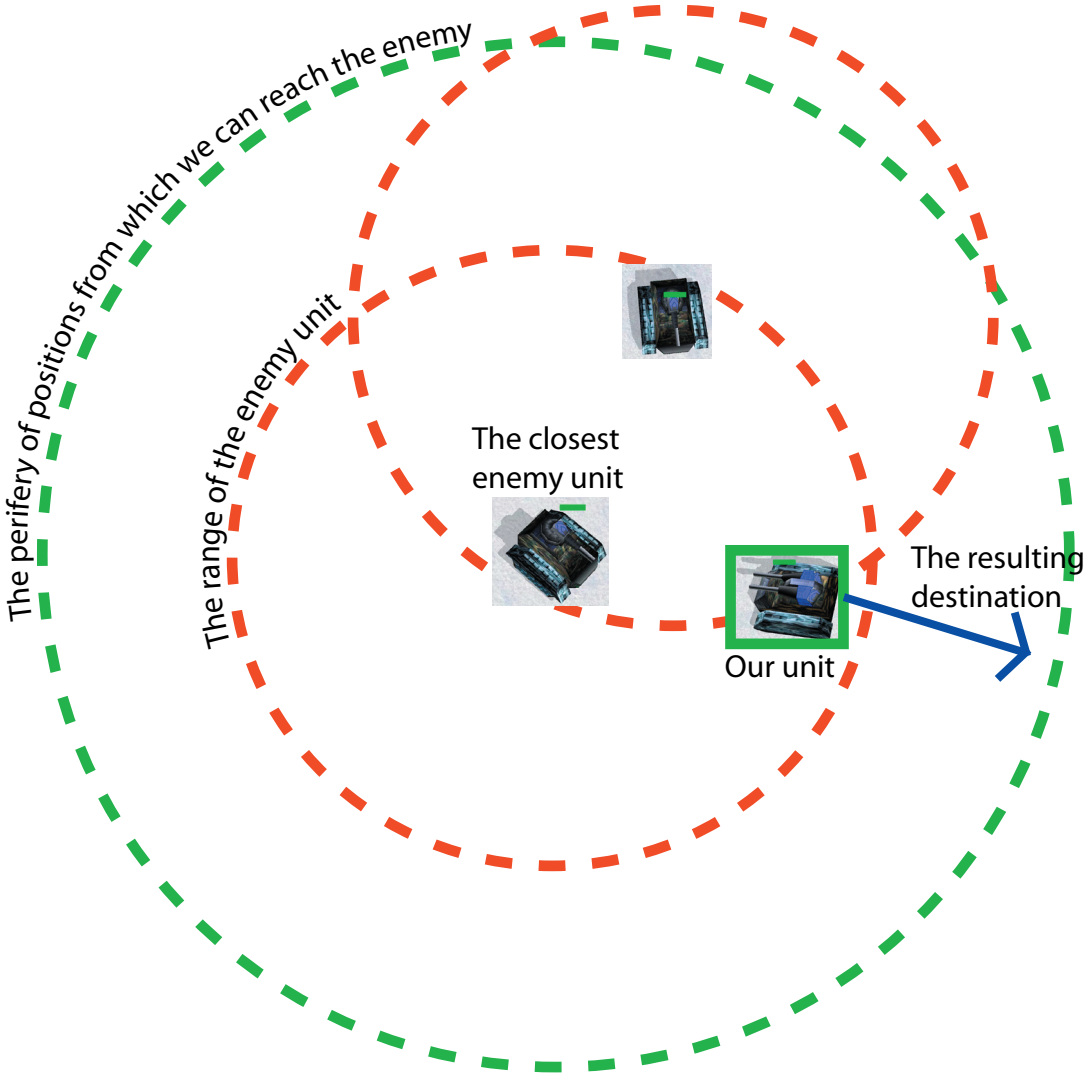


Figure 17. An example situation where the AI micromanages a combat unit. The unit is moved out of the attack range of the enemy units, but stays close enough to be able to fire at the enemy.

Figure 17 illustrates the setting where the pathfinder call is done. The combat unit belonging to the AI (the one with a green square around it) has the enemy unit well within its range, in fact it is closer than optimal. This is the situation where the maneuver function in the pathfinder is called. The green dashed circle with the closest enemy unit in the center indicates all the positions where it would be optimal to be with regards to our units maximum attack range. Those points along that circle are added as multiple goal nodes for the pathfinder search. Then, the threat map which includes the damage that the enemy units can inflict is used as the travel costs in the pathfinder search, and the search is started. The best path will then be the path which most quickly is able to get our unit to a position along the green dashed circle periphery, and while doing so minimizing the amount of time spent within attack range of the enemy units. This is because moving within the attack range of the enemy units bears a higher cost for the pathfinder, since its the damage that the enemy is able to inflict at each position which determines the cost of travelling. The blue line in figure 17 marks the resulting path which is given as a move order to the unit. When the unit controlled by our AI is at the optimal position, a new order to attack the closest enemy unit will be given, and it is checked again if a more optimal position can be achieved, and then the same procedure is repeated if necessary. This is all that is needed to maneuver units efficiently in combat, and if done properly results in elegant behavior during games. The code part performing micromanagement is over 100 lines long, but this is the part performing the maneuvering to each units maximum attack range:

```
//where is the best position to maneuver to, use pathfinder:
float cost = ai->pather->ManeuverToPosRadius(&moveHere, myPos, enemyPos, myMinRange);
float dist = myPos.distance2D(moveHere);
//Will the target still be in line of sight at the goal position?
//Is the position between the proposed destination and the enemy higher than the average
// of mine and his height? -cheap line-of-sight test so we dont move behind an obstacle
bool losCheck = ((moveHere.y + enemyPos.y)/2.0f) + UNIT_MAX_MANEUVER_HEIGHT_DIFFERENCE_UP > ai-
>cb->GetElevation((moveHere.x+enemyPos.x)/2.0f, (moveHere.z+enemyPos.z)/2);
//get threat info at current position and at target location:
float currentThreat = ai->tm->ThreatAtThisPoint(myPos);
float newThreat = ai->tm->ThreatAtThisPoint(moveHere);
//Will moving result in less threat of enemy fire, and is it far enough
// to make a difference, and is it in line of sight?
if (newThreat < currentThreat && dist > max((UNIT_MIN_MANEUVER_RANGE_PERCENTAGE*myMinRange),
UNIT_MIN_MANEUVER_DISTANCE) && losCheck) {
    //Approximate the time before another maneuver update will be required
    myUnitReference->maneuverCounter = ceil(max((float)UNIT_MIN_MANEUVER_TIME/frameSpread,
(dist/myUnitDef->speed)));
    //Order the unit to move to the new position
    myUnitReference->Move(moveHere);
}
```

This behavior is illustrated in Figure 18. The green circle shows which enemy units are being targeted, and all the red circles indicate the maximum range of the red units. The red circles indicate the area which the red units are able to reach with a projectile attack from their current position. By looking closely at the green circle it is noticeable that the periphery of several red circles align close to it, and the closest blue unit is within each of the red circles. This means that many of the red units have been maneuvered to their maximum attack range from the blue units, with the exception of a few who are in the process of adjusting their position in order to achieve the same goal. This is a tactical advantage for the red AI controlled player, since all his units are able to shoot at the blue enemy, while the blue player has its units in some disarray. If the enemy moves closer, the AI micromanagement procedure will adjust the position of each red unit further away, and if the enemy begins to retreat the AI will order units to chase.

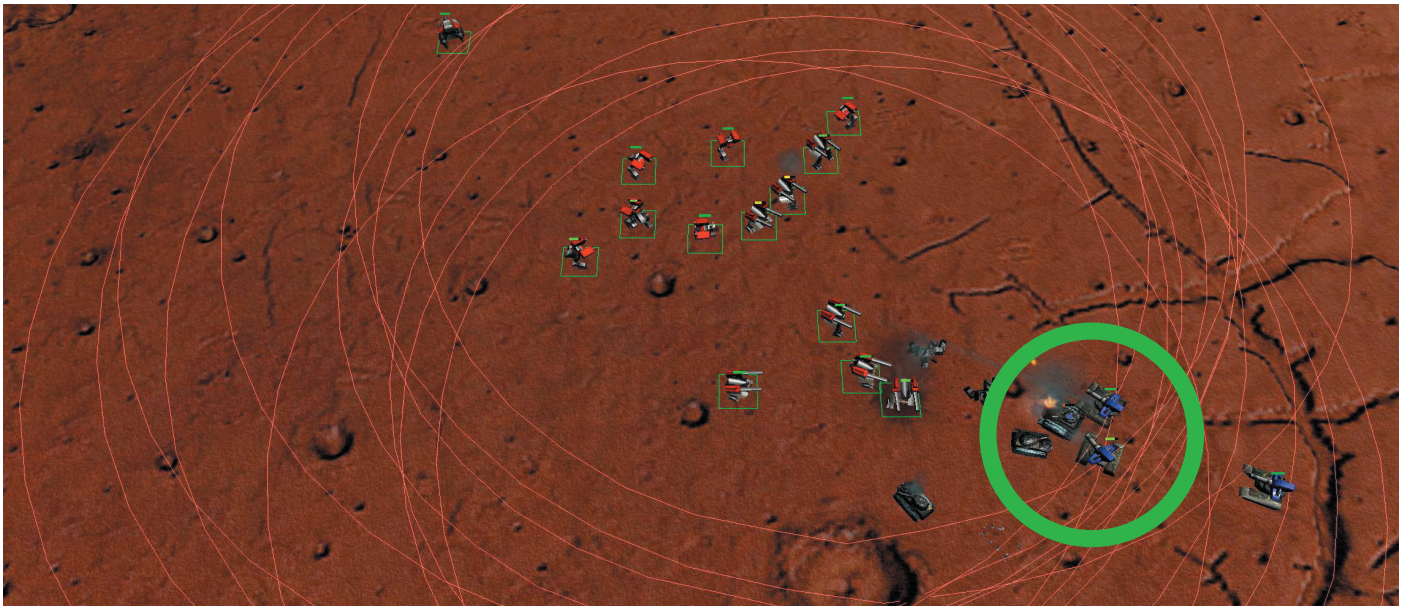


Figure 18. Red AI controlled units in the process of aligning themselves at their maximum attack range of enemy blue units, and attacking the same unit. The large thin red circles mark the maximum attack ranges of the red units. Note that the blue units marked by the green circle are all in range of the red units, which is one of the desired combat behaviors.

In addition to adjusting positions, the micromanagement routine orders each unit in the group to attack whichever enemy is closest. Micromanaging weapons targeting is called “focus fire” in gaming terminology. Focus fire is the second part of the micromanagement routine, and it refers to the general knowledge that a benefit can often be gained by finishing destroying a single enemy before engaging the others. It is fairly simple, although a lot of things can be taken into consideration when choosing which target to focus on first when there are several nearby enemies. Most TA Spring weapons are ballistic and their accuracy is better at short range, so our approach in the TA Spring AI is to focus on the enemy closest to the AI controlled units. This is also sensible with regards to the maneuvering to maximum attack distance as explained above, since it is then more likely that only few (but at least one) enemies are within range of all the AIs units. In cases where not all enemies in range are able to return fire, the AI will select targets that are able to return fire first, before destroying other units and buildings.

All this micromanagement mentioned is done through a fairly basic technique, but it relies on various infrastructure of the AI. It relies both on the threat map and the pathfinder, and maintaining and up-to-date threat map of high enough resolution for this purpose can be time consuming.

7.5 Defense Placement Algorithm

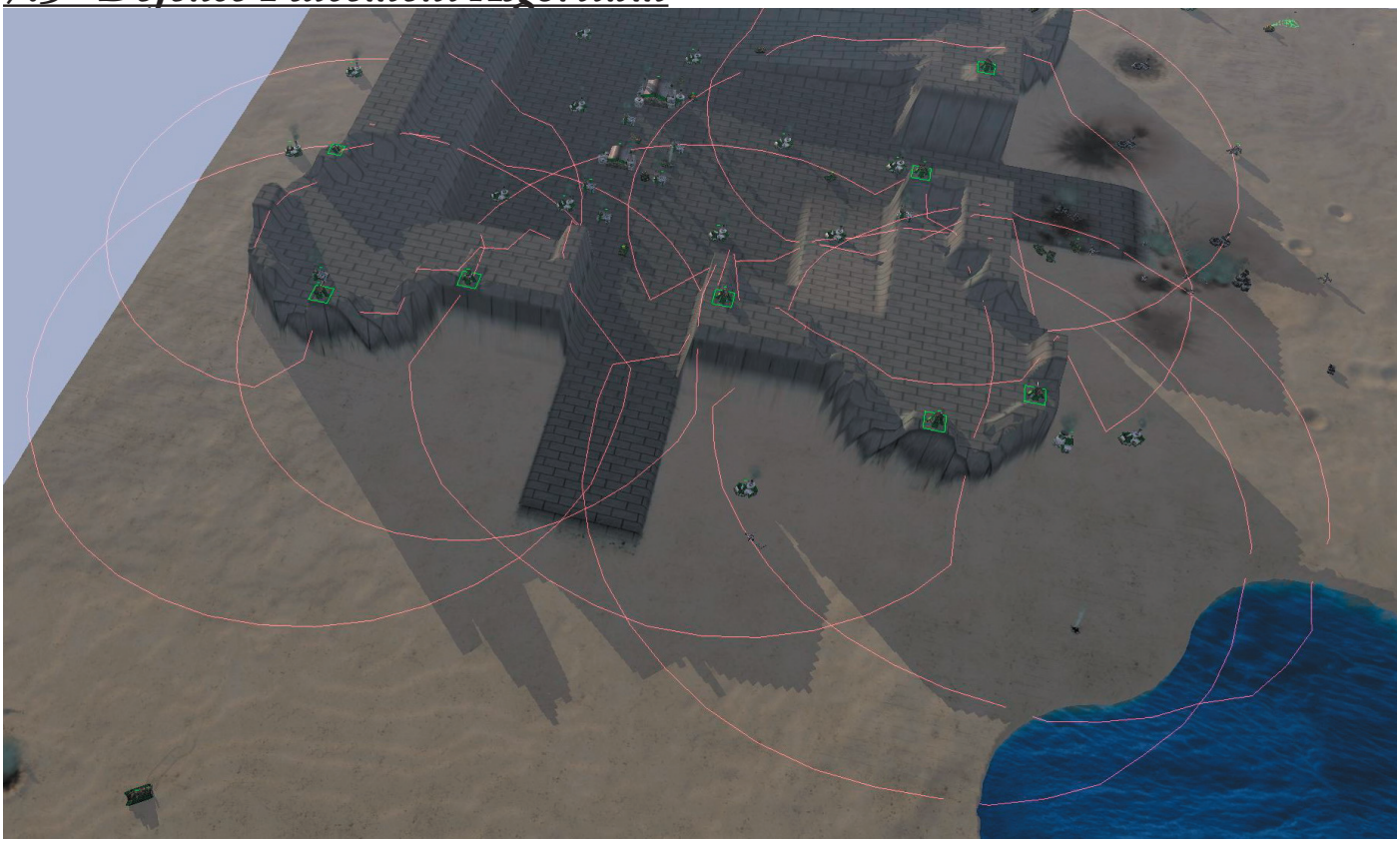


Figure 19: A screenshot of KAI playing on the “Castles” map. The green squares enclose defensive structures, and the red circle around each defensive structure shows how far it can shoot.

The defense placement algorithm is as the name suggests an algorithm for finding optimal locations for defensive structures, and it uses terrain analysis for doing so in terms of analyzing travel routes and height. An example of the resulting defensive structure positions is shown in figure 19. This algorithm was originated by Leon Palm and is not one of the main focuses of this thesis.

8 Discussion

8.1 Evaluating Achievement of Research Goals:

Goal number 1: An RTS AI CP capable of defeating experienced human players.

The goal of being able to defeat experienced players was met, and this was verified by various players in the TA Spring community. An example of this is shown in detail in Appendix 1, and it was also verified in various tests both by myself and other players in the TA Spring community. The AI CP was able to play in all of the major "mods" for TA Spring, which was a desired feature, and KAI performed well in all supported mods. Only one mod out of the more than 20 mods tested was found which KAI did not support well, it is named "Swarm v.0.0", is in alpha stage of development, and has a high amount of tiny units with huge attack range. The reason why KAI does not support this mod well is because the huge attack ranges of some kind of secondary weapon causes the frequent updates of the threat maps to become time consuming when a large enough amount of units has been built, but it is something that is likely to be fixed by improving the mod. In general, this goal was met.

Goal number 2: An RTS AI CP with combat behavior and detailed unit control similar to a human player.

The micromanagement routine guiding combat behavior in combination with the path planning went a long way towards this goal. Similar to a human player, KAI attempts to find the best attack routes, and adjusts its behavior with changing circumstances. It also moves units in groups like human players do, and micromanages units while they are fighting like good human players do. The version of the AI which was released to the TA Spring community was tested by many players, and the feature which resulted in the most positive feedback was the micromanagement routine, because this was an unusual feature for an RTS AI. The example in appendix 1 as well as numerous other tests done during development supports this claim. However, doing detailed micromanagement too frequently has a negative side effect as far as the goal as it is stated is concerned. Adjusting the position of every single unit within seconds does give the impression of intelligence. The interviewee in the experiment in appendix 1 hints towards simultaneous micromanagement going a bit overboard with regards to imitating humanlike behavior, since too many actions are sometimes being taken at once. Adjusting behavior to result in lower skill is easier than the other way around, and even if unrealistic behavior is the result, it is still better than the alternative of gaining advantages by creating free units or resources, as is common in many RTS game AIs.

Evaluation of the AI Architecture.

The AI architecture enabled the implementation of an AI CP which accomplished the goals set for this thesis. The various components of KAI succeeded at adapting their behavior to changing circumstances in terms of enemy actions. Different parts of the AI architecture were able to perform their tasks so that a general strategy emerged. The micromanagement behavior rules were a particularly important contributing factor towards emergent behavior.

Evaluation of the AI Algorithms.
The threat map served its purpose, providing detailed information about enemy threats for other algorithms such as the path planner, pathfinder and the micromanagement routines. It is definitely an algorithm worth considering when developing of a new RTS AI. However, updating a highly detailed threat map frequently can be a time consuming process. There is also the possibility of refining the threat map to separate between types of weapons damage, but this requires multiple data for each position, and would therefore be considerably more computationally expensive.

The K-means algorithm was intended to be generally useful, but ended up having only one purpose. It was used for making distributed average base positions, and worked well for enabling the agents controlling combat units to reason about where the base is. It was also attempted to restrict areas to build in by defining safe and less safe regions using k-means and distance to the enemy. The result was too restrictive with regards to expansion and therefore less than optimal. The limits to its usefulness is clearly shown, and also it was difficult to adjust the details for how it should work other than by adjusting the total number of means. In general it was not a very useful algorithm, it only made some of the path planning operations related to defense more efficient. Another approach to base planning and reasoning about areas on the map is needed.

The path planning algorithm in combination with threat maps and known enemy units positions was one of the most useful algorithms, succeeding in distributing combat units across several enemy targets at once. Its strength came from being able to create a detailed attack plan to a suitable targets, taking into account terrain, the strength of the attacking units, the threat along the possible paths, and the strength of the potential targets, and doing so in a reasonably efficient way. However it does not solve the problem of island hopping, which is a difficult problem[16]. Island hopping is the problem of using transports to move ground units between land areas separated by obstacles such as water, and there are some approaches towards solving this problem[28]. This problem is avoided in KAI by having a preference for building amphibious units and air units on island maps whenever these are available.

The micromanagement behavior was one of the biggest successes of the AI (sometimes in combination with the path planning), at least as far as the impression it made on players is concerned. The interview in appendix 1 supports this claim, and so does the general feedback received about KAI from the TA Spring community.

Many problem areas have multiple solution, this fact is illustrated by the fact that a cluster analysis algorithm could be used for reasoning about base perimeters, therefore it should not necessarily be concluded that specific algorithms are the best way to solve a given problem just because they are shown to work.

8.2 Contribution
First of all, the goals for this thesis has been outlined and accomplished to a large extent. There are also certain contributions as a product, such as wide support for mods and ease of install due to no extra files.

It has been shown that the architecture is viable for using to implement an RTS AI CP, with its usage of the concepts of situational awareness and emergence.

There are also some new contributions made by the AI algorithms and their usage. The creative use of pathfinding algorithms for target allocation is one such contribution.

Also, an effort has been made to include explanations of concepts common in gaming, in order to help bring this knowledge into scientific literature.

8.3 Suggestions for Future Research

There is a number of areas where improvements can be made to the state of the art in RTS AI. We have earlier pointed to the multitude of practical problems that can be solved, and there are possible solutions to each of them. If actions are performed in different ways in the lower level agents, this may result in other possibilities for higher level agents. For instance, using threat maps to help in path planning was a possibility that arose from implementing influence maps at a lower level agent for a different purpose, namely pathfinding. Instead it provided an opportunity to use it for selecting targets as well as finding safe paths. Therefore, new reasoning related algorithms at low levels may be helpful when doing high level reasoning in new ways, in a network of agents.

Perhaps a new approach can be found to the problem of combat unit allocation. There are so many different unit properties with an effect on their optimal use in combat, such as speed, DPS, strength, movement type, weapon range, various weapon properties relating to damage and projectile speed, and many other things. This is a difficult problem to overcome, and perhaps there are better solutions to the problem than trying to simulate their relative strengths with calculation.

Learning is considered to be an important aspect of intelligence. However, we find it unlikely that a detailed and complete plan for dealing with the extremely high amount of states and frequent changing in situations in an RTS environment can be made in advance. Instead, agent approaches to modeling intelligence in combination with emergence points towards an approach of applying reasoning mechanisms at different levels in an agent hierarchy, and we see no reason why learning can not be applied for this purpose. The challenge is identifying specific goals that can be met with learning mechanisms, whether it is on a high level that depends on lower level agents, a low level sensing the environment, or at any level in between.

Another aspect of an RTS AI CP which needs to be solved is the problem of reasoning about areas of the map in a way that supports the same kind of reasoning as human players seem to be able to do easily. For example for the purposes of planning where the player will build his base, based on the terrain and paths near the player's starting position. There are also various issues relating to using the terrain and formations for tactical benefits which human players seem to be able to grasp, but are very rare to see in game AI.

The various uses found for pathfinder searches in this work points towards the need of finding more efficient pathfinder searches which work for more advanced types of searches. It would be beneficial for the RTS AI field if a pathfinder algorithm is found, which supports multiple separate

travel costs (threat and terrain effects on travel speed), or enables cached or best-first searches in cases of multiple goal nodes or multiple points of origin. However, if such a pathfinder is to be used for searching a frequently updated influence map, then caching seems impossible.

References

[1]: Nils J. Nilsson, (1998, April 1) "Artificial Intelligence - A New Synthesis" - ISBN: 1-55860-535-5
[2]: Alexander Seizinger, (2006, July 1) "AAI 0.70 Released" - The only public documentation of the TA Spring AI "AAI" v0.70 and it's learning system. - [Online] Available: http://taspring.clan-sy.com/phpbb/viewtopic.php?t=6094
[3]: Dave C. Pottinger (2000) -"Terrain Analysis in Realtime Strategy Games" - [Online] Available: http://www.gamasutra.com/features/gdcarchive/2000/pottinger.doc
[4]: Benjamin Wootton (2006) "Designing for Emergence" - University of Leeds - "AI Game Programming Wisdom 3" pp 45-55, Steve Rabin (ed), Charles River Media, Inc. ISBN 1-58450-457-9
[5]: Mehmed Kantardzic (2003) "Data Mining - Concepts, Models, Methods, and Algorithms" pp 129-132 - ISBN 0-471-22852-4
[6]: Alex J. Champandard (2003) - "Path-Planning from Start to Finish" - [Online] Available: http://ai-depot.com/BotNavigation/Path.html
[7]: Steve Rabin - "Introduction to Game Development", Steve Rabin (ed), Charles River Media, Inc. ISBN: 1-58450-377-7
[8]: The Entertainment Software Association (2006) - Essential Facts 2006 - [Online] Available: http://www.theesa.com/archives/files/Essential%20Facts%202006.pdf
[9]: Bruce Geryk "A History of Real-Time Strategy Games" - Gamespot Magazine - [Online] Available: http://www.gamespot.com/gamespot/features/all/real_time/
[10]: "Dune - The Official Website" - [Online] Available: http://www.dunenovels.com/
[11]: Blizzard Entertainments Battle Net - [Online] Available: http://battle.net/
[12]: TA Spring Project - [Online] Available: http://www.ta-spring.com/
[13]: StarCraft by Blizzard Entertainment - [Online] Available: http://www.blizzard.com/starcraft/
[14]: Homeworld 2 by Sierra Entertainment - [Online] Available: http://homeworld2.sierra.com/
[15]: Christian Baekkelund (2002) "Academic AI Research and Relations with the Games Industry", Massachusetts Institute of Technology (MIT), AI Game Programming Wisdom, Steve Rabin (ed), Charles River Media, Inc. ISBN 1-58450-077-8
[16]: Bob Scott (2002) "Architecting an RTS AI", Stainless Steel Studios, AI Game Programming Wisdom, Steve Rabin (ed), ISBN 1-58450-077-8
[17]: Paul Tozour (2002)"The Evolution of Game AI", Ion Storm Austin, AI Game Programming Wisdom, Steve Rabin (ed), Charles River Media Inc., ISBN 1-58450-077-8
[18]: Description of the FBI file format used to store unit definitions in a TA Spring "mod". - [Online] Available: http://taspring.clan-sy.com/wiki/Units:FBI
[19]: J.M.P. van Waveren (2001) "The Quake III Arena Bot", University of Technology Delft, [Online] Available: http://www.kbs.twi.tudelft.nl/docs/MSc/2001/Waveren_Jean-Paul_van/thesis.pdf
[20]: "Saga of Ryzom", Nevrax, [Online] Available: http://www.ryzom.com
[21]: "Voronoi diagrams for spatial partitioning", [Online] Available: http://mathworld.wolfram.

com/VoronoiDiagram.html
[22]: “TA Spring source code repository”, [Online] Available: http://svn.berlios.de/viewcvs/taspring-linux/trunk/
[23]: Mike McShaffry (2005) “Controlling the Main Loop”, Game Coding complete Second Edition pp. 159-161, Paraglyph Press, ISBN 1-932111-91-3
[24]: Marvin Minsky (1986), “Society of Mind”, Simon & Schuster Inc. ISBN 0671657135
[25]: Derek J. Smith (2004) “Situation(al) Awareness in Effective Command and Control”, [Online] Available: http://www.smithsrisca.demon.co.uk/situational-awareness.html
[26]: Emmanuel Chiva, Julien Devade, Jean-Yves Donnart, Stéphane Maruéjouls, (2004) “Motivational Graphs: A New Architecture for complex Behavior Simulation”, MASA Group, AI Game Programming Wisdom 2, Steve Rabin(ed), Charles River Media Inc. ISBN 1-58450-289-4
[27]: Guest lecture on learning to win complex games (TIELT) held at NTNU, Trondheim, Norway (2006, Sept 27), David W. Aha, Navy Center for Applied Research in Artificial Intelligence, Naval Research Lab.
[28]: Shawn Shoemaker (2004) “Transport Unit AI for Strategy Games”, Stainless Steel Studios, AI Game Programming Wisdom 2, Steve Rabin(ed), Charles River Media Inc. ISBN 1-58450-289-4

Appendix

Appendix 1: Experiment

With the help of a volunteer experienced TA Spring player with an interest in AI CPs, a more detailed experiment was performed. The experiment took the form of the player playing against KAI, and it was followed by a short interview to get his opinion of KAIs performance. During development, many experiments were run while improving KAIs combat behavior, both against human players and other AIs, and performed very well. However, the experiment in this appendix is well documented and is therefore included in the thesis for reference. The version of KAI used was 0.11, which is the version released to the community 18th of June in 2006, and was the most commonly used version of KAI for TA Spring players for several months.

The Settings of the Experiment:

The mod selected was "Expand & Exterminate v0.163".

The map used was "Silver Lake", which is a map including most of the possible features for TA Spring maps such as sea, mountains, steep hills, and flat areas with multiple attack routes. Figure 20 shows an image of the map. The amount of available resources on the map is medium to high. All other settings were at default value.

The Match:

The match lasted approximately 30 minutes. After three minutes, both players had a few base buildings as shown in Figure 21. In the following minutes there were various minor battles involving 1-5 units in both the bottom right corner and in the top of the map. During this time KAI is able to expand its economy more than the human player is, partially due to that the player has to focus on the battles that are being fought. The amount of units in combat on both sides increased steadily over the next minutes, and the context of the battle was mostly of KAI sending attack trying to kill the forces that the human player was getting ready to use in an attack.

Eleven minutes into the game, the human player launches a major attack. Figure 22 shows this attack while in progress. The attack is partially successful in that it destroys a major part of KAIs base in that area. However, as figure 23 shows, it was not enough to tip the balance, and KAI still has a lot of resource buildings on the left side. KAI still has more resource income than the human player after this time, and still has 101 units and buildings in total. The player has only 54 units and buildings left after the attacking forces are destroyed. 28 minutes into the game, KAI has pushed on with a number of attacks and has further increased its resource production. Figure 24 shows that KAI now controls most of the map, and that the human player is almost defeated. The game ends shortly after.

Figure 25 shows a graph illustrating the number of units built by both sides during the 28 minute long battle. Note that from 0 to 3 minutes, the amount of units produced by both sides were largely the same, and after that KAI gains an advantage. The first attempted attacks made by KAI were made during the period after 3 minutes, and this is the period where KAI starts to gain an advantage in military power. Increasing the AIs military power is made possible because of additional resource income, due to having contructed more resource buildings than the human players while simultaneously attacking.

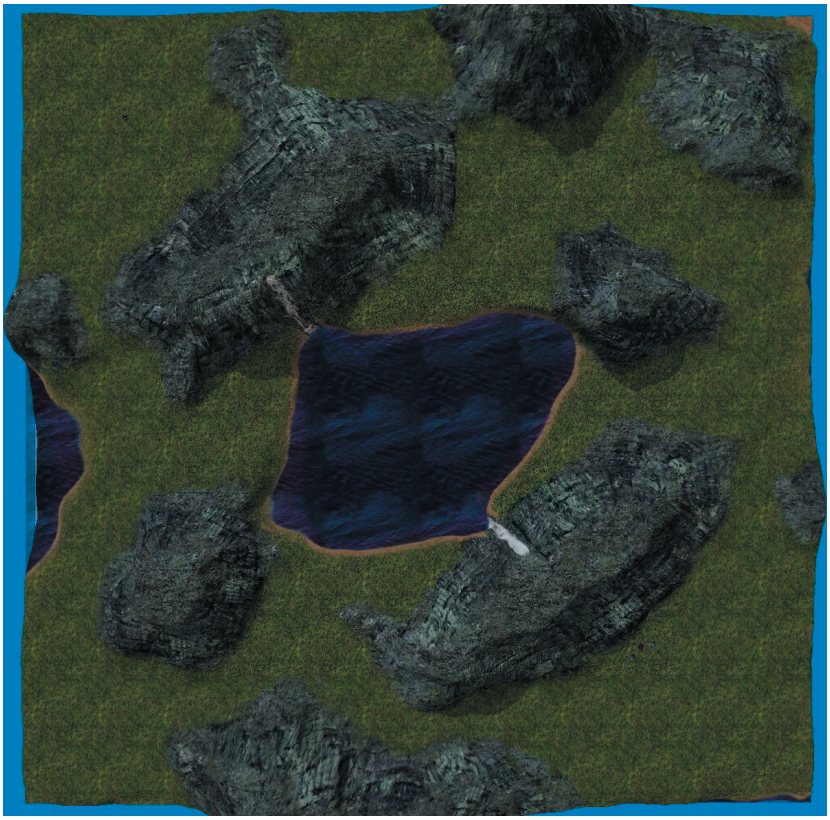


Figure 20: The map used in the experiment.

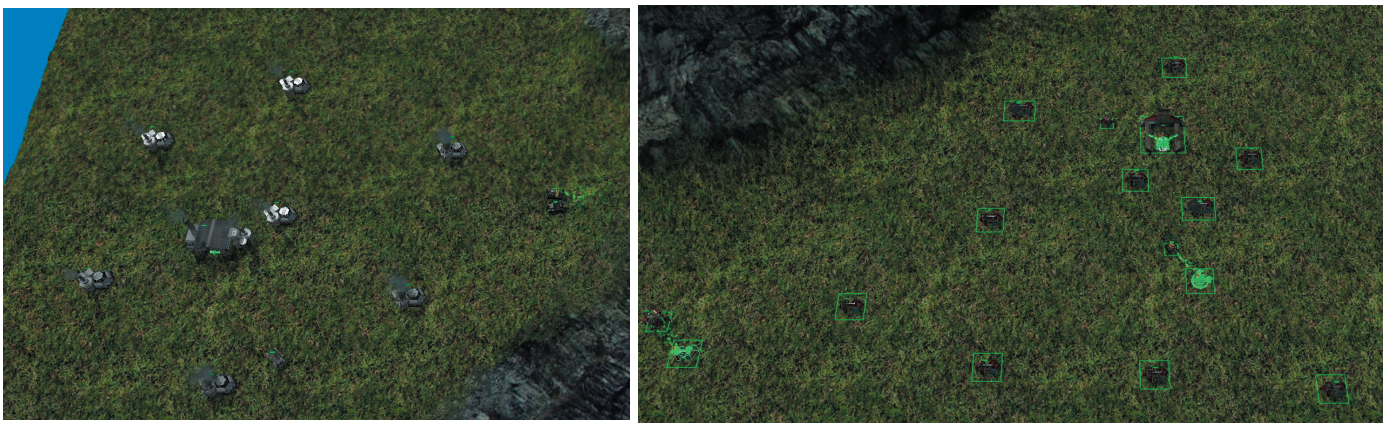


Figure 21: The bases of the human player(left) and the AI CP(right) after a few minutes.

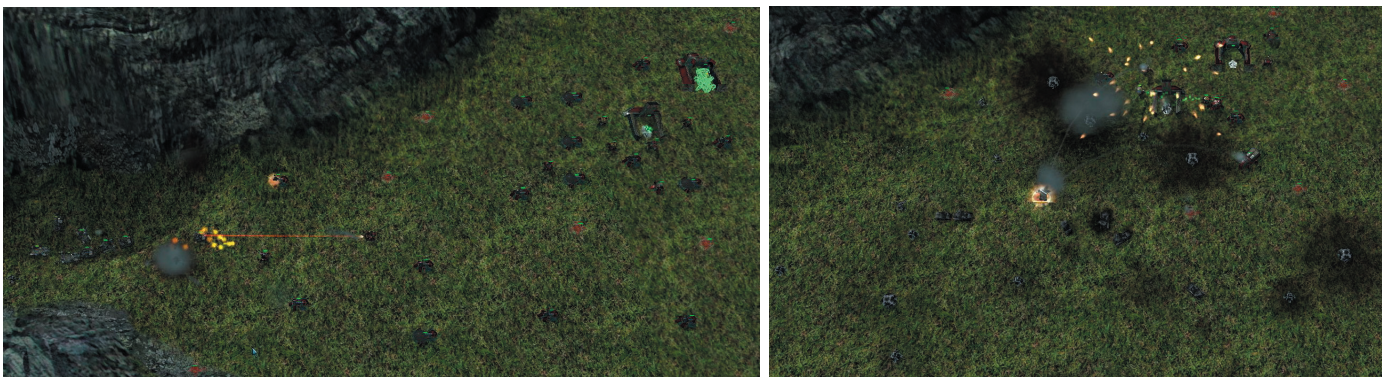


Figure 22: The main attack made by the player. The left image shows the players forces as they enter the enemy base. The right image shows that a large portion of the base has been destroyed shortly after.

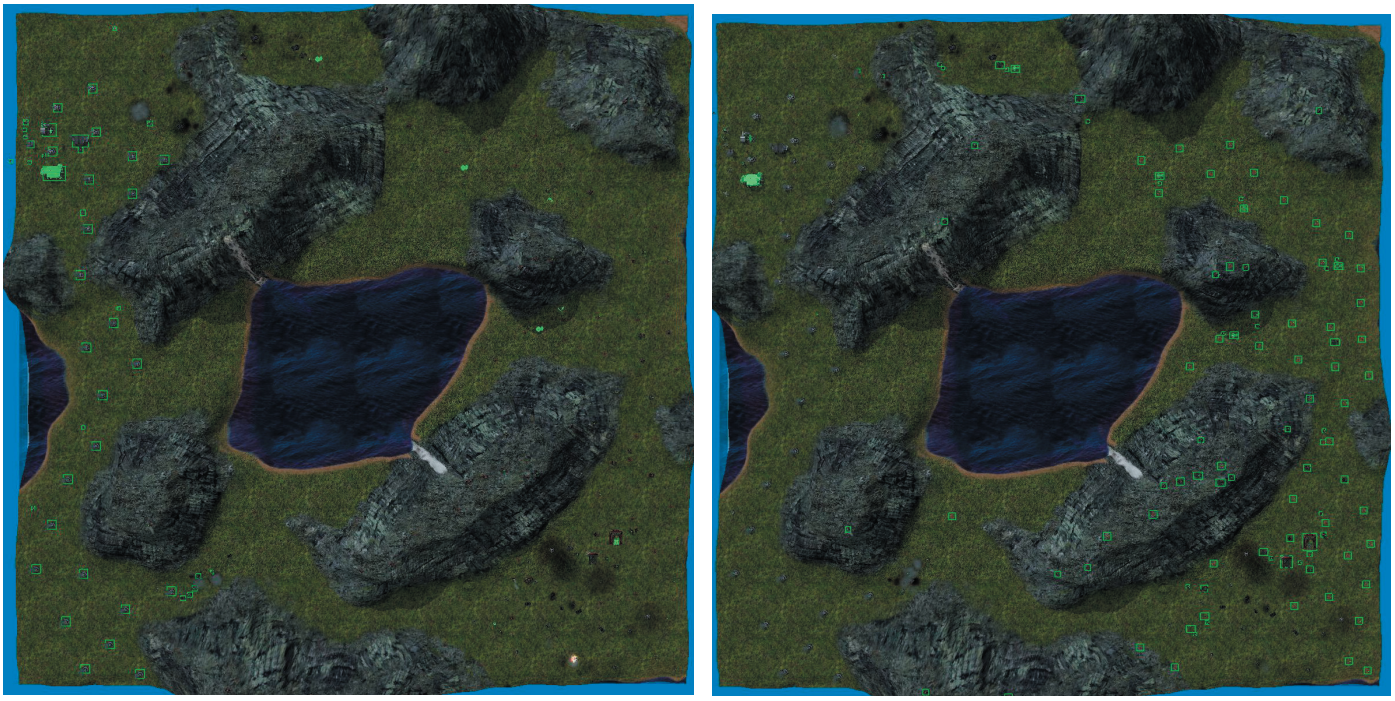


Figure 23: The status and number of units of the human player (left) and the AI CP (right) just after the major attack done by the human player.

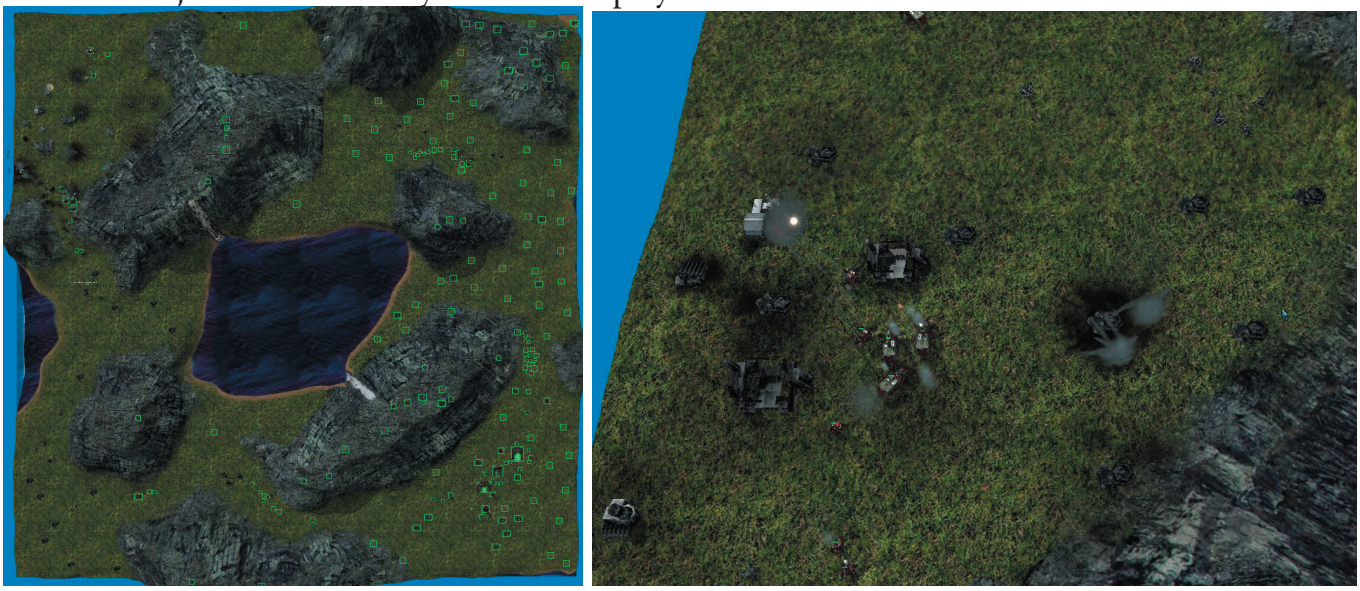


Figure 24: The status of the game after 28 minutes. The left image shows the amount of AI CP controlled units, and the right image shows the human player's last buildings being destroyed.

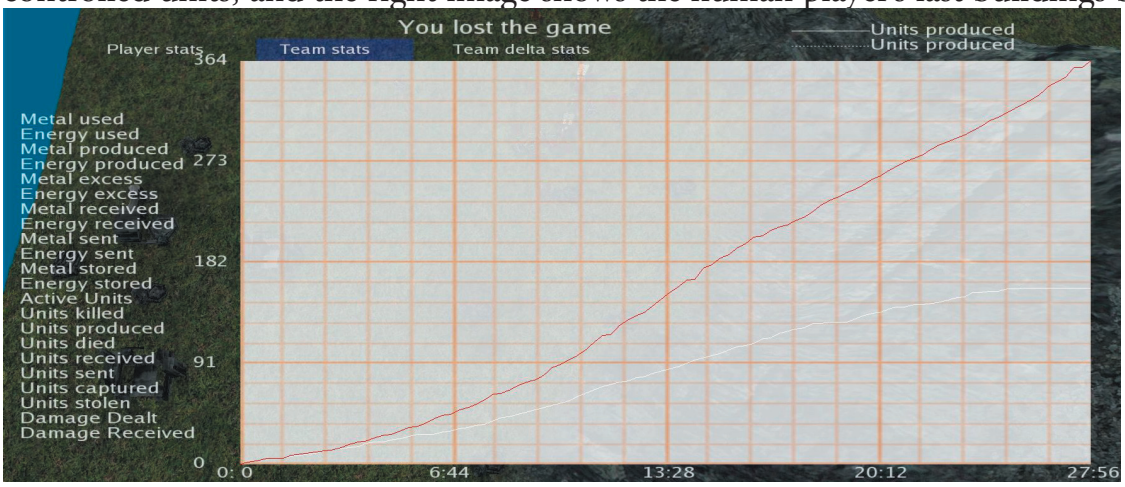


Figure 25: A graph showing the number of units produced by KAI(red) and the player(white).

Interview with the Player After the Match.

Sindre Berg Stene: Did KAI play well, in your opinion?
Tully Elliston: KAI played very well in my opinion, with superior micro, flanking and use of high-ground.

SBS: Did you get the impression that KAI was cheating?
TE: I think KAI must have either cheated or have lots of stealthy cams about (which I doubt) because its mortar bots seemed to have lots of LOS.

SBS: Was the match entertaining? Did it serve the purpose of an AI?
TE: Yeah it was just as fun as a human player, maybe because I knew one of its devs was watching its progress. I use KAI in my own time.. It's a good AI.

SBS: What did you think about the micromanagement?
TE: LOL very impressive. Certainly better than I could do without three mouse hands.

SBS: What did you think about how it chose which targets to attack and which paths to travel to get there?
TE: It didnt target particularly intelligently.. It seemed to just pick targets which were closest (attacking a defence not the 200hp builder repairing it in range etc). Paths: were good, I didnt see its const units which put defenses right on top of my base.
TE: In short the best AI for TA ever made.

SBS: Do you think perhaps the micromanagement was too detailed?
TE: LOL, too detailed micro is always good. Making the more difficult non-cheating AI makes it more fun.

(Please note that spelling in this interview has been corrected, and some additional parts of the conversation including before and after the included part were removed.)

Appendix 2: Copyrights

This thesis references a number of registered trademarks, which are the property of their respective owners.

Dune 2 is abandonware, and therefore freely distributeable and not guarded by copyrights other than in name.

Total Annihilation is the property of Cavedog Entertainment.

TA Spring, and its source code, is released under a GNU General Public License, but depending on your configuration may include content from the original Total Annihilation and if so you have to own a copy of the original TA to legally play it.

Warcraft and StarCraft are registered trademarks of Blizzard Entertainment.

Command & Conquer" is a registered trademark by Westwood Studios.

"The Age of Empires" is a registered trademark by Ensemble Studios.

"HomeWorld" is a registered trademark of Sierra Entertainment.

"Sid Meier's Civilization IV" is a registered trademark of Firaxis Games.

"SimCity" is a registered trademark of EA Games.

Credits should also be given to various creators of maps and mods used, but no real names are given by them in the common released maps used to create this document.

Appendix 3: Example Code

This appendix includes some of the source code of KAI. The total mass of code is over ten thousand lines so all of it is obviously not included. All high level code relating to unit combat movement and micromanagement is included.

Pathfinder Interfaces

Function headers for the various pathfinder uses:

```
	float CPathFinder::PathToPos(vector<float3>* pathToPos, float3 startPos, float3 target);
	float CPathFinder::PathToPosRadius(vector<float3>* pathToPos, float3 startPos, float3
target, float radius);
	float CPathFinder::PathToSet(vector<float3>* pathToPos, float3 startPos, vector<float3>*
possibleTargets);
	float CPathFinder::PathToSet(vector<float3>* pathToPos, float3 startPos, vector<float3>*
possibleTargets, float threatCutoff);
	float CPathFinder::PathToSetRadius(vector<float3>* pathToPos, float3 startPos,
vector<float3>* possibleTargets, float radius);
	float CPathFinder::PathToSetRadius(vector<float3>* pathToPos, float3 startPos,
vector<float3>* possibleTargets, float radius, float threatCutoff);
	bool  CPathFinder::PathExists(float3 startPos, float3 target);
	bool  CPathFinder::PathExistsToAny(float3 startPos, vector<float3> targets);
	float CPathFinder::ManeuverToPos(float3* destination, float3 startPos, float3 target);
	float CPathFinder::ManeuverToPosRadius(float3* destination, float3 startPos, float3
target, float radius);
	float CPathFinder::ManeuverToPosRadiusAndCanFire(float3* destination, float3 startPos,
float3 target, float radius);
	float CPathFinder::PathToPrioritySet(vector<float3>* pathToPos, float3 startPos,
vector<float3>* possibleTargets);
	float CPathFinder::PathToPrioritySet(vector<float3>* pathToPos, float3 startPos,
vector<float3>* possibleTargets, float threatCutoff);
```

There are some things of interest here. KAI does not merely use pathfinding from point A to point B, it has various other uses. Among the uses are:

* When the path is to a target enemy, find a path which leads to a point which is at the moving units maximum attack range away from the target enemy.
* Search for the shortest path to any of a number of goal nodes.
* Search for a path to the maximum attack range periphery of any of a number of enemy units.
* Check if a path exists to a given target or not. This is more efficient than finding the actual path.
* Check if a path exists to any of a number of target units.
* Use the pathfinder to find a target position for a micromanagement maneuver. The path found is a single waypoint at the periphery of the circle marking the enemys maximum range.
* Find a path to any of a set of targets, but return a path to the first in the list of targets if possible, if not then return a path to the second, and so on up to the last.

Please note that the pathfinder supports replacing the node costs with a number of different lists, the most used one being a threat map. Unit move types also determine which paths are found, by filtering which positions can be moved to by a given unit. This data can be set by the following function:

```
void SetMapData(unsigned int *canMoveIntMaskArray, float *costArray, int mapSizeX, int mapSizeY,
unsigned int canMoveBitMask)
```

The AttackHandler and AttackGroup classes.

The attack handler is an essential part of the military control module and does the path planning and target location. The attack group handles following the path and micromanagement in combat.

CAttackHandler.h:

```
#pragma once
#include "GlobalAI.h"

class CAttackGroup;

class CAttackHandler
{
	public:
	CAttackHandler(AIClasses* ai);

	virtual ~CAttackHandler();

	void CAttackHandler::AddUnit(int unitID);
	void CAttackHandler::Update();
	void CAttackHandler::UnitDestroyed(int unitID);

	float CAttackHandler::DistanceToBase(float3 pos);
	float3 CAttackHandler::GetClosestBaseSpot(float3 pos);

	float3 CAttackHandler::FindSafeSpot(float3 myPos, float minSafety, float maxSafety);
	float3 CAttackHandler::FindSafeArea(float3 pos);
	float3 CAttackHandler::FindVerySafeArea(float3 pos);
	float3 CAttackHandler::FindUnsafeArea(float3 pos);

	vector<float3>* CAttackHandler::GetKMeansBase();
	vector<float3>* CAttackHandler::GetKMeansEnemyBase();

	bool CAttackHandler::CanTravelToBase(float3 pos);
	bool CAttackHandler::CanTravelToEnemyBase(float3 pos);

	//for new and dead unit spring calls:
	bool CAttackHandler::CanHandleThisUnit(int unit);

	int ah_timer_totalTime;
	int ah_timer_totalTimeMinusPather;
	int ah_timer_MicroUpdate;
	int ah_timer_Defend;
	int ah_timer_Flee;
	int ah_timer_MoveOrderUpdate;
	int ah_timer_NeedsNewTarget;

	bool debug;
	bool debugDraw;

private:
	void CAttackHandler::UpdateKMeans();
	void CAttackHandler::UpdateAir();
	void CAttackHandler::AssignTargets();
	void CAttackHandler::AssignTarget(CAttackGroup* group);
	bool CAttackHandler::UnitGroundAttackFilter(int unit);
	bool CAttackHandler::UnitBuildingFilter(const UnitDef *ud);
	bool CAttackHandler::UnitReadyFilter(int unit);
	void CAttackHandler::CombineGroups();
	bool CAttackHandler::PlaceIdleUnit(int unit);

	vector<float3> CAttackHandler::KMeansIteration(vector<float3> means, vector<float3>
unitPositions, int newK);

	AIClasses *ai;
	list<int> units;
```

```
	list<pair<int,float3> > stuckUnits;
	list<int> airUnits;
	bool airIsAttacking;
	bool airPatrolOrdersGiven;
	int airTarget;
	int newGroupID;
	list<CAttackGroup> attackGroups;
	int unitArray[MAXUNITS];
	vector<float3> kMeansBase;
	int kMeansK;
	vector<float3> kMeansEnemyBase;
	int kMeansEnemyK;
};
```

CAttackHandler.cpp:

```
#include "AttackHandler.h"

#define K_MEANS_ELEVATION 40
#define IDLE_GROUP_ID 0
#define STUCK_GROUP_ID 1
#define AIR_GROUP_ID 2
#define GROUND_GROUP_ID_START 1000
#define SAFE_SPOT_DISTANCE 300
#define KMEANS_ENEMY_MAX_K 32
#define KMEANS_BASE_MAX_K 32
#define KMEANS_MINIMUM_LINE_LENGTH 8*THREATRES
#define GROUP_MAX_NUM_UNITS 24
#define ATTACK_MAX_THREAT_DIFFERENCE 0.65f
#define ATTACKED_AREA_RADIUS  600


CAttackHandler::CAttackHandler(AIClasses* ai)
{
	this->ai=ai;
	//test: setting initial position to the middle of the map
	float mapWidth = ai->cb->GetMapWidth()*8.0f;
	float mapHeight = ai->cb->GetMapHeight()*8.0f;
	//adding getmyteam number so that different KAI instances
	//will not interfere with each others debug graphics drawing (figure id's are global)
	newGroupID = GROUND_GROUP_ID_START + GROUND_GROUP_ID_START*ai->cb->GetMyTeam();
	this->kMeansK = 1;
	this->kMeansBase.push_back(float3(mapWidth/2.0f,K_MEANS_ELEVATION,mapHeight/2.0f));
	this->kMeansEnemyK = 1;
	this->kMeansEnemyBase.push_back(float3(mapWidth/2.0f,K_MEANS_ELEVATION,mapHeight/2.0f));

	//initialize the timers used for testing CPU time taken by each algorithm:
	this->ah_timer_totalTime = ai->math->GetNewTimerGroupNumber("CAttackHandler and
CAttackGroup");
	this->ah_timer_totalTimeMinusPather = ai->math->GetNewTimerGroupNumber("CAttackHandler and
CAttackGroup, everything except pather calls");
	this->ah_timer_MicroUpdate = ai->math->GetNewTimerGroupNumber("CAttackGroup::
MicroUpdate()");
	this->ah_timer_Defend = ai->math->GetNewTimerGroupNumber("CAttackGroup::Defend()");
	this->ah_timer_Flee = ai->math->GetNewTimerGroupNumber("CAttackGroup::Flee()");
	this->ah_timer_MoveOrderUpdate = ai->math->GetNewTimerGroupNumber("CAttackGroup::
MoveOrderUpdate()");
	this->ah_timer_NeedsNewTarget = ai->math->GetNewTimerGroupNumber("CAttackGroup::
NeedsNewTarget()");

	UpdateKMeans();
	airIsAttacking = false;
	airPatrolOrdersGiven = false;
	airTarget = -1;
	this->debug = true;
	this->debugDraw = true;
	if (debug) L("constructor of CAttackHandler");
}
```

```
CAttackHandler::~CAttackHandler()
{
	//not a lot of new'ness going on
}

void CAttackHandler::AddUnit(int unitID)
{
	const UnitDef* ud = ai->cb->GetUnitDef(unitID);
	if (debug) L("CAttackHandler::AddUnit frame:"<<ai->cb->GetCurrentFrame()<<"
unit:"<<unitID<<" name:"<<ud->humanName<<" movetypething:"<< ai->MyUnits[unitID]->GetMoveType());
	ai->math->StartTimer(ai->ah->ah_timer_totalTime);
	ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
	/*
	if (ai->MyUnits[unitID]->def()->canfly) {
		//the groupID of this "group" is 0, to separate them from other idle units
		ai->MyUnits[unitID]->groupID = AIR_GROUP_ID;
		//this might be a new unit with the same id as an older dead unit
		ai->MyUnits[unitID]->stuckCounter = 0;
		//do some checking then essentially add it to defense group
		airUnits.push_back(unitID);
		//patrol orders need to be updated
		airPatrolOrdersGiven = false;
	} else
	*/
	{
		//the groupID of this "group" is 0, to separate them from other idle units
		ai->MyUnits[unitID]->groupID = IDLE_GROUP_ID;
		//this might be a new unit with the same id as an older dead unit
		ai->MyUnits[unitID]->stuckCounter = 0;
		//do some checking then essentially add it to defense group
		units.push_back(unitID);
		//this aint that good tbh, but usually does move it away from the factory
		this->PlaceIdleUnit(unitID);
	}
	ai->math->StopTimer(ai->ah->ah_timer_totalTime);
	ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
}

void CAttackHandler::UnitDestroyed(int unitID)
{
	ai->math->StartTimer(ai->ah->ah_timer_totalTime);
	ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
	int attackGroupID = ai->MyUnits[unitID]->groupID;
//	L("AttackHandler: unitDestroyed id:" << unitID << " groupID:" << attackGroupID << "
numGroups:" << attackGroups.size());
	//if its in the defense group:
	if (attackGroupID == IDLE_GROUP_ID) {
		bool found_dead_unit_in_attackHandler = false;
		for (list<int>::iterator it = units.begin(); it != units.end(); it++) {
			if (*it == unitID) {
				units.erase(it);
				found_dead_unit_in_attackHandler = true;
				break;
			}
		}
		assert(found_dead_unit_in_attackHandler);
	} else if (attackGroupID >= GROUND_GROUP_ID_START) {
		//its in an attackgroup
		bool foundGroup = false;
		bool removedDeadUnit = false;
		list<CAttackGroup>::iterator it;
		//bool itWasAttGroup = false;
		for (it = attackGroups.begin(); it != attackGroups.end(); it++) {
			if (it->GetGroupID() == attackGroupID) {
				removedDeadUnit = it->RemoveUnit(unitID);
				foundGroup = true;
				//itWasAttGroup = true;
				break;
			}
```

```
			}
			assert(foundGroup);
			assert(removedDeadUnit);
			//check if the group is now empty
//			L("AH: about to check if a group needs to be deleted entirely");
			int groupSize = it->Size();
			if (groupSize == 0) {
//				L("AH: yes, its ID is " << it->GetGroupID());
				attackGroups.erase(it);
			}
		} else if(attackGroupID == AIR_GROUP_ID) {
//			L("AH: unit destroyed and its in the air group, trying to remove");
			bool found_dead_unit_in_airUnits = false;
			for (list<int>::iterator it = airUnits.begin(); it != airUnits.end(); it++) {
				if (*it == unitID) {
					airUnits.erase(it);
					found_dead_unit_in_airUnits = true;
					break;
				}
			}
			assert(found_dead_unit_in_airUnits);
		}else {
			//its in stuckunits
			bool found_dead_in_stuck_units = false;
			list<pair<int,float3> >::iterator it;
			for (it = stuckUnits.begin(); it != stuckUnits.end(); it++) {
				if (it->first == unitID) {
					stuckUnits.erase(it);
					found_dead_in_stuck_units = true;
					break;
				}
			}
			assert(found_dead_in_stuck_units);
		}
		if (debug) L("AH: unit deletion done");
		ai->math->StopTimer(ai->ah->ah_timer_totalTime);
		ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
}

vector<float3>* CAttackHandler::GetKMeansBase()
{
		return &this->kMeansBase;
}

vector<float3>* CAttackHandler::GetKMeansEnemyBase()
{
		return &this->kMeansEnemyBase;
}

//if something is on this position, can it drive to any parts of our base?
bool CAttackHandler::CanTravelToBase(float3 pos) // add movetype
{
		ai->math->StartTimer(ai->ah->ah_timer_totalTime);
		ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, PATHTOUSE);
		bool res = ai->pather->PathExistsToAny(pos, *this->GetKMeansBase());
		ai->math->StopTimer(ai->ah->ah_timer_totalTime);
		return res;
}

//if something is on this position, can it drive to any parts of the enemy base?
bool CAttackHandler::CanTravelToEnemyBase(float3 pos) // add movetype
{
		ai->math->StartTimer(ai->ah->ah_timer_totalTime);
		ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, PATHTOUSE);
		bool res = ai->pather->PathExistsToAny(pos, *this->GetKMeansEnemyBase());
		if (debugDraw) {
			AIHCAddMapPoint amp;
```

```
		if (res) {
			amp.label = "CanTravelToEnemyBase";
		} else {
			amp.label = "NOT CanTravelToEnemyBase";
		}
		amp.pos = pos;
		ai->cb->HandleCommand(&amp);
	}
	ai->math->StopTimer(ai->ah->ah_timer_totalTime);
	return res;
}

bool CAttackHandler::PlaceIdleUnit(int unit)
{
	if (ai->cb->GetUnitDef(unit) != NULL) {
		float3 moo = FindUnsafeArea(ai->cb->GetUnitPos(unit));
		if (moo != ZEROVECTOR && moo != ERRORVECTOR) {
		ai->MyUnits[unit]->Move(moo);
		}
	}
	return false;
}

//returns a safe spot from k-means, adjacent to myPos, safety params are (0..1).
//this is going away or changing.
//change to: decide on the random float 0...1 first, then find it. waaaay easier.
float3 CAttackHandler::FindSafeSpot(float3 myPos, float minSafety, float maxSafety)
{
	//find a safe spot
	int startIndex = (minSafety * this->kMeansK);
	if (startIndex < 0) startIndex = 0;
	int endIndex = (maxSafety * this->kMeansK);
	if (endIndex < 0) startIndex = 0;
	if (startIndex > endIndex) startIndex = endIndex;
	if (kMeansK <= 1 || startIndex == endIndex) {
		if (startIndex >= kMeansK) startIndex = kMeansK-1;
		float3 pos = kMeansBase[startIndex] + float3((RANDINT%SAFE_SPOT_DISTANCE), 0,
(RANDINT%SAFE_SPOT_DISTANCE));
		pos.y = ai->cb->GetElevation(pos.x, pos.z);
		return pos;
	}
	assert(startIndex < endIndex);
	assert(startIndex < kMeansK);
	assert(endIndex <= kMeansK);
	//get a subset of the kmeans
	int size = endIndex - startIndex;
	vector<float3> subset;
	for(int i = startIndex; i < endIndex; i++) {
		assert ( i < kMeansK);
		subset.push_back(kMeansBase[i]);
	}
	//then find a position on one of the lines between those points (pather)
	int whichPath;
	if (subset.size() > 1) whichPath = RANDINT % (int)subset.size();
	else whichPath = 0;

	assert (whichPath < (int)subset.size());
	assert (subset.size() > 0);
	if (whichPath+1 < (int)subset.size() && subset[whichPath].distance2D(subset[whichPath+1])
> KMEANS_MINIMUM_LINE_LENGTH) {
	 vector<float3> posPath;
		ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, PATHTOUSE);
		float cost = ai->pather->PathToPos(&posPath, subset[whichPath],
subset[whichPath+1]);
		float3 res;
		if (cost > 0) {
			int whichPos = RANDINT % (int)posPath.size();
			res = posPath[whichPos];
```

```
		} else {
			res = subset[whichPath];
		}
		return res;
	} else {
		assert (whichPath < (int)subset.size());
		float3 res = subset[whichPath];
		return res;
	}
}

float3 CAttackHandler::FindSafeArea(float3 pos) {
	if(this->DistanceToBase(pos) < SAFE_SPOT_DISTANCE) return pos;
	float min = 0.6;
	float max = 0.95;
	float3 safe = this->FindSafeSpot(pos, min, max);
	//: this is a hack
	safe += pos;
	safe /= 2;
	return safe;
}

float3 CAttackHandler::FindVerySafeArea(float3 pos) {
	float min = 0.9;
	float max = 1.0;
	return this->FindSafeSpot(pos, min, max);
}

float3 CAttackHandler::FindUnsafeArea(float3 pos) {
	float min = 0.1;
	float max = 0.3;
	return this->FindSafeSpot(pos, min, max);
}

float CAttackHandler::DistanceToBase(float3 pos) {
	float closestDistance = FLT_MAX;
	for(int i = 0; i < this->kMeansK; i++) {
		float3 mean = this->kMeansBase[i];
		float distance = pos.distance2D(mean);
		closestDistance = min(distance, closestDistance);
	}
	return closestDistance;
}

float3 CAttackHandler::GetClosestBaseSpot(float3 pos) {
	float closestDistance = FLT_MAX;
	int index = 0;
	for(int i = 0; i < this->kMeansK; i++) {
		float3 mean = this->kMeansBase[i];
		float distance = pos.distance2D(mean);
		if (distance < closestDistance) {
			closestDistance = distance, closestDistance;
			index = i;
		}
	}
	return kMeansBase[index];
}

vector<float3> CAttackHandler::KMeansIteration(vector<float3> means, vector<float3>
unitPositions, int newK)
{
	assert(newK > 0 && means.size() > 0);
	int numUnits = unitPositions.size();
	//change the number of means according to newK
	int oldK = means.size();
	means.resize(newK);
	//add a new means, just use one of the positions
	float3 newMeansPosition = unitPositions[0];
	newMeansPosition.y = ai->cb->GetElevation(newMeansPosition.x, newMeansPosition.z) +
```

```
K_MEANS_ELEVATION;
	for (int i = oldK; i < newK; i++) {
		means[i] = newMeansPosition;
	}
	//check all positions and assign them to means, complexity n*k for one iteration
	vector<int> unitsClosestMeanID;
	unitsClosestMeanID.resize(numUnits, -1);
	vector<int> numUnitsAssignedToMean;
	numUnitsAssignedToMean.resize(newK, 0);
	for (int i = 0; i < numUnits; i++) {
		float3 unitPos = unitPositions.at(i);
		float closestDistance = FLT_MAX;
		int closestIndex = -1;
		for (int m = 0; m < newK; m++) {
			float3 mean = means[m];
			float distance = unitPos.distance2D(mean);
			if (distance < closestDistance) {
				closestDistance = distance;
				closestIndex = m;
			}
		}
		//position nr i is closest to the mean at closestIndex
		unitsClosestMeanID[i] = closestIndex;
		numUnitsAssignedToMean[closestIndex]++;
	}
	//change the means according to which positions are assigned to them
	//use meanAverage for indexes with 0 pos'es assigned
	//make a new means list
	vector<float3> newMeans;
	newMeans.resize(newK, float3(0,0,0));
	for(int i = 0; i < numUnits; i++) {
		int meanIndex = unitsClosestMeanID[i];
		float num = max(1, numUnitsAssignedToMean[meanIndex]); //dont want to divide by 0
		newMeans[meanIndex] += unitPositions[i] / num;
	}
	//do a check and see if there are any empty means //, and set the height.
	for(int i = 0; i < newK; i++) {
		//if a newmean is unchanged, set it to the new means pos instead of 0,0,0
		if (newMeans[i] == float3(0,0,0)) {
			newMeans[i] = newMeansPosition;
		} else {
			//get the proper elevation for the y coord
			newMeans[i].y = ai->cb->GetElevation(newMeans[i].x, newMeans[i].z) +
K_MEANS_ELEVATION;
		}
	}
	return newMeans;
}

//remove cloaked things (like perimeter cameras) so they wont be a defense priority
void CAttackHandler::UpdateKMeans() {
	const int arrowDuration = 30;
	{//want local variable definitions
		//get positions of all friendly units and put them in a vector (completed buildings
only)
		int numFriendlies = 0;
		vector<float3> friendlyPositions;
		int friendlies[MAXUNITS];
		numFriendlies = ai->cb->GetFriendlyUnits(friendlies);
		for (int i = 0; i < numFriendlies; i++) {
			int unit = friendlies[i];
			CUNIT* u = ai->MyUnits[unit];
			//its a building, it has hp, and its mine (0)
			if (this->UnitBuildingFilter(u->def()) && this->UnitReadyFilter(unit) &&
u->owner() == 0) {
				friendlyPositions.push_back(u->pos());
			}
		}
		//hack to make it at least 1 unit, should only happen when you have no base:
```

```
		if (friendlyPositions.size() < 1) {
			//it has to be a proper position, unless there are no proper positions.
			if (numFriendlies > 0 && ai->cb->GetUnitDef(friendlies[0]) && ai-
>MyUnits[friendlies[0]]->owner() == 0) friendlyPositions.push_back(ai->cb->GetUnitPos(friendlies
[0]));
			else friendlyPositions.push_back(float3(RANDINT % (ai->cb->GetMapWidth()*8),
1000, RANDINT % (ai->cb->GetMapHeight()*8))); //when everything is dead
		}
		//calculate a new K. change the formula to adjust max K, needs to be 1 minimum.
		this->kMeansK = min(KMEANS_BASE_MAX_K, 1.0f + sqrt((float)numFriendlies+0.01f));
		//iterate k-means algo over these positions and move the means
		this->kMeansBase = KMeansIteration(this->kMeansBase, friendlyPositions, this-
>kMeansK);
	}
	//update enemy position k-means
	//get positions of all enemy units and put them in a vector (completed buildings only) :
buildingslist from surveillance
	int numEnemies = 0;
	vector<float3> enemyPositions;
	const int* enemies = ai->sh->GetEnemiesList();
	numEnemies = ai->sh->GetNumberOfEnemies(); // ai->cheat->GetEnemyUnits(enemies);
	for (int i = 0; i < numEnemies; i++) {
		const UnitDef *ud = ai->cheat->GetUnitDef(enemies[i]);
		if (this->UnitBuildingFilter(ud)) { // && this->UnitReadyFilter(unit)) {
			enemyPositions.push_back(ai->cheat->GetUnitPos(enemies[i]));
		}
	}
	//hack to make it at least 1 unit, should only happen when you have no base:
	if (enemyPositions.size() < 1) {
		//it has to be a proper position, unless there are no proper positions.
		if (numEnemies > 0 && ai->cheat->GetUnitDef(enemies[0])) enemyPositions.push_
back(ai->cheat->GetUnitPos(enemies[0]));
		else enemyPositions.push_back(float3(RANDINT % (ai->cb->GetMapWidth()*8), 1000,
RANDINT % (ai->cb->GetMapHeight()*8))); //when everything is dead
	}
	//calculate a new K. change the formula to adjust max K, needs to be 1 minimum.
	this->kMeansEnemyK = min(KMEANS_ENEMY_MAX_K, 1.0f + sqrt((float)numEnemies+0.01f));
//		L("AttackHandler: doing k-means k:" << kMeansK << " numPositions=" <<
numFriendlies);
	//iterate k-means algo over these positions and move the means
	this->kMeansEnemyBase = KMeansIteration(this->kMeansEnemyBase, enemyPositions, this-
>kMeansEnemyK);
	//base k-means and enemy base k-means are updated.
	//approach: add up (max - distance) to enemies
	vector<float> proximity;
	proximity.resize(kMeansK, 0.0000001f);
	const float mapDiagonal = sqrt(pow((float)ai->cb->GetMapHeight()*8,2) + pow((float)ai->cb-
>GetMapWidth()*8,2) + 1.0f);

	for (int f = 0; f < kMeansK; f++) {
		for (int e = 0; e < kMeansEnemyK; e++) {
			proximity[f] += mapDiagonal - kMeansBase[f].distance2D(kMeansEnemyBase[e]);
		}
	}
	//sort kMeans by the proximity score
	float3 tempPos;
	float temp;
	for (int i = 1; i < kMeansK; i++) { //how many are completed
		for (int j = 0; j < i; j++) { //compare to / switch with
			if (proximity[i] > proximity[j]) { //switch
				tempPos = kMeansBase[i];
				kMeansBase[i] = kMeansBase[j];
				kMeansBase[j] = tempPos;
				temp = proximity[i];
				proximity[i] = proximity[j];
				proximity[j] = temp;
			}
		}
	}
```

```
	//okay, so now we have a kMeans list sorted by distance to enemies, 0 being risky and k
being safest.
	if (debugDraw) {
		int lineWidth = 75;
		//now, draw these means on the map
		for (int i = 0; i < kMeansK; i++) {
			int figureGroup = (RANDINT % 59549847);
//			ai->cb->CreateLineFigure(kMeansBase[i-1]+float3(0,100,0), kMeansBase[i]+floa
t3(0,100,0), lineWidth, 1, arrowDuration, figureGroup);
			ai->cb->CreateLineFigure(kMeansBase[i]+float3(0,300,0), kMeansBase[i]+float3
(0,50,0), lineWidth, 1, arrowDuration, figureGroup);
			ai->cb->SetFigureColor(figureGroup, 240, 32, 240, 0.4);
		}
	}
}

bool CAttackHandler::UnitGroundAttackFilter(int unit) {
	CUNIT u = *ai->MyUnits[unit];
	bool result = u.def() != NULL && u.def()->canmove && u.category() == CAT_G_ATTACK;
	return result;
}

bool CAttackHandler::UnitBuildingFilter(const UnitDef *ud) {
	bool result = ud != NULL && ud->speed <= 0;
	return result;
}

bool CAttackHandler::CanHandleThisUnit(int unit) {
	const UnitDef *ud = ai->cb->GetUnitDef(unit);
	bool result = false;
	if (ud != NULL) {
		result = ud->speed > 0 && !ud->builder;
	}
	return result;
}

bool CAttackHandler::UnitReadyFilter(int unit) {
	CUNIT u = *ai->MyUnits[unit];
	bool result = u.def() != NULL
			&& !ai->cb->UnitBeingBuilt(unit)
			&& ai->cb->GetUnitHealth(unit) > ai->cb->GetUnitMaxHealth(unit)*0.5f;
	return result;
}

void CAttackHandler::UpdateAir() {
	if (airUnits.size() == 0) return;
	/*
	if enemy is dead, attacking = false
	every blue moon, do an air raid on most valuable thingy
	*/
	assert(!(airIsAttacking && airTarget == -1));
	if (airIsAttacking &&  (airUnits.size() == 0 || ai->cheat->GetUnitDef(airTarget) == NULL))
{
		airTarget = -1;
		airIsAttacking = false;
	}
	if (ai->cb->GetCurrentFrame() % (60*30*5) == 0  //5 mins
			|| (ai->cb->GetCurrentFrame() % (30*30) == 0 && airUnits.size() > 8)) { //30
secs && 8+ units
		if (debug) L("AH: trying to attack with air units.");
		int numOfEnemies = ai->cheat->GetEnemyUnits(unitArray);
		int bestID = -1;
		float bestFound = -1.0;
		for (int i = 0; i < numOfEnemies; i++) {
			int enemy = unitArray[i];
			if (enemy != -1 && ai->cheat->GetUnitDef(enemy) != NULL && ai->cheat-
>GetUnitDef(enemy)->metalCost > bestFound) {
				bestID = enemy;
				bestFound = ai->cheat->GetUnitDef(enemy)->metalCost;
```

```
                }
            }
            if (debug) L("AH: selected the enemy: " << bestID);
            if (bestID != -1 && ai->cheat->GetUnitDef(bestID)) {
                //give the order
                for (list<int>::iterator it = airUnits.begin(); it != airUnits.end(); it++)
{
                    CUNIT* u = ai->MyUnits[*it];
                    u->Attack(bestID);
                }
                airIsAttacking = true;
                airTarget = bestID;
                ai->cb->SendTextMsg("AH: air group is attacking", 0);
            }
        }
        //units currently being built. (while the others are off attacking)
        if (ai->cb->GetCurrentFrame() % 1800 == 0) {
            airPatrolOrdersGiven = false;
        }
        if (!airPatrolOrdersGiven && !airIsAttacking) {
            if (debug) L("AH: updating air patrol routes");
            //get / make up some outer base perimeter points
            vector<float3> outerMeans;
            const int num = 3;
            outerMeans.reserve(num);
            if (kMeansK > 1) {
                int counter = 0;
                counter += kMeansK / 8; //offsetting the outermost one
                for (int i = 0; i < num; i++) {
                    outerMeans.push_back(kMeansBase[counter]);
                    if (counter < kMeansK-1) counter++;
                }
            } else {
                //theres just 1 kmeans and we need three
                for (int i = 0; i < num; i++) {
                    outerMeans.push_back(kMeansBase[0] + float3(250*i, 0, 0));
                }
            }
            assert(outerMeans.size() == num);
            //give the patrol orders to the outer means
            for (list<int>::iterator it = airUnits.begin(); it != airUnits.end(); it++) {
                CUNIT* u = ai->MyUnits[*it];
                u->Move(outerMeans[0] + float3(0,50,0)); //do this first in case theyre in
the enemy base
                for (int i = 0; i < num; i++) {
                    u->PatrolShift(outerMeans[i]);
                }
            }
            airPatrolOrdersGiven = true;
        }
}

void CAttackHandler::AssignTarget(CAttackGroup* group_in) {

        if (debug) L("AH: assign target to group " << group_in->GetGroupID());
        //do a target finding check
        //GET ENEMIES, FILTER THEM BY WHATS TAKEN, PATH TO THEM THEN GIVE THE PATH TO THE GROUP
        int numOfEnemies = ai->sh->GetNumberOfEnemies(); //ai->cheat->GetEnemyUnits(unitArray);
        const int* allEnemies = ai->sh->GetEnemiesList();
//      L("AH: initial num of enemies: "  << numOfEnemies);
        if (numOfEnemies) {
            //build vector of enemies
            static vector<int> allEligibleEnemies;
            allEligibleEnemies.reserve(numOfEnemies);
            allEligibleEnemies.clear();
            //make a vector with the positions of all enemies
            for (int i = 0; i < numOfEnemies; i++) {
                if(allEnemies[i] != -1) {
                    const UnitDef * ud = ai->cheat->GetUnitDef(allEnemies[i]);
```

```
                        //filter out cloaked and air enemies
                        if (ud != NULL && !ud->canfly
                                    && !(ud->canCloak
                                          //&& ud->startCloaked
                                          && ai->cb->GetUnitPos(allEnemies[i]) ==
ZEROVECTOR)) {
                                allEligibleEnemies.push_back(allEnemies[i]);
                        }
                  }
            }
//          L("AH: num of enemies of acceptable type: "  << allEligibleEnemies.size());
            static vector<int> availableEnemies;
            availableEnemies.reserve(allEligibleEnemies.size()); // its fewer though, really
            availableEnemies.clear();
            //now, filter out enemies currently in locations that existing groups are attacking
            //make a list of all assigned enemies:
            list<int> takenEnemies;
            for (list<CAttackGroup>::iterator groupIt = attackGroups.begin(); groupIt !=
attackGroups.end(); groupIt++) {
                  if (!groupIt->Defending() && groupIt->GetGroupID() != group_in->GetGroupID())
{ // when caching, fix this
                        list<int> theGroup = *groupIt->GetAssignedEnemies();
                        //list<int> hack = *theGroup;
                        takenEnemies.splice(takenEnemies.end(), theGroup);
//                      L("AH debug: added to takenEnemies, size is now " << takenEnemies.
size());
                  }
            }
            //put in availableEnemies those allEligibleEnemies that are not in takenEnemies
            for (vector<int>::iterator enemy = allEligibleEnemies.begin(); enemy !=
allEligibleEnemies.end(); enemy++) {
                  int enemyID = *enemy;
                  bool found = false;
                  for (list<int>::iterator it = takenEnemies.begin(); it != takenEnemies.
end(); it++) {
                        if (*it == enemyID) {
                              found = true;
                              break;
                        }
                  }
                  if (!found) availableEnemies.push_back(enemyID);
            }
//          L("AH: num of enemies that arent taken: "  << availableEnemies.size());
            // cache availableEnemies
            //make a list of the positions of these available enemies' positions
            static vector<float3> enemyPositions;
            enemyPositions.reserve(availableEnemies.size());
            enemyPositions.clear();
            for (vector<int>::iterator it = availableEnemies.begin(); it != availableEnemies.
end(); it++) {
                  //eligible target filtering has been done earlier
                  enemyPositions.push_back(ai->cheat->GetUnitPos(*it));
            }
//          L("AH: num of positions of enemies sent to pathfinder: " << enemyPositions.size());
            //find cheapest target
            static vector<float3> pathToTarget;
            pathToTarget.clear();
            float3 groupPos = group_in->GetGroupPos();
            //ai->pather->micropather->SetMapData(ai->pather->MoveArrays[group_in-
>GetWorstMoveType()],ai->tm->ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight);
//          float costToTarget = ai->pather->PathToSet(&pathToTarget, groupPos,
&enemyPositions);
            float myGroupDPS = group_in->DPS();
            float maxAcceptableThreat = myGroupDPS * ATTACK_MAX_THREAT_DIFFERENCE;
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
            ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, group_in->GetWorstMoveType());
            float costToTarget = ai->pather->PathToSetRadius(&pathToTarget, groupPos,
&enemyPositions, group_in->GetLowestAttackRange(), maxAcceptableThreat);
```

```
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
		if (pathToTarget.size() > 2) { //if it found something below max threat
			int lastIndex = pathToTarget.size()-1;
			float3 endPos = pathToTarget[lastIndex];//the position at the end
			//add move type of the threat map when support for that is added
			float targetThreat = ai->tm->ThreatAtThisPoint(endPos) - ai->tm->GetUnmodi-
fiedAverageThreat();
			L("AH: AssignTarget() group:" << group_in->GetGroupID() << " groupdps:"
<< myGroupDPS << " targetThreat:" << targetThreat << " num units:" << group_in->Size() <<" --
>ATTACK!");
			//ATTACK !
//			L("AH: during a power check, defense group is strong enough: attacking !");
			group_in->AssignTarget(pathToTarget, pathToTarget.back(), ATTACKED_AREA_
RADIUS);
		}//endif nopath
		else { //dont re-give the same order and so on
			// maybe combine groups here
			L("AH: AssignTarget() group:" << group_in->GetGroupID() << " groupdps:" <<
myGroupDPS << " num units:" << group_in->Size() <<" -->continue defending...");
			group_in->Defend();
		}

	}//endif noenemies
}

void CAttackHandler::AssignTargets() {
	int frameNr = ai->cb->GetCurrentFrame();
	//assign targets
//	if(frameNr % 600 == 0) {
		//L("AH debug: supposedly a full update cycle of targets");
//	}
	for(list<CAttackGroup>::iterator it = attackGroups.begin(); it != attackGroups.end();
it++) {
		CAttackGroup *group = &*it;
		if ((group->NeedsNewTarget() &&  frameNr % 60 == 0)) {//  || frameNr % 300 == 0) {
			AssignTarget(group);
		}
//		if (group->NeedsNewTarget()) {
//			ai->cb->SendTextMsg("AH: target no longer needed? o_O or no targets...", 0);
//		}
	}
}

void CAttackHandler::CombineGroups() {
	bool removedSomething = false;
	//pick a group A
	for (list<CAttackGroup>::iterator groupA = attackGroups.begin(); groupA != attackGroups.
end(); groupA++) {
		//if it is defending
		if (groupA->Defending()) {
			int groupAid = groupA->GetGroupID();
			float3 groupApos = groupA->GetGroupPos();
			unsigned groupAMoveType = groupA->GetWorstMoveType();
			//look for other groups that are defending.
			for (list<CAttackGroup>::iterator groupB = attackGroups.begin(); groupB !=
attackGroups.end(); groupB++) {
				//if they are close, combine.
				float3 groupBpos = groupB->GetGroupPos();
				int groupBid = groupB->GetGroupID();
				unsigned groupBMoveType = groupB->GetWorstMoveType();
				if (groupB->Defending()
						&& groupAid != groupBid
						&& groupApos.distance2D(groupBpos) < 1500
						&& groupAMoveType == groupBMoveType
						&& groupA->Size()+groupB->Size() < GROUP_MAX_NUM_UNITS)
{ // get a better pragmatic filter
					if (debug) L("AH:CombineGroups():: adding group " << groupB-
>GetGroupID() << " to group " << groupA->GetGroupID());
					//deeeeeeeeeeeeeestroooooooooooooy
```

```
                                    vector<int>* bUnits = groupB->GetAllUnits();
                                    for (vector<int>::iterator groupBunit = bUnits->begin();
groupBunit != bUnits->end(); groupBunit++) {
                                          groupA->AddUnit(*groupBunit);
                                    }
                                    this->attackGroups.erase(groupB);
                                    removedSomething = true;
                                    break;
                              }
                        }
                  }
                  if (removedSomething) break; //both for loops iterate the same list
            }

}

void CAttackHandler::Update()
{
ai->math->StartTimer(ai->ah->ah_timer_totalTime);
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
//     ai->math->TimerStart();
       int frameNr = ai->cb->GetCurrentFrame();
       if (frameNr < 2) UpdateKMeans();
       //set the map data here so i dont have to do it in each group or whatever
       // movement map PATHTOUSE = hack
       ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, PATHTOUSE);
       //update the k-means
       int frameSpread = 30; //frames between each update
       //calculate and draw k-means for the base perimeters
       if (frameNr % frameSpread == 0) {
             UpdateKMeans();
       }

       //check for stuck units in the groups
       if (frameNr % 30 == 0) {
             for (list<CAttackGroup>::iterator it = attackGroups.begin(); it != attackGroups.
end(); it++) {
                   int stuckUnit = it->PopStuckUnit();
                   if (stuckUnit != -1 && ai->cb->GetUnitDef(stuckUnit) != NULL) {
                         //this->AddUnit(stuckUnit);
                         pair<int, float3> foo;
                         foo.first = stuckUnit;
                         foo.second = ai->cb->GetUnitPos(stuckUnit);
                         stuckUnits.push_back(foo);
                         if (debug) L("AttackHandler: popped a stuck unit from attack group "
<< it->GetGroupID() << " and put it in the stuckUnits list which is now of size " << stuckUnits.
size());
                         ai->MyUnits[stuckUnit]->Stop();
                         ai->MyUnits[stuckUnit]->groupID = STUCK_GROUP_ID;
                   }
                   //check if the group is now empty
                   int groupSize = it->Size();
                   if (groupSize == 0) {
                         attackGroups.erase(it);
                         break;
                   }
             }
       }
       //attempt to combine groups that are defending and too weak to attack anything
       if (frameNr % 30 == 0) {
             CombineGroups();
       }
       //check if we have any new units, add them to a group.
       if (frameNr % 30 == 0 && units.size() > 0) {
//           L("AH: add units to group");
             for (list<int>::iterator unitIt = units.begin(); unitIt != units.end(); unitIt++) {
                   int unit = *unitIt;
                   const UnitDef* ud = ai->cb->GetUnitDef(unit);
```

```
                    if (ud != NULL) {
                            unsigned unitMoveType = ai->MyUnits[unit]->GetMoveType();//ai->ut-
>unittypearray[ud->id].moveSlopeType | ai->ut->unittypearray[ud->id].moveDepthType;
                            // add the water thing and maybe crush thing
                            if (debug) L("CAttackHandler::Update() adding unit:" << unit << "
type:" << ud->humanName << " unitMoveType:" << unitMoveType);
                            CAttackGroup* existingGroup = NULL;
                            //find a defending group:
                            const int JOIN_GROUP_DISTANCE = 300;
                            const float MAX_SPEED_DIFFERENCE_TO_JOIN = 1.5;
                            for (list<CAttackGroup>::iterator it = attackGroups.begin(); it !=
attackGroups.end(); it++) {
                                    if (it->Size() < GROUP_MAX_NUM_UNITS
                                                    && it->Defending()
                                                    && this->DistanceToBase(it->GetGroupPos()) <
JOIN_GROUP_DISTANCE
                                                    && ai->MyUnits[unit]->def()->speed <= it-
>GetHighestUnitSpeed() * MAX_SPEED_DIFFERENCE_TO_JOIN
                                                    && ai->MyUnits[unit]->def()->speed * MAX_SPEED_
DIFFERENCE_TO_JOIN >= it->GetHighestUnitSpeed()
                                                    && it->GetWorstMoveType() == unitMoveType) {
                                            existingGroup = &*it;
                                            break;
                                    }
                            }
                            if (existingGroup != NULL) {
                                    if (debug) L("CAttackHandler::Update() adding unit to existing
group:" << existingGroup->GetGroupID() << " num groups:" << attackGroups.size() << " added
unit:" << unit << " type:" << ai->cb->GetUnitDef(unit)->humanName);
                                    existingGroup->AddUnit(unit);
                            } else {
                                    if (debug) L("CAttackHandler::Update() creating new group
for unit, num groups:" << attackGroups.size() << " added unit:" << unit << " type:" << ai->cb-
>GetUnitDef(unit)->humanName);
                                    //we dont have a good group, make one
                                    newGroupID++;
            //                      L("creating new att group: group " << newGroupID);
                                    CAttackGroup newGroup(ai, newGroupID);
                                    //newGroup.defending = true;
                                    newGroup.AddUnit(unit);
            //                                      L("added unit:" << unit << " type:" << ai->cb-
>GetUnitDef(unit)->humanName);
                                    attackGroups.push_back(newGroup);
                            }
                    }
            }
            units.clear();
    }
    //update all the air units:
    UpdateAir();
    //basic attack group formation from defense units:
    this->AssignTargets();
    //update current groups
    int counter = 0;
    //ai->cb->GetSelectedUnits()

    //adding extremely basic cyborg play support: (cyborg = human and AI sharing unit control)
    int numSelectedUnits = ai->cb->GetSelectedUnits(unitArray);
    for (list<CAttackGroup>::iterator it = attackGroups.begin(); it != attackGroups.end();
it++) {
            int thisGroupID = it->GetGroupID();
            bool thisGroupTaken = false;
            //cyborg style commented out while testing
/*          for (int i = 0; i < numSelectedUnits; i++) {
                    if (ai->cb->GetUnitDef(unitArray[i]) && ai->MyUnits[unitArray[i]]->groupID
== thisGroupID) {
                            thisGroupTaken = true;
                            L("AH::Update for group " << thisGroupID << " frame:" << frameNr << "
unit in group taken by player");
```

```
                        break;
                    }
                }
*/
                if (!thisGroupTaken) it->Update();
        }
        //print out the totals
        frameSpread = 7200; //frames between each update
        if (debug && ai->cb->GetCurrentFrame() % frameSpread == 0) {
                L("--------------------------------------------------------------------------");
                L("AttackHandler: writing out the number of total units");
                int sum = 0;
                float powerSum = 0;
                for (list<CAttackGroup>::iterator it = attackGroups.begin(); it != attackGroups.
end(); it++) {
                        int size = it->Size();
                        sum += size;
                        L("an attack group had " << size << " units and DPS:" << it->DPS() << ":");
                        powerSum += it->DPS();
                        it->Log();
                }
                float airPower = 0;
                int counter_ = 0;
                for (list<int>::iterator it = airUnits.begin(); it != airUnits.end(); it++) {
                        airPower += ai->ut->unittypearray[ai->MyUnits[*it]->def()->id].AverageDPS;
                        L("" << counter++ << ":" << ai->MyUnits[*it]->def()->humanName);
                }
                L("airUnits: " << airUnits.size() << " units and dps:" << airPower);
                sum += airUnits.size();
                powerSum += airPower;
                L("total units: " << sum << " total dps:" << powerSum);
                L("in inactive group: " << units.size());
                L("in stuck group: " << stuckUnits.size());
                L("--------------------------------------------------------------------------");
        }
ai->math->StopTimer(ai->ah->ah_timer_totalTime);
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
}
```

CAttackGroup.h:

This is the code managing the detailed actions of groups of units, as groups. Planned paths for attacking are given by CAttackHandler to each attack group, to be executed by the attack groups.

```
#pragma once
#include "GlobalAI.h"


class CAttackGroup
{
        public:
                CAttackGroup(AIClasses* ai, int groupID_in);

        virtual ~CAttackGroup();

        //update-related functions:
        void CAttackGroup::MicroUpdate(float3 groupPosition);
        void CAttackGroup::MoveOrderUpdate(float3 groupPosition);
        void CAttackGroup::StuckUnitFix();
        void CAttackGroup::Update();

        //Set/Get / Add/Remove
        bool CAttackGroup::Defending();
        int CAttackGroup::Size();
        int CAttackGroup::GetGroupID();
        float3 CAttackGroup::GetGroupPos();
        int CAttackGroup::PopStuckUnit();
        float CAttackGroup::DPS();
        void CAttackGroup::Log();
```

```
	unsigned CAttackGroup::GetWorstMoveType();
	float CAttackGroup::GetLowestUnitSpeed();
	float CAttackGroup::GetHighestUnitSpeed();
	float CAttackGroup::GetLowestAttackRange();
	float CAttackGroup::GetHighestAttackRange();
	vector<int>* CAttackGroup::GetAllUnits(); // for combining groups
	list<int>* CAttackGroup::GetAssignedEnemies();
	void CAttackGroup::AddUnit(int unitID);
	bool CAttackGroup::RemoveUnit(int unitID);
	int CAttackGroup::PopDamagedUnit();

	//attack order related functions, used from AH:
	void CAttackGroup::ClearTarget();
	bool CAttackGroup::NeedsNewTarget();
	void CAttackGroup::AssignTarget(vector<float3> path, float3 target, float radius);
	void CAttackGroup::Flee();
	void CAttackGroup::Defend();

	private:
	void CAttackGroup::DrawGroupPosition();
	bool CAttackGroup::FindDefenseTarget(float3 groupPosition);
	void CAttackGroup::PathToBase(float3 groupPosition);
	bool CAttackGroup::CloakedFix(int enemy);
	void CAttackGroup::RecalcGroupProperties();
	bool CAttackGroup::ConfirmPathToTargetIsOK();

	AIClasses *ai;
	vector<int> units;
	float lowestAttackRange;
	float highestAttackRange;
	float lowestUnitSpeed;
	float highestUnitSpeed;
	float groupPhysicalSize;//close to the size of the radius of the unit
	float highestTurnRadius;

	unsigned worstMoveType;

	float3 attackPosition;
	float attackRadius;

	vector<float3> pathToTarget;
	int pathIterator;

	//static:
	int unitArray[MAXUNITS];

	int groupID;
	//states:
	bool isMoving;
	bool isFleeing;
	bool isDefending;
	bool isShooting;

	float groupDPS;
	void CAttackGroup::RecalcDPS();

	float3 groupPos;
	void CAttackGroup::RecalcGroupPos();
	int lastGroupPosUpdateFrame;

	float3 baseReference;
	int lastBaseReferenceUpdate;

	void CAttackGroup::RecalcAssignedEnemies();
	list<int> assignedEnemies;
};
```

CAttackGroup.cpp:

```
#include "AttackGroup.h"

//not moving for 5*60 frames = stuck :
#define UNIT_STUCK_COUNTER_MANEUVER_LIMIT 60
//gives it (STUCK_COUNTER_LIMIT - STUCK_COUNTER_MANEUVER_LIMIT)*60 seconds
// to move longer than UNIT_STUCK_MOVE_DISTANCE (7 sec)
#define UNIT_STUCK_COUNTER_LIMIT 120
//stuck maneuver distance, how far it moves when trying to get unstuck. has to be higher than
minimum pather resolution
#define UNIT_STUCK_MANEUVER_DISTANCE 250
//the max amount of difference in height there may be at the position to maneuver to (dont
retreat into a hole)
#define UNIT_MAX_MANEUVER_HEIGHT_DIFFERENCE_UP 20
//minimum of offset between my maxrange and the enemys position before considering moving
#define UNIT_MIN_MANEUVER_RANGE_DELTA (THREATRES*8)
//minimum amount to maneuver when getting to max range
#define UNIT_MIN_MANEUVER_DISTANCE (THREATRES*4)
//second requirement, minimum percentage of my range to move when getting to max range (to stop
slow long range from dancing around so much that they cant fire)
#define UNIT_MIN_MANEUVER_RANGE_PERCENTAGE 0.2f
//minimum time to maneuver (frames), used for setting maneuvercounter (in case the speed/dist
formula is weird)
#define UNIT_MIN_MANEUVER_TIME 15
#define UNIT_DESTINATION_SLACK (THREATRES*4.0f*1.4f)
#define GROUP_DESTINATION_SLACK THREATRES*8
#define GROUP_MEDIAN_UNIT_SELECTION_SLACK 10.0f
#define MINIMUM_GROUP_RETREAT_DISTANCE 400
#define BASE_REFERENCE_UPDATE_FRAMES 1800
#define FLEE_MAX_THREAT_DIFFERENCE 1.15 //1.5
#define FLEE_MIN_PATH_RECALC_SECONDS 3
#define MIN_DEFENSE_RADIUS 100
#define DAMAGED_UNIT_FACTOR 0.50f

CAttackGroup::CAttackGroup(AIClasses* ai, int groupID_in)
{
	this->ai=ai;
	this->groupID = groupID_in;
	this->pathIterator = 0;
	this->lowestAttackRange = 100000;
	this->highestAttackRange = 1;
//	this->movementCounterForStuckChecking = 0;
	this->isDefending = true;
	this->isMoving = false;
	this->isShooting = false;
	this->attackPosition = float3(0,0,0);
	this->attackRadius = 1;
	this->isFleeing = false;
	this->groupPos = ZEROVECTOR;
	this->lastGroupPosUpdateFrame = -1;
	this->baseReference = ERRORVECTOR;
	this->lastBaseReferenceUpdate = -BASE_REFERENCE_UPDATE_FRAMES-1;
	this->worstMoveType = 0xFFFFFFFF;
	this->groupPhysicalSize = 1.0f;
}

CAttackGroup::~CAttackGroup()
{
//class has no dynamically allocated objects.
}

void CAttackGroup::Log() {
	int temp = 0;
	if (ai->ah->debug) L("AG: logging contents of group " << groupID << ":");
	for (vector<int>::iterator it = units.begin(); it != units.end(); it++) {
		temp++;
		if (ai->ah->debug) {
			if (ai->cb->GetUnitDef(*it)) {
```

```
                        L("" << temp << ": " << *it << " type:" << ai->cb->GetUnitDef(*it)-
>humanName);
                } else {
                        L("" << temp << ": " << *it << " ILLEGAL UNIT - has no unit def");
                }
        }
    }
}

void CAttackGroup::AddUnit(int unitID)
{
    if (ai->cb->GetUnitDef(unitID)) {
        //add to my structure
        units.push_back(unitID);
        //set its group ID:
        ai->MyUnits[unitID]->groupID = this->groupID;
        //update the attack range properties of this group
        //this->lowestAttackRange = min(this->lowestAttackRange, this->ai->ut-
>GetMaxRange(ai->cb->GetUnitDef(unitID)));
        //this->highestAttackRange = max(this->highestAttackRange, this->ai->ut-
>GetMaxRange(ai->cb->GetUnitDef(unitID)));
    } else {
        bool dead_unit_added_to_group = false;
        assert(dead_unit_added_to_group);
    }
    this->RecalcGroupProperties();
}

bool CAttackGroup::RemoveUnit(int unitID) {
    bool found = false;
    vector<int>::iterator it;
    for (it = units.begin(); it != units.end(); it++) {
        if (*it == unitID) {
            found = true;
//                  L("AttackGroup: erasing unit with id:" << unitID);
            break;
        }
    }
    if (found) {
        units.erase(it);
        if( ai->cb->GetUnitDef(unitID) != NULL) {
            ai->MyUnits[unitID]->groupID = 0;
//                  L("AttackGroup: groupid = 0 --> success");
        }
    }
    assert(found);
    this->RecalcGroupProperties();
    return found;
}

int CAttackGroup::Size()
{
    return units.size();
}

int CAttackGroup::GetGroupID() {
    return groupID;
}

unsigned CAttackGroup::GetWorstMoveType() {
    return worstMoveType;
}

vector<int>* CAttackGroup::GetAllUnits() {
    return &this->units;
}

bool CAttackGroup::Defending() {
    return isDefending;
```

```
}

void CAttackGroup::RecalcGroupProperties() {
	this->worstMoveType = 0xFFFFFFFF;
	this->lowestAttackRange = 10000.0f;
	this->highestAttackRange = 1.0f;
	this->lowestUnitSpeed = 10000.0f;
	this->highestUnitSpeed = 1.0f;
	this->groupPhysicalSize = 1.0f;
	this->groupDPS = 0.00001f;
	this->highestTurnRadius = 0.00001f;
	if (units.size() == 0) return;
	const UnitDef* ud;
	for (vector<int>::iterator it = units.begin(); it != units.end(); it++) {
		int unitID = *it;
		ud = ai->cb->GetUnitDef(unitID);
		if(ud != NULL) {
			worstMoveType &= ai->MyUnits[unitID]->GetMoveType();
			this->lowestAttackRange = min(this->lowestAttackRange, this->ai->ut-
>GetMinRange(ud));
			this->highestAttackRange = max(this->highestAttackRange, this->ai->ut-
>GetMaxRange(ud));
			this->lowestUnitSpeed = min(this->lowestUnitSpeed, ud->speed);
			this->highestUnitSpeed = max(this->highestUnitSpeed, ud->speed);
			this->groupPhysicalSize += (ud->xsize+ud->ysize) * 0.3f;
			this->groupDPS += ai->MyUnits[unitID]->GetAverageDPS();
			if(ud->canfly)this->highestTurnRadius = max(this->highestTurnRadius, ud-
>turnRadius);
		}
	}
	if (ai->ah->debug) L("CAttackGroup::RecalcGroupProperties() worstMoveType:"<<worstMove
Type<<" lowestAttackRange:"<<lowestAttackRange<<" highestAttackRange:"<<highestAttackRange<<"
lowestUnitSpeed:"<<lowestUnitSpeed<<" highestUnitSpeed:"<<highestUnitSpeed<<" groupPhysicalSize:
"<<groupPhysicalSize<<" groupDPS:"<<groupDPS<<" highestTurnRadius:"<<highestTurnRadius);
}


//combined ut-dps of the group
float CAttackGroup::DPS() {
	return this->groupDPS;
}

float CAttackGroup::GetLowestUnitSpeed() {
	return this->lowestUnitSpeed;
}

float CAttackGroup::GetHighestUnitSpeed() {
	return this->highestUnitSpeed;
}

int CAttackGroup::PopStuckUnit() {
	//removes a stuck unit from the group if there is one, and puts a marker on the map
	for (vector<int>::iterator it = units.begin(); it != units.end(); it++) {
		if (ai->MyUnits[*it]->stuckCounter > UNIT_STUCK_COUNTER_LIMIT && ai->MyUnits[*it]-
>attemptedUnstuck) {
			int id = *it;
			//mark it:
			if (ai->ah->debugDraw) {
				char text[512];
				sprintf(text, "stuck %i:%i, dropping from group:%i. isMoving=%i", id,
ai->MyUnits[*it]->stuckCounter, groupID, isMoving);
				AIHCAddMapPoint amp;
				amp.label = text;
				amp.pos = ai->cb->GetUnitPos(id);
				ai->cb->HandleCommand(&amp);

				sprintf(text, "humanName:%s", ai->MyUnits[*it]->def()->humanName.
c_str());
				amp.label = text;
```

```
                        amp.pos = ai->cb->GetUnitPos(id) + float3(0, 0, 30);
                        ai->cb->HandleCommand(&amp);
                }
                ai->MyUnits[id]->stuckCounter = 0;
                //units.erase(it);
                this->RemoveUnit(id);
                return id;
            }
        }
        return -1;
}

int CAttackGroup::PopDamagedUnit() {
        for (vector<int>::iterator it = units.begin(); it != units.end(); it++) {
            if (ai->MyUnits[*it]->Health() < ai->MyUnits[*it]->MaxHealth()*DAMAGED_UNIT_FACTOR)
{
                this->RemoveUnit(*it);
           return *it;
            }
        }
        return -1;
}

bool CAttackGroup::CloakedFix(int enemy) {
        const UnitDef * ud = ai->cheat->GetUnitDef(enemy);
//      return ud != NULL && !(ud->canCloak && ud->startCloaked && (ai->cb->GetUnitPos(enemy) ==
ZEROVECTOR));
        return ud != NULL && !(ud->canCloak && ai->cb->GetUnitPos(enemy) == ZEROVECTOR);
}

void CAttackGroup::RecalcGroupPos() {
        //whats the groups position (for distance checking when selecting targets)
        int unitCounter = 0;
        float3 newGroupPosition = float3(0,0,0);
        int numUnits = units.size();
        for (int i = 0; i < numUnits; i++) {
            int unit = units[i];
            if (ai->cb->GetUnitDef(unit) != NULL) {
                unitCounter++;
                newGroupPosition += ai->cb->GetUnitPos(unit);
            }
        }
        if (unitCounter > 0) {
            newGroupPosition /= unitCounter;
            //find the unit closest to the center (since the actual center might be on a hill
or something)
            float closestSoFar = FLT_MAX;
            int closestUnitID = -1;
            float temp;
            int unit;
            for (int i = 0; i < numUnits; i++) {
                unit = units[i];
                //is it closer. consider also low unit counts, then the first will be used
since its < and not <=, assuming sufficient float accuracy
                if (ai->cb->GetUnitDef(unit) != NULL && (temp = newGroupPosition.
distance2D(ai->cb->GetUnitPos(unit))) < closestSoFar-GROUP_MEDIAN_UNIT_SELECTION_SLACK) {
                        closestSoFar = temp;
                        closestUnitID = unit;
                }
            }
            assert(closestUnitID != -1);
            newGroupPosition = ai->cb->GetUnitPos(closestUnitID);
        } else {
            if (ai->ah->debug) L("AttackGroup: empty attack group when calcing group pos!");
            newGroupPosition = ERRORVECTOR; //<-----------------

        }
        this->groupPos = newGroupPosition;
        this->lastGroupPosUpdateFrame = ai->cb->GetCurrentFrame();
```

```
}

float3 CAttackGroup::GetGroupPos() {
	if (this->lastGroupPosUpdateFrame < ai->cb->GetCurrentFrame()) {
		this->RecalcGroupPos();
	}
	return this->groupPos;
}

float CAttackGroup::GetLowestAttackRange() {
	return this->lowestAttackRange;
}
float CAttackGroup::GetHighestAttackRange() {
	return this->highestAttackRange;
}

void CAttackGroup::DrawGroupPosition() { //debug-filtered at call
	//draw arrow pointing to the group center:
	int figureID = groupID + 1700;
	float3 groupPosition = GetGroupPos();
	if (!isDefending) { //att
		ai->cb->SetFigureColor(figureID, 65535.0f, 0.0f, 0.0f, 0.2f);
		ai->cb->CreateLineFigure(groupPosition + float3(0,140,0), groupPosition + float3(0,
20,0),48,1,300,figureID);
		ai->cb->SetFigureColor(figureID, 65535.0f, 0.0f, 0.0f, 0.2f);
	} else if(!isFleeing) { //def but not flee
		ai->cb->SetFigureColor(figureID, 0.0f, 65535.0f, 0.0f, 0.2f);
		ai->cb->CreateLineFigure(groupPosition + float3(0,80,0), groupPosition + float3(0,2
0,0),96,1,300,figureID);
		ai->cb->SetFigureColor(figureID, 0.0f, 65535.0f, 0.0f, 0.2f);
	} else {//flee
		ai->cb->SetFigureColor(figureID, 65535.0f, 65535.0f, 0.0f, 0.2f);
		ai->cb->CreateLineFigure(groupPosition + float3(0,60,0), groupPosition + float3(0,2
0,0),220,0,300,figureID);
		ai->cb->SetFigureColor(figureID, 65535.0f, 65535.0f, 0.0f, 0.2f);
	}
}

//returns enemies in my attack area
list<int>* CAttackGroup::GetAssignedEnemies() {
	bool why_would_you_ask_if_a_defending_group_has_assigned_enemies = !isDefending;
	assert(why_would_you_ask_if_a_defending_group_has_assigned_enemies);
	return &assignedEnemies;
}

//Note: trace enemies..
void CAttackGroup::RecalcAssignedEnemies() {
	assignedEnemies.clear();
	if (!isDefending) {
		int numTaken = ai->cheat->GetEnemyUnits(unitArray, attackPosition, attackRadius);
		for(int i = 0; i < numTaken; i++) {
			int takenEnemy = unitArray[i];
			assignedEnemies.push_back(unitArray[i]);
		}
	}
}

//the function which the attack handler uses to give an attack path and area (point+size)
void CAttackGroup::AssignTarget(vector<float3> in_path, float3 in_position, float in_radius) {
	this->attackPosition = in_position;
	this->attackRadius = in_radius;
	this->pathToTarget = in_path;
	this->isMoving = true;
	this->isShooting = false;
	this->pathIterator = 0;
	this->isDefending = false;
	this->isFleeing = false;
	if (ai->ah->debugDraw) {
		int figureID = (groupID+4200);
```

```
		ai->cb->SetFigureColor(figureID, 65535.0f, 0.0f, 0.0f, 0.2f);
		for (int i = 1; i < (int)pathToTarget.size(); i++) {
			ai->cb->CreateLineFigure(pathToTarget[i-1] + float3(0,50,0), pathToTarget[i]
+ float3(0,50,0), 8, 0, 300, figureID);
		}
		ai->cb->SetFigureColor(figureID, 65535.0f, 0.0f, 0.0f, 0.2f);
	}
//	L("AG: target assigning complete. path size: " << pathToTarget.size() << " targetx:"
<< attackPosition.x << " targety:" << attackPosition.y << " radius:" << attackRadius << "
isMoving:" << isMoving << " frame:" << ai->cb->GetCurrentFrame() << " groupID:" << groupID);
	RecalcAssignedEnemies();
}

void CAttackGroup::ClearTarget() {
	this->isMoving = false;
	this->isFleeing = false;
	this->isDefending = true;
	this->attackPosition = ZEROVECTOR;
	this->attackRadius = 0.0f;
	this->pathToTarget.clear();
	this->isShooting = false;
}

void CAttackGroup::Defend() {
ai->math->StartTimer(ai->ah->ah_timer_Defend);
	this->ClearTarget();
	if (!this->FindDefenseTarget(GetGroupPos())) {
		if (ai->ah->debug) L("AG: Defend() Couldnt find defense target !!");
		this->PathToBase(GetGroupPos());
	}
ai->math->StopTimer(ai->ah->ah_timer_Defend);
}

void CAttackGroup::Flee() {
	//ignore if we are close to base
	this->ClearTarget();
	this->isFleeing = true;
	this->PathToBase(this->GetGroupPos());
}

bool CAttackGroup::NeedsNewTarget() {
ai->math->StartTimer(ai->ah->ah_timer_NeedsNewTarget);
	bool needs = false;
	//if we arent doing anything
	if (!isShooting) {
		if (!isMoving) {
ai->math->StopTimer(ai->ah->ah_timer_NeedsNewTarget);
			return true;
		}
		//if we're supposed to have a target but theres no enemies in the target area
		if (!isFleeing && isMoving && ai->cheat->GetEnemyUnits(this->unitArray,
pathToTarget.back(), max(highestAttackRange, max(this->attackRadius, MIN_DEFENSE_RADIUS))) ==
0){
ai->math->StopTimer(ai->ah->ah_timer_NeedsNewTarget);
			return true;
		}
		if (isFleeing && groupDPS > ai->tm->ThreatAtThisPoint(groupPos) + ai->tm->GetUn-
modifiedAverageThreat()) {
ai->math->StopTimer(ai->ah->ah_timer_NeedsNewTarget);
			return true;
		}
		//if we're not fighting but theres nowwhere to go, OR the path has too high threat
somewhere
		if (!ConfirmPathToTargetIsOK()) {// all states?
ai->math->StopTimer(ai->ah->ah_timer_NeedsNewTarget);
			return true;
		}
	}
ai->math->StopTimer(ai->ah->ah_timer_NeedsNewTarget);
```

```
	return needs;
}

bool CAttackGroup::ConfirmPathToTargetIsOK()
{
	float threatLimit = (groupDPS * FLEE_MAX_THREAT_DIFFERENCE) + ai->tm->GetUnmodifiedAverag-
eThreat();
	int stopPoint = (int)pathToTarget.size();//it should only path to lowest range anyway
- (lowestAttackRange/THREATRES);
	int startPoint = this->pathIterator;
	if (isFleeing) {
		startPoint += ceil((lowestUnitSpeed*FLEE_MIN_PATH_RECALC_SECONDS)/THREATRES);
//		L(" CAttackGroup::ConfirmPathToTargetIsOK startpoint: " << startPoint);
	}
	for (int i = startPoint; i < stopPoint; i++) {
		if (ai->tm->ThreatAtThisPoint(pathToTarget[i]) > threatLimit) {
			return false;
		}
	}
	return true;
}

void CAttackGroup::PathToBase(float3 groupPosition)
{
//	L("AG: pathing back to base. group:" << groupID);
    pathToTarget.clear();
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
	ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, this->GetWorstMoveType());
	float costToTarget = ai->pather->PathToSet(&pathToTarget, groupPosition, ai->ah-
>GetKMeansBase());
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
	if (pathToTarget.size() <= 2) {
		//isMoving = false;
		this->ClearTarget();
	} else {
		isMoving = true;
		isShooting = false;
		this->pathIterator = 0;
		if (ai->ah->debugDraw) {
			int figureID = (groupID+600);
			ai->cb->SetFigureColor(figureID, 65535.0f, 65535.0f, 0.0f, 0.2f);
			for (int i = 1; i < (int)pathToTarget.size(); i++) {
				ai->cb->CreateLineFigure(pathToTarget[i-1] + float3(0,50,0),
pathToTarget[i] + float3(0,50,0), 8, 0, 300, figureID);
			}
			ai->cb->SetFigureColor(figureID, 65535.0f, 65535.0f, 0.0f, 0.2f);
		}
	}
}

bool CAttackGroup::FindDefenseTarget(float3 groupPosition)
{
	//ai->cb->SendTextMsg("AG: FindDefenseTarget - group is defending and selecting a
target.", 0);
	int frameNr = ai->cb->GetCurrentFrame();
	//the "find new enemy" part
	int numOfEnemies;
	numOfEnemies = ai->sh->GetNumberOfEnemies();
	const int* enemyArray = ai->sh->GetEnemiesList();
	if (numOfEnemies > 0) {
		//build vector of enemies
		static vector<float3> enemyPositions;
		enemyPositions.reserve(numOfEnemies);
		enemyPositions.clear();
		//make a vector with the positions of all enemies
		for (int i = 0; i < numOfEnemies; i++) {
			if (enemyArray[i] != -1) {
				const UnitDef *enemy_ud = ai->cheat->GetUnitDef(enemyArray[i]);
```

```
                        float3 enemyPos = ai->cheat->GetUnitPos(enemyArray[i]);
                        //do some filtering and then
                        if (ai->cb->GetUnitDef(enemyArray[i]) != NULL
                                && this->CloakedFix(enemyArray[i])
                                && !enemy_ud->canfly
                                && !enemy_ud->weapons.size() == 0) {
                                //pathfinder removes units not reachable by my unit type/
position
                                enemyPositions.push_back(enemyPos);
                        }
                }
        }
        // if all units are cloaked or otherwise not eligible (but there are units) then
target them anyway
        if (enemyPositions.size() < 1) {
                for (int i = 0; i < numOfEnemies; i++) {
                        if (enemyArray[i] != -1) {
                                const UnitDef *enemy_ud = ai->cheat->GetUnitDef(enemyArray[i]);
                                float3 enemyPos = ai->cheat->GetUnitPos(enemyArray[i]);
                                enemyPositions.push_back(enemyPos);
                        }
                }
        }

        //now it becomes sexy :p
        static vector<float3> tempPathToTarget;
        tempPathToTarget.clear();
        //verify the validity of the current base reference point
        if (this->lastBaseReferenceUpdate < (frameNr - BASE_REFERENCE_UPDATE_FRAMES)) {
                //find closest reachable base spot
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
                ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, this->GetWorstMoveType());
                float cost = ai->pather->PathToPrioritySet(&tempPathToTarget, groupPosition,
ai->ah->GetKMeansBase());
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
                if (cost != 0) { //tempPathToTarget.size() >= 2) {
//                      L("AG: FindDefTarget: base spot to search from found");
                        //searchStartPosition = tempPathToTarget.back();
                        baseReference = tempPathToTarget.back();
                } else { //no base spot found :o
                        //searchStartPosition = groupPosition;
                        baseReference = ERRORVECTOR;
                        if (ai->ah->debug) L("AG: no base reference found for group " <<
groupID);
                }
                lastBaseReferenceUpdate = frameNr;
        }

        float3 searchStartPosition;
        if (baseReference != ERRORVECTOR) searchStartPosition = baseReference;
        else searchStartPosition = groupPosition;

        //search for enemy from closest reachable base spot
        tempPathToTarget.clear();
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
        ai->pather->micropather->SetMapData(ai->pather->canMoveIntMaskArray,ai->tm-
>EmptyThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, this->GetWorstMoveType());
        float costToTarget = ai->pather->PathToSet(&tempPathToTarget, searchStartPosition,
&enemyPositions);
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
        if (tempPathToTarget.size() <= 2) { //if theres no enemies from either your pos or
closest base spot pos
                isMoving = false;
//                      L("AG: FindNewTarget sets isMoving to false.");
        } else {
//                      L("AG: FindDefTarget: we selected the enemy which is closest to the base");
                float3 selectedEnemyPosition = tempPathToTarget.back();
/*
```

```
                    AIHCAddMapLine aml;
                    aml.posfrom = groupPosition;
                    aml.posto = selectedEnemyPosition;
                    ai->cb->HandleCommand(&aml);
*/
                    //lets make the actual path

                    float costToEnemy;
                    //only calc new path if it will actually be different (if searchStartPosition
is different from in previous search
                    if (searchStartPosition == groupPosition) {
                costToEnemy = costToTarget;
                    } else {
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
                            ai->pather->micropather->SetMapData(ai->pather-
>canMoveIntMaskArray,ai->tm->EmptyThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight,
this->GetWorstMoveType());
                            costToEnemy = ai->pather->PathToPos(&tempPathToTarget, groupPosition,
selectedEnemyPosition);
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
                    }
                    //if tempPathToTarget is a valid attack path
                    if (costToEnemy > 0 && tempPathToTarget.size() > 2) { //we have found
cheapest way to the enemy which is closest to base (or us if theres no reachable base)
                            this->pathToTarget = tempPathToTarget;
                            isMoving = true;
                            isFleeing = false;
                            this->pathIterator = 0;
                            //draw the path
                            if (ai->ah->debugDraw) {
                                    int figureID = (groupID+2700);
                                    ai->cb->SetFigureColor(figureID, 0.0f, 65535.0f, 0.0f, 0.2f);
                                    for (int i = 1; i < (int)pathToTarget.size(); i++) {
                                            ai->cb->CreateLineFigure(pathToTarget[i-1] +
float3(0,50,0), pathToTarget[i] + float3(0,50,0), 8, 0, 300, figureID);
                                    }
                                    ai->cb->SetFigureColor(figureID, 0.0f, 65535.0f, 0.0f, 0.2f);
                            }
                            return true;
                    }
            }
      } else { //endif there are enemies
            //attempt to path back to base if there are no targets
            //L("AG: found no defense targets, pathing back to base. group:" << groupID);
            //this->PathToBase(groupPosition);
      }
      if (!isShooting && !isMoving) {
            if (ai->ah->debug) L("AttackGroup: There are no accessible enemies and we're
idling. groupid:" << this->groupID << " we are defending: " << this->isDefending);
      }
      return false;
}

void CAttackGroup::MicroUpdate(float3 groupPosition) {
ai->math->StartTimer(ai->ah->ah_timer_MicroUpdate);
      int numUnits = (int)units.size();
      assert (numUnits > 0);
      if (numUnits == 0) return;
      int frameNr = ai->cb->GetCurrentFrame();
      int frameSpread = 15;
      if (!isFleeing && frameNr % frameSpread == (groupID*4) % frameSpread) {
//            L("AG: isShooting start");
            this->isShooting = false;
            //get all enemies within attack range:
            float searchRange = this->highestAttackRange + highestTurnRadius +
groupPhysicalSize;
            int numEnemies = ai->cheat->GetEnemyUnits(unitArray, groupPosition, searchRange);
//            L("AG shoot debug 1 numEnemies:" << numEnemies << " range:" << range);
            if (numEnemies > 0) {
```

```
		//selecting one of the enemies
		int enemySelected = -1;
		{ //local variable space for selecting the enemy
			float distanceToTempTarget = FLT_MAX;
			float temp;
			//bool closestSoFarIsBuilding = false; shoot units first
			CUNIT* exampleUnit = ai->MyUnits[units.front()];
			bool foundUnitThatCanAttackMe = false;
			bool tempItCanAttackMe = false;
			const UnitDef* enemyDef;
			int currentEnemyID;
			for (int i = 0; i < numEnemies; i++) {
				//pick our general target preference
				currentEnemyID = unitArray[i];
				enemyDef = ai->cheat->GetUnitDef(currentEnemyID);
				if (	enemyDef != NULL
						&& CloakedFix(unitArray[i])
						&& (((temp = groupPosition.distance2D(ai->cheat-
>GetUnitPos(currentEnemyID))) < distanceToTempTarget && foundUnitThatCanAttackMe) //when were
are only considering threats AND this is better
							|| (temp < distanceToTempTarget && !foundUn
itThatCanAttackMe) //the regular version with no threats
							|| (!foundUnitThatCanAttackMe &&
(tempItCanAttackMe = exampleUnit->CanAttackMe(currentEnemyID))))//its the first that can attack
me
						&& exampleUnit->CanAttack(currentEnemyID)) {
					enemySelected = i;
					distanceToTempTarget = temp;
					if (!foundUnitThatCanAttackMe && tempItCanAttackMe)
foundUnitThatCanAttackMe = true;
				}
			}
		}
/*
mass Note::
check dps of selected enemy
check your dps vs selected enemy and check if its a high percentage of our highest average dps
and thus if its worth considering
store these things in the target selection process
Watch out, the code indentation gets bad around here.
*/
		// theres some kind of target in range
		if (enemySelected != -1) {
			//X marks the target
			float3 enemyPos = ai->cheat->GetUnitPos(unitArray[enemySelected]);
			int xSize = 40;
			int lineGroupID = groupID+5000;
			int heightOffset = 20;
			if (ai->ah->debugDraw) ai->cb->CreateLineFigure(enemyPos + float3(-
xSize,heightOffset,-xSize), enemyPos + float3(xSize,heightOffset,xSize), 5, 0, frameSpread,
lineGroupID);
			if (ai->ah->debugDraw) ai->cb->CreateLineFigure(enemyPos + float3(-
xSize,heightOffset,xSize), enemyPos + float3(xSize,heightOffset,-xSize), 5, 0, frameSpread,
lineGroupID);
//			float3 enemyPos = ai->cheat->GetUnitPos(unitArray[enemySelected]); //
Note:: add half the height of the unit, or something of that effect. or our unit weapon height.
			assert (CloakedFix(unitArray[enemySelected]));
			//Note:: add some check verifying that the target can actually be
reached by a given unit.
			float myDPS = this->DPS();
			//float threatAtEnemy = ai->tm->ThreatAtThisPoint(enemyPos) - ai->tm-
>GetUnmodifiedAverageThreat();
			//const float FLEE_MAX_THREAT_DIFFERENCE = 1.30f;
			//Note:, This is at fault, wrong position is being checked
			//me-----------range------>enemy
			//           check here   not here
			float3 vectorUsToEnemy = enemyPos - groupPos;
			//Note:: maybe averagerange? hmm
			float distanceToEnemy = groupPos.distance2D(enemyPos);
```

```
                        float distanceCheckPoint = max (distanceToEnemy - this-
>lowestAttackRange, 0); //shouldnt be behind our current pos
                        float factor = distanceCheckPoint/distanceToEnemy;
                        float3 pointAtRange = groupPos + (vectorUsToEnemy*factor);
                        float threatAtRange = ai->tm->ThreatAtThisPoint(pointAtRange) - ai-
>tm->GetUnmodifiedAverageThreat();

                        if (ai->ah->debugDraw) {
                              //Debug: draw the threat-checked position
                              xSize = 10;
                              lineGroupID = groupID+15000;
                              heightOffset = 20;
                              ai->cb->CreateLineFigure(pointAtRange + float3(-
xSize,heightOffset,-xSize), pointAtRange + float3(xSize,heightOffset,xSize), 5, 0, frameSpread,
lineGroupID);
                              ai->cb->CreateLineFigure(pointAtRange + float3(-xSize,heightOff
set,xSize), pointAtRange + float3(xSize,heightOffset,-xSize), 5, 0, frameSpread, lineGroupID);
                        }
                        //GIVING ATTACK ORDER to each unit:
                        if (myDPS * FLEE_MAX_THREAT_DIFFERENCE > threatAtRange) { //attack
                              this->isShooting = true;
                              int myUnit;
                              const UnitDef* myUnitDef;
                              float3 myPos;
                              int enemyID = unitArray[enemySelected];
                              const UnitDef* enemyUnitDef = ai->cheat->GetUnitDef(enemyID);
                              CUNIT* myUnitReference;
                              float myMaxRange;
                              float myMinRange;
                              for (int i = 0; i < numUnits; i++) {
                                    myUnit = units[i];
                                    myUnitDef = ai->cb->GetUnitDef(myUnit);
                                    //does our unit exist and its not currently maneuvering

                                    if (myUnitDef != NULL && (myUnitReference = ai-
>MyUnits[myUnit])->maneuverCounter-- <= 0) {
                                          //add a routine finding the best(not just closest)
target, but for now just fire away
                                          //Note:: add canattack
                                          //position focus fire for buildings
                                          if(enemyUnitDef->speed > 0) {
                                                myUnitReference->Attack(enemyID); //
                                          } else {
                                                myUnitReference->Attack(enemyPos, 0.001f);
                                          }
                                          //Note:: this should be the max range of the
weapon the unit has with the lowest range
                                          //assuming you want to rush in with the heavy
stuff, that is
                                          //SINGLE UNIT MANEUVERING:

                                          //Note: SOMEHOW PICK CLOSEST UNIT IN THE ARRAY
THAT CAN FIRE AT ME or something
                                          //since targetting selection changed this might
have to change too

                                          //testing the possibility of retreating to max
range if target is too close
                                          myPos = ai->cb->GetUnitPos(myUnit);
                                          myMaxRange = this->ai->ut->GetMaxRange(myUnitDef);
                                          myMinRange = this->ai->ut->GetMinRange(myUnitDef);
                                          //is it air, //or air thats landed
                                          if(!myUnitDef->canfly //if we are not a plane
                                                      // AND difference makes it a good
idea
                                                      && (myMinRange - UNIT_MIN_MANEUVER_
RANGE_DELTA) > myPos.distance2D(enemyPos)
                                                      //AND it can attack me
                                                      && ai->MyUnits[myUnit]->CanAttackMe(
```

```
unitArray[enemySelected])) {
                                                        bool debug1 = true;
                                                        bool debug2 = false;
                                                        static vector<float3> tempPath;
                                                        tempPath.clear();
                                                        //two things: 1. dont need a path,
just a position and 2. it should avoid other immediate friendly units and/or immediate enemy
units+radius
                                                        //maybe include the height parameter in the
search? probably not possible
                                                        //Note:: OBS doesnt this mean pathing might
happen every second? outer limit should be more harsh than inner
                                                        float3 moveHere;
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
                                                        //OBS: the move type of the single unit.
maybe the group center unit will maneuver to positions that the group itself cant path from ?
                                                        ai->pather->micropather->SetMapData(ai-
>pather->canMoveIntMaskArray,ai->tm->ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight,
myUnitReference->GetMoveType());
                                                        float cost = ai->pather->ManeuverToPosRadiu
s(&moveHere, myPos, enemyPos, myMinRange);
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
                                                        if (cost > 0) { //Note:: is this the right
check at this point ?
                                                                float dist = myPos.
distance2D(moveHere);
                                                                //is the position between the
proposed destination and the enemy higher than the average of mine and his height?
                                                                bool losCheck = ((moveHere.y +
enemyPos.y)/2.0f) + UNIT_MAX_MANEUVER_HEIGHT_DIFFERENCE_UP > ai->cb->GetElevation((moveHere.
x+enemyPos.x)/2, (moveHere.z+enemyPos.z)/2);
                                                                //im here assuming the pathfinder
returns correct Y values
                                                                //get threat info:
                                                                float currentThreat = ai->tm->Threat
AtThisPoint(myPos);
                                                                float newThreat = ai->tm->ThreatAtTh
isPoint(moveHere);
        //                                                      if (newThreat <= currentThreat &&
dist > max((UNIT_MIN_MANEUVER_RANGE_PERCENTAGE*myRange), UNIT_MIN_MANEUVER_DISTANCE)
                                                                if (newThreat < currentThreat &&
dist > max((UNIT_MIN_MANEUVER_RANGE_PERCENTAGE*myMinRange), UNIT_MIN_MANEUVER_DISTANCE)
                                                                        && losCheck) { //(enemyPos.y
- moveHere.y) < UNIT_MAX_MANEUVER_HEIGHT_DIFFERENCE_UP) {
                                                                        debug2 = true;
                                                                        //Draw where it would move:
                                                                        if (ai->ah->debugDraw) {
                                                                                int figID =
(RANDINT%1000000) + 400;
                                                                                ai->cb-
>CreateLineFigure(myPos, moveHere, 65, 1, frameSpread*4, figID);
                                                                                ai->cb-
>SetFigureColor(figID, 65535.0f, 65535.0f, 0.0f, 0.2f);
                                                                        }
                                                                        myUnitReference-
>maneuverCounter = ceil(max((float)UNIT_MIN_MANEUVER_TIME/frameSpread, (dist/myUnitDef->speed)));
        //                                                      L("AG maneuver debug,
maneuverCounter set to " << ai->MyUnits[unit]->maneuverCounter << " and cost:" << cost << "
speed:" << ai->MyUnits[unit]->def()->speed << " frameSpread:" << frameSpread << " dist/speed:"
<< (dist/ai->MyUnits[unit]->def()->speed) << " its a " << ai->MyUnits[unit]->def()->humanName);
                                                                        myUnitReference-
>Move(moveHere);
                                                                }
                                                        }
                                                        if (debug1 && !debug2) {
                                                                //L("AG: maneuver: pathfinder run
but path not used");
                                                                //Note:: hack, and re-enable L()
                                                                myUnitReference->maneuverCounter =
```

```
2;
                                                        }
                                                }//endif we are not a plane && we want to manuver
right now
                                        }// we exist and arent already involved in maneuvering
                                }
                        //} else { //there are enemies but my dps is lower than at in range of
enemy
                                //Flee();
                        }
                }
        }
//          L("AG: isShooting end");
    }
ai->math->StopTimer(ai->ah->ah_timer_MicroUpdate);
}

/*
Note:: make it support planes by:
1. letting speed determine how far ahead the maxStepsAhead is - DONE
2. using priority-searches to enemy structures for fast GROUPS
will support bomb runs and also zipper runs (if group formation permits)
Note:2: make units rush to their real target and not stop to mess around with whatever they find
(needs integration with micro)
*/

void CAttackGroup::MoveOrderUpdate(float3 groupPosition) {
ai->math->StartTimer(ai->ah->ah_timer_MoveOrderUpdate);
       int numUnits = this->units.size();
       int frameNr = ai->cb->GetCurrentFrame();
       int frameSpread = 60;
       if (!isShooting && isMoving && frameNr % frameSpread == 0 && pathToTarget.size() > 2) {
              //do the moving
              float distPerStep = pathToTarget[0].distance2D(pathToTarget[1]);
              const int maxStepsAhead = max(1, ceil(frameSpread/30 * this->highestUnitSpeed/
distPerStep));
              int pathMaxIndex = (int)pathToTarget.size()-1;
              int step1 = min(this->pathIterator + maxStepsAhead, pathMaxIndex);
              int step2 = min(this->pathIterator + maxStepsAhead*2, pathMaxIndex);
              //Note:: maybe add step 3 ? :P
              float3 moveToHereFirst = this->pathToTarget[step1];
              float3 moveToHere = this->pathToTarget[step2];
              //drawing destination
        //if we aint there yet
              if (groupPosition.distance2D(pathToTarget[pathMaxIndex]) > GROUP_DESTINATION_SLACK
+ this->groupPhysicalSize + this->highestTurnRadius) {
                     //increase pathIterator
                     pathIterator = 0;
                     float3 tempAhead;
                     tempAhead = pathToTarget[min(pathIterator + maxStepsAhead, pathMaxIndex)];
                     float3 tempBehind;
                     if (pathIterator - maxStepsAhead < 0) {
                            tempBehind = pathToTarget[0];
                     } else {
                            tempBehind = pathToTarget[pathIterator - maxStepsAhead];
                     }
                     while(pathIterator < pathMaxIndex
                                   && groupPosition.distance2D(tempAhead) - this-
>groupPhysicalSize - this->highestTurnRadius < groupPosition.distance2D(tempBehind)) {
                            pathIterator = min(pathIterator + maxStepsAhead, pathMaxIndex);
                            tempAhead = pathToTarget[min(pathIterator + maxStepsAhead,
pathMaxIndex)];
                            if (pathIterator - maxStepsAhead < 0) {
                                   tempBehind = pathToTarget[0];
                            } else {
                                   tempBehind = pathToTarget[pathIterator - maxStepsAhead];
                            }
                     }
                     //check the threat at group pos and the threat at the next pos on the path,
```

```
if its high, flee
                    float threatCheck = max(ai->tm->ThreatAtThisPoint(pathToTarget[pathIterator]
), ai->tm->ThreatAtThisPoint(groupPos)) - ai->tm->GetUnmodifiedAverageThreat();
                    if (!isFleeing && threatCheck > this->groupDPS*FLEE_MAX_THREAT_DIFFERENCE) {
                          ai->math->StopTimer(ai->ah->ah_timer_MoveOrderUpdate);
                          this->Flee();
                          if (ai->ah->debug) L("CAttackGroup::MoveOrderUpdate caused
group:"<<groupID<<" to flee");
                          return;
                    }
                    //move the units
                    for(int i = 0; i < numUnits; i++) {
                          int unit = units[i];
                          if (ai->cb->GetUnitDef(unit) != NULL) {
                                //Note:: when they are near target, change this so they line up
or something while some are here and some arent. attack checker will take over soon.
                                //if the single unit aint there yet
                                if (ai->cb->GetUnitPos(unit).distance2D(pathToTarget[pathMaxInd
ex]) > UNIT_DESTINATION_SLACK) {
//                                    ai->cb->CreateLineFigure(ai->cb->GetUnitPos(unit) +
float3(0,50,0), moveToHereFirst + float3(0,50,0),15,1,10,groupID+50000);
                                    if (moveToHere != moveToHereFirst) {
                                          ai->MyUnits[unit]->MoveTwice(&moveToHereFirst,
&moveToHere);
//                                          ai->cb->CreateLineFigure(moveToHereFirst +
float3(0,50,0), moveToHere + float3(0,50,0),15,1,10,groupID+50000);
                                    } else {
                                          ai->MyUnits[unit]->Move(moveToHereFirst);
                                    }
                                    //draw thin line from unit to groupPos
                                    if (ai->ah->debugDraw) ai->cb->CreateLineFigure(ai->cb-
>GetUnitPos(unit) + float3(0,50,0), groupPosition + float3(0,50,0),4,0,frameSpread,groupID+500);
                                }
                          }
                    }
              } else {
                    if (ai->ah->debug) L("AG: group thinks it has arrived at the destination:"
<< groupID);
                    this->ClearTarget();
              }
       }//endif move
ai->math->StopTimer(ai->ah->ah_timer_MoveOrderUpdate);
}

void CAttackGroup::StuckUnitFix() {
       int frameNr = ai->cb->GetCurrentFrame();
       int frameSpread = 300;
       if (isMoving && !isShooting && frameNr % frameSpread == 0) {
//            L("AG: unit stuck checking start");
              //stuck unit update. (at a 60 frame modulo)
              for (vector<int>::iterator it = units.begin(); it != units.end(); it++) {
                    int unit = *it;
                    if (ai->cb->GetUnitDef(unit) != NULL){
                          CUNIT *u = ai->MyUnits[unit];
                          //hack for slight maneuvering around buildings, spring fails at this
sometimes
                          if (u->stuckCounter >= UNIT_STUCK_COUNTER_MANEUVER_LIMIT && !u-
>attemptedUnstuck) {
                                if (ai->ah->debug) L("AG: attempting unit unstuck manuevering
for " << unit << " at pos " << u->pos().x << " " << u->pos().y << " " << u->pos().z);
                                if (ai->ah->debugDraw) {
                                      AIHCAddMapPoint amp;
                                      amp.label = "stuck-fix";
                                      amp.pos = u->pos();
                                      ai->cb->HandleCommand(&amp);
                                }
                                //findbestpath to periphery of circle around mypos, distance 2x
resolution or somesuch
                                vector<float3> tempPath;
```

```
                                float3 destination;
ai->math->StopTimer(ai->ah->ah_timer_totalTimeMinusPather);
                                ai->pather->micropather->SetMapData(ai->pather-
>canMoveIntMaskArray,ai->tm->ThreatArray,ai->tm->ThreatMapWidth,ai->tm->ThreatMapHeight, this-
>GetWorstMoveType());
                                float dist = ai->pather->ManeuverToPosRadius(&destination, u-
>pos(), u->pos(), UNIT_STUCK_MANEUVER_DISTANCE);
ai->math->StartTimer(ai->ah->ah_timer_totalTimeMinusPather);
                                if (dist > 0) {
                                        if (ai->ah->debugDraw) ai->cb->CreateLineFigure(u-
>pos(), destination, 14, 1, frameSpread, RANDINT%1000000);
                                        u->Move(destination);
                                }
                                u->stuckCounter = 0;
                                u->attemptedUnstuck = true;
                        }
                }
            }//endfor unstuck
        }
}

void CAttackGroup::Update()
{
        int frameNr = ai->cb->GetCurrentFrame();
        int numUnits = (int)units.size();
        if(!numUnits) {
                if (ai->ah->debug) L("updated an empty AttackGroup!");
                return;
        }

//      int frameSpread; // variable used as a varying "constant" for determining how often a part
of the update scheme is done
        float3 groupPosition = this->GetGroupPos();
        if(groupPosition == ERRORVECTOR) return;

        if (ai->ah->debugDraw) DrawGroupPosition();

        //sets the isShooting variable. - this part of the code checks for nearby enemies and does
focus fire/maneuvering
        this->MicroUpdate(groupPosition);

        if (ai->ah->debugDraw && frameNr % 30 == 0 && !isDefending) {
                //drawing the target region:
//              L("AG update: drawing target region (attacking)");
                int heightOffset = 50;
                vector<float3> circleHack;
                float diagonalHack = attackRadius * (2.0f/3.0f);
                circleHack.resize(8, attackPosition);
                circleHack[0] += float3(-diagonalHack,heightOffset,-diagonalHack);
                circleHack[1] += float3(0,heightOffset,-attackRadius);
                circleHack[2] += float3(diagonalHack,heightOffset,-diagonalHack);
                circleHack[3] += float3(attackRadius,heightOffset,0);
                circleHack[4] += float3(diagonalHack,heightOffset,diagonalHack);
                circleHack[5] += float3(0,heightOffset,attackRadius);
                circleHack[6] += float3(-diagonalHack,heightOffset,diagonalHack);
                circleHack[7] += float3(-attackRadius,heightOffset,0);
                for(int i = 0; i < 8; i++) {
                        ai->cb->CreateLineFigure(circleHack[i], circleHack[(i+1)%8], 5, 0, 300,
GetGroupID()+6000);
                }
                //from pos to circle center
                ai->cb->CreateLineFigure(groupPosition+float3(0,heightOffset,0), attackPosition+flo
at3(0,heightOffset,0), 5, 1, 30, GetGroupID()+6000);
                ai->cb->SetFigureColor(GetGroupID() + 6000, 0, 0, 0, 1.0f);
//              L("AG update: done drawing stuff");
        }

        //if we're defending and we dont have a target, every 2 secs look for a target
//      if (defending && !isShooting && !isMoving && frameNr % 60 == groupID % 60) {
```

```
//            FindDefenseTarget(groupPosition);
//     }
       // GIVE MOVE ORDERS TO UNITS ALONG PATHTOTARGET
       this->MoveOrderUpdate(groupPosition);

       //stuck fix stuff.
       StuckUnitFix();
}
```

Metal Maker economic control class:

Please note that this part of the code includes references to TA Spring engine source code, because of the callback functions used to retrieve details about the economic situation of the AI.

Metalmaker.h:

```
#pragma once
#include "ExternalAI/IGroupAI.h"
#include <map>
#include <set>
using namespace std;

class CMetalMaker
{
public:
       CMetalMaker(IAICallback* aicb);
       virtual ~CMetalMaker();
//     virtual void CMetalMaker::Init(IAICallback* aicb);
       virtual bool Add(int unit);
       virtual bool Remove(int unit);
       virtual bool AllAreOn();
       virtual void Update();
       struct UnitInfo{
              int id;
              float energyUse;
              float metalPerEnergy;
              bool turnedOn;
       };
       vector<UnitInfo> myUnits;
       float lastEnergy;
       IAICallback* aicb;
       int listIndex;
       int addedDelay;

};
```

MetalMaker.cpp:

```
// implementation of the logic in a metal maker class which adjusts energy consumption
// to buildings which expend energy in order to produce or extract metal.

#include "MetalMaker.h"
#include "ExternalAI/IAICallback.h"
#include "Sim/Units/UnitDef.h"
#include <vector>


//namespace std{
//     void _xlen(){};
//}

CMetalMaker::CMetalMaker(IAICallback* aicb)
{
//     lastUpdate=0;
       listIndex=0;
       lastEnergy=0;
       addedDelay=0;
       this->aicb=aicb;
}
```

```
CMetalMaker::~CMetalMaker()
{
//	for(map<int,UnitInfo*>::iterator ui=myUnits.begin();ui!=myUnits.end();++ui)
//		delete ui->second;
	myUnits.clear();
}

/*void CMetalMaker::Init()
{
//	this->callback=callback;
	this->aicb=aicb;
}*/

bool CMetalMaker::Add(int unit)
{
	const UnitDef* ud=aicb->GetUnitDef(unit);
	if(!(ud->energyUpkeep>0.0f && ud->makesMetal>0.0f)){ // || !ud->extractsMetal>0.0f) {
//		aicb->SendTextMsg("Can only use metal makers",0);
		return false;
	}
	//analyse the command and stuff on or off into the info thingy
	UnitInfo info;//=new UnitInfo;
	const std::vector<CommandDescription>* cd=aicb->GetUnitCommands(unit);
	for(std::vector<CommandDescription>::const_iterator cdi=cd->begin();cdi!=cd->end();++cdi){
		if(cdi->id==CMD_ONOFF){
			int on=atoi(cdi->params[0].c_str());
			if(on)
				info.turnedOn=true;
			else
				info.turnedOn=false;
			break;
		}
	}
	info.id = unit;
	info.energyUse=ud->energyUpkeep;
	info.metalPerEnergy = ud->makesMetal/ud->energyUpkeep;
	//myUnits[unit]=info;

	//myUnits.push_back(info);

	//insert sorted by metalPerEnergy
	bool inserted = false;
	int counter = 0;
	vector<UnitInfo>::iterator theIterator = myUnits.begin();
	while(theIterator != myUnits.end() && !inserted) {
		//insert it where it should be in order for the list to remain sorted (insertion
sort)
		if (info.metalPerEnergy > theIterator->metalPerEnergy
			//in addition, provide a sexy ordering ability ;P
				|| (info.metalPerEnergy == theIterator->metalPerEnergy
					&& (aicb->GetUnitPos(info.id).x == aicb-
>GetUnitPos(theIterator->id).x
					&& aicb->GetUnitPos(info.id).z > aicb->GetUnitPos(theIterator-
>id).z)
				|| (info.metalPerEnergy == theIterator->metalPerEnergy
					&& aicb->GetUnitPos(info.id).x > aicb->GetUnitPos(theIterator-
>id).x))
			) {
			myUnits.insert(theIterator, info);
			inserted = true;
			break;
		}
		theIterator++;
		counter++;
	}
	if(!inserted) {
		myUnits.push_back(info);
	}
	if (counter < listIndex) {
```

```
		//make sure its on and increase index
		if(!myUnits[counter].turnedOn)
		{
			Command c;
			c.id=CMD_ONOFF;
			c.params.push_back(1);
			aicb->GiveOrder(myUnits[counter].id,&c);
			myUnits[counter].turnedOn=true;
		}
		listIndex++;

	} else {
		//make sure its off
		if(myUnits[counter].turnedOn)
		{
			Command c;
			c.id=CMD_ONOFF;
			c.params.push_back(0);
			aicb->GiveOrder(myUnits[counter].id,&c);
			myUnits[counter].turnedOn=false;
		}
	}
//	char text[500];
//	sprintf(text, "CFirenuMM: metal maker added, id=%i, index=%i, num=%i, energyUpkeep=%f,
makesMetal=%f", unit, counter, (int)myUnits.size(), ud->energyUpkeep, ud->makesMetal);
//	aicb->SendTextMsg(text, 0);
//	for(int i = 0; i < (int)myUnits.size(); i++) {
//		sprintf(text, "-%f", myUnits[i].metalPerEnergy);
//		aicb->SendTextMsg(t, 0);
//	}
	//aicb->SendTextMsg(t, 0);



	return true;
}

bool CMetalMaker::Remove(int unit)
{
	bool found = false;
	vector<UnitInfo>::iterator it;
	int counter = 0;
	for (it = myUnits.begin(); it != myUnits.end(); it++) {
		if (it->id == unit) {
			found = true;
			break;
		}
		counter++;
	}
	if (found) myUnits.erase(it);
	//is this index below listIndex
	if (counter < listIndex && listIndex > 0) {
		listIndex--;
	}
	return found;
}

bool CMetalMaker::AllAreOn() {
	return this->listIndex == ((int)this->myUnits.size()-1) || myUnits.size() == 0;
}

void CMetalMaker::Update()
{
	int frameNum=aicb->GetCurrentFrame();
	const int updateSpread = 33;
	int numUnits = (int)myUnits.size();
	if(frameNum % updateSpread == 0 && numUnits > 0 && addedDelay-- <= 0){
		//lastUpdate=frameNum;
		float energy=aicb->GetEnergy();
```

```
		float estore=aicb->GetEnergyStorage();

		float difference=energy-lastEnergy;
		difference /= 4.0f;
		lastEnergy=energy;

		//turn something off
		if(energy<estore*0.6) {// && listIndex > 0){
			while (difference < 0 && listIndex > 0) {
				//listIndex points at the first offline one
				listIndex--; //now it should be the last online one
//				char t[500];
//				sprintf(t, "CFirenuMM: %i-->off %f", listIndex, energy/estore);
//				aicb->SendTextMsg(t, 0);
				if(myUnits[listIndex].turnedOn)
				{
					Command c;
					c.id=CMD_ONOFF;
					c.params.push_back(0);
					aicb->GiveOrder(myUnits[listIndex].id,&c);
					myUnits[listIndex].turnedOn=false;
					//break; //moo
					difference += myUnits[listIndex].energyUse;
				}
			}
			addedDelay = 4;
		//turn something on
		} else if(energy>estore*0.9 && listIndex < numUnits){
//			char t[500];
//			sprintf(t, "CFirenuMM: %i--->on %f", listIndex, energy/estore);
//			aicb->SendTextMsg(t, 0);
			if(!myUnits[listIndex].turnedOn)
			{
				Command c;
				c.id=CMD_ONOFF;
				c.params.push_back(1);
				aicb->GiveOrder(myUnits[listIndex].id,&c);
				myUnits[listIndex].turnedOn=true;
				//break; //moo
				if (difference < myUnits[listIndex].energyUse) addedDelay = 4;
			}
			listIndex++;//set it to point to the next one, which should be offline
		}


/*
		float dif=energy-lastEnergy;
		lastEnergy=energy;
		//turn something off
		if(energy<estore*0.8){
			float needed=-dif+4;			//how much energy we need to save to turn
positive
			for(int i = 0; i < (int)myUnits.size(); i++) {		//find makers to turn
off
				if(needed<0)
					break;
				if(myUnits[i].turnedOn){
					needed-=myUnits[i].energyUse;
					Command c;
					c.id=CMD_ONOFF;
					c.params.push_back(0);
					aicb->GiveOrder(myUnits[i].id,&c);
					myUnits[i].turnedOn=false;
					//break; //moo
				}
			}
		//turn something on
		} else if(energy>estore*0.9){
			float needed=dif+4;		//how much energy we need to start using to turn
```

```
negative
                    for(int i = 0; i < (int)myUnits.size(); i++) {        //find makers to turn
on
                          if(needed<0)
                                break;
                          if(!myUnits[i].turnedOn){
                                needed-=myUnits[i].energyUse;
                                Command c;
                                c.id=CMD_ONOFF;
                                c.params.push_back(1);
                                aicb->GiveOrder(myUnits[i].id,&c);
                                myUnits[i].turnedOn=true;
                                //break; //moo
                          }
                    }
                    //lastUpdate=frameNum-updateSpread; //want to be sure i dont waste any E,
while using too much E is slightly more acceptable
             }
             */
      }
      if ((aicb->GetCurrentFrame() % 1800) == 0) {
             //once a minute, turn everything off and reset.
             for (int i = 0; i < (int)myUnits.size(); i++) {
                    Command c;
                    c.id=CMD_ONOFF;
                    c.params.push_back(0);
                    aicb->GiveOrder(myUnits[i].id,&c);
                    myUnits[i].turnedOn = false;
             }
             this->listIndex = 0;
             this->addedDelay = 0;
//           aicb->SendTextMsg("MM: turned all off.", 0);
      }
}
```

CGlobalAI

This part has the CGlobalAI code which includes the main AI update function which executes most of the other code in the AI. This part of the code also includes most of the interface which passes event driven messages from the engine to the AI.

CGlobalAI.h:

```
#pragma once


#include "include.h"

const char AI_NAME[]="KAI";

using namespace std;
//#pragma warning(disable: 4244 4018 4996 4129)

//edit:
class CAttackHandler;

class CGlobalAI : public IGlobalAI
{
public:
	CGlobalAI();
	virtual ~CGlobalAI();

	void InitAI(IGlobalAICallback* callback, int team);

// LIST OF EVENT DRIVEN FUNCTIONS CALLED FROM THE SPRING ENGINE:
	void UnitCreated(int unit);
//called when a new unit is created on ai team
	void UnitFinished(int unit);                                                    //called
when an unit has finished building
	void UnitDestroyed(int unit,int attacker);
//called when a unit is destroyed

	void EnemyEnterLOS(int enemy);
	void EnemyLeaveLOS(int enemy);

	void EnemyEnterRadar(int enemy);
	void EnemyLeaveRadar(int enemy);

	void EnemyDestroyed(int enemy,int attacker);                                                //
will be called if an enemy inside los or radar dies (note that leave los etc will not be called
then)

	void UnitIdle(int unit);
//called when a unit go idle and is not assigned to any group

	void GotChatMsg(const char* msg,int player);                                //called when
someone writes a chat msg

	void UnitDamaged(int damaged,int attacker,float damage,float3 dir);
	//called when one of your units are damaged
	void UnitMoveFailed(int unit);
	int HandleEvent (int msg,const void *data);

//END OF EVENT DRIVEN MESSAGES

//update is called every frame, 30 times per second, and all other AI updates are done from
here.
	void Update();


	AIClasses *ai;
	vector<CUNIT> MyUnits;
```

```
        char c[512];

        private:
        int totalSumTime;
        int updateTimerGroup;
        int econTrackerFrameUpdate;
        int updateTheirDistributionTime;
        int updateMyDistributionTime;
        int builUpTime;
        int idleUnitUpdateTime;
        int attackHandlerUpdateTime;
        int MMakerUpdateTime;
        int unitCreatedTime;
        int unitFinishedTime;
        int unitDestroyedTime;
        int unitIdleTime;
        int economyManagerUpdateTime;
        int globalAILogTime;
        int threatMapTime;


};
```

CGlobalAI.cpp:

```
// GroupAI.cpp: implementation of the CGroupAI class.
//
//////////////////////////////////////////////////////////////////////

#include "GlobalAI.h"
#include "unit.h"


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

namespace std{
	void _xlen(){};
}

//constructor. see initAI function.
CGlobalAI::CGlobalAI()
{

}

//deconstructor
CGlobalAI::~CGlobalAI()
{
	delete ai->LOGGER;
	delete ai->ah;
	delete ai->bu;
	delete ai->econTracker;
	delete ai->parser;
	delete ai->math;
	delete ai->debug;
	delete ai->pather;
	delete ai->tm;
	delete ai->ut;
	delete ai->mm;
	delete ai->uh;
	delete ai;
}

void CGlobalAI::InitAI(IGlobalAICallback* callback, int team)
{
	// Create folders if theyre not there already
	int timetaken = clock();
	_mkdir(ROOTFOLDER);
	_mkdir(LOGFOLDER);
	_mkdir(METALFOLDER);
	_mkdir(TGAFOLDER);


	// Initialize Log filename
	string mapname = string(callback->GetAICallback()->GetMapName());
	mapname.resize(mapname.size()-4);
	time_t now1;
    time(&now1);
    struct tm *now2;
    now2 = localtime(&now1);
	// Date logfile name
	sprintf(c, "%s%s %2.2d-%2.2d-%4.4d %2.2d%2.2d (%d).log",string(LOGFOLDER).c_str(),
		mapname.c_str(),now2->tm_mon+1, now2->tm_mday, now2->tm_year + 1900,
			now2->tm_hour,now2->tm_min, team);

//INITIALIZE ALL AI MODULES AND SAVE REFERENCE IN THE CALLBACK STRUCT
	ai                          = new AIClasses;
```

```
	ai->cb              = callback->GetAICallback();
	ai->cheat           = callback->GetCheatInterface();

	MyUnits.reserve(MAXUNITS);
	ai->MyUnits.reserve(MAXUNITS);
	for (int i = 0; i < MAXUNITS;i++){
		MyUnits.push_back(CUNIT(ai));
		MyUnits[i].myid = i;
		MyUnits[i].groupID = -1;
		ai->MyUnits.push_back(&MyUnits[i]);
	}

	ai->debug           = new CDebug(ai);
	ai->math            = new CMaths(ai);
	ai->LOGGER          = new std::ofstream(c);
	ai->parser          = new CSunParser(ai);
	ai->sh              = new CSurveillanceHandler(ai);
	ai->ut              = new CUnitTable(ai);
	ai->mm              = new CMetalMap(ai);
	ai->pather          = new CPathFinder(ai);
	ai->tm              = new CThreatMap(ai);
	ai->uh              = new CUnitHandler(ai);
	ai->dm              = new CDefenseMatrix(ai);
	ai->econTracker = new CEconomyTracker(ai); // This is a temp only
	ai->bu              = new CBuildUp(ai);
	ai->ah              = new CAttackHandler(ai);
	ai->dc              = new CDamageControl(ai);
	ai->em              = new CEconomyManager(ai);
	L("All Class pointers initialized");

	// INITIALIZE ALL TIMERS USED FOR EVALUATING ALGORITHM EFFICIENCY AND CPU USE:

	totalSumTime = 0;
	updateTimerGroup = ai->math->GetNewTimerGroupNumber("CGlobalAI::Update()");
	econTrackerFrameUpdate = ai->math->GetNewTimerGroupNumber("ai->econTracker-
>frameUpdate()");
	updateTheirDistributionTime = ai->math->GetNewTimerGroupNumber("ai->dc->UpdateTheirDistrib
ution()");
	updateMyDistributionTime = ai->math->GetNewTimerGroupNumber("ai->dc->UpdateMyDistribution(
)");
	threatMapTime = ai->math->GetNewTimerGroupNumber("ai->tm->Create()  (threatMap)");
	builUpTime = ai->math->GetNewTimerGroupNumber("ai->bu->Update()  (buildup)");
	idleUnitUpdateTime = ai->math->GetNewTimerGroupNumber("idleUnitUpdateTime");
	attackHandlerUpdateTime = ai->math->GetNewTimerGroupNumber("ai->ah->Update()
(attackHandler)");
	MMakerUpdateTime = ai->math->GetNewTimerGroupNumber("ai->uh->MMakerUpdate()");
	economyManagerUpdateTime = ai->math->GetNewTimerGroupNumber("ai->em->Update()
(economyManager)");
	globalAILogTime = ai->math->GetNewTimerGroupNumber("GlobalAI log time  ( L() )");
	unitCreatedTime = ai->math->GetNewTimerGroupNumber("CGlobalAI::UnitCreated(int unit)");
	unitFinishedTime = ai->math->GetNewTimerGroupNumber("CGlobalAI::UnitFinished(int unit)");
	unitDestroyedTime = ai->math->GetNewTimerGroupNumber("CGlobalAI::UnitDestroyed(int
unit,int attacker)");
	unitIdleTime = ai->math->GetNewTimerGroupNumber("CGlobalAI::UnitIdle(int unit)");

	L("Timers initialized");
	ai->mm->Init();
	L("ai->mm->Init(); done");
	ai->ut->Init();
	L("ai->ut->Init(); done");
	ai->pather->Init();
	L("init done");
	ai->dc->GenerateDPSTables();
	L("GenerateDPSTables done");
}

void CGlobalAI::UnitCreated(int unit)
{
	ai->math->StartTimer(totalSumTime);
```

```
	ai->math->StartTimer(globalAILogTime);
	L("GlobalAI::UnitCreated is called on unit:" << unit <<". its groupid:" << ai-
>MyUnits[unit]->groupID << " name:" << ai->MyUnits[unit]->def()->humanName);
	ai->math->StopTimer(globalAILogTime);
	ai->math->StartTimer(unitCreatedTime);
	ai->uh->UnitCreated(unit);
	ai->econTracker->UnitCreated(unit);
	ai->math->StopTimer(unitCreatedTime);
	ai->math->StopTimer(totalSumTime);
}

void CGlobalAI::UnitFinished(int unit)
{
	ai->math->StartTimer(totalSumTime);
	ai->math->StartTimer(globalAILogTime);
	L("GlobalAI::UnitFinished is called on unit:" << unit <<". its groupid:" << ai-
>MyUnits[unit]->groupID << " name:" << ai->MyUnits[unit]->def()->humanName);
	ai->math->StopTimer(globalAILogTime);
	ai->math->StartTimer(unitFinishedTime);
	ai->econTracker->UnitFinished(unit);
	if(ai->ah->CanHandleThisUnit(unit)) {
		ai->ah->AddUnit(unit);
	}
	else if((ai->cb->GetCurrentFrame() < 2 || ai->cb->GetUnitDef(unit)->speed <= 0)) {//Add
comm at begginning of game and factories when theyre built
		ai->uh->IdleUnitAdd(unit);
	}
	ai->uh->BuildTaskRemove(unit);
	ai->math->StopTimer(unitFinishedTime);
	ai->math->StopTimer(totalSumTime);
}

void CGlobalAI::UnitDestroyed(int unit,int attacker)
{
	ai->bu->MexUpgraders.remove(unit);
	ai->math->StartTimer(totalSumTime);
	ai->math->StartTimer(globalAILogTime);
	L("GlobalAI::UnitDestroyed is called on unit:" << unit <<". its groupid:" << ai-
>MyUnits[unit]->groupID << " name:" << ai->MyUnits[unit]->def()->humanName);
	ai->math->StopTimer(globalAILogTime);
	ai->math->StartTimer(unitDestroyedTime);
	ai->econTracker->UnitDestroyed(unit);
	if(GUG(unit) != -1) {
		ai->ah->UnitDestroyed(unit);
	}
	ai->uh->UnitDestroyed(unit);
	ai->math->StopTimer(unitDestroyedTime);
	ai->math->StopTimer(totalSumTime);
}

void CGlobalAI::EnemyEnterLOS(int enemy)
{
}

void CGlobalAI::EnemyLeaveLOS(int enemy)
{
}

void CGlobalAI::EnemyEnterRadar(int enemy)
{
}

void CGlobalAI::EnemyLeaveRadar(int enemy)
{
}

void CGlobalAI::EnemyDestroyed(int enemy,int attacker)
{
}
```

```
void CGlobalAI::UnitIdle(int unit)
{
	ai->math->StartTimer(totalSumTime);
	ai->math->StartTimer(globalAILogTime);
	L("Idle: " << unit);
	ai->math->StopTimer(globalAILogTime);
	ai->math->StartTimer(unitIdleTime);
	ai->econTracker->frameUpdate();
	//attackhandler handles cat_g_attack units atm
	if(GCAT(unit) == CAT_G_ATTACK && ai->MyUnits.at(unit)->groupID != -1) {
		//attackHandler->UnitIdle(unit);
	} else {
		ai->uh->IdleUnitAdd(unit);
	}
	ai->math->StopTimer(unitIdleTime);
	ai->math->StopTimer(totalSumTime);
}

void CGlobalAI::GotChatMsg(const char* msg,int player)
{
}

void CGlobalAI::UnitDamaged(int damaged,int attacker,float damage,float3 dir)
{
	ai->econTracker->UnitDamaged(damaged, damage);
}

void CGlobalAI::UnitMoveFailed(int unit)
{
	if (ai->cb->GetUnitDef(unit) != NULL) { //habit :P
		ai->MyUnits[unit]->stuckCounter++;
	}
}

int CGlobalAI::HandleEvent(int msg,const void* data)
{
	// Unit steal and donate support:
	L("msg: " << msg);
	if(msg == AI_EVENT_UNITGIVEN)
	{
		const IGlobalAI::ChangeTeamEvent* cte = (const IGlobalAI::ChangeTeamEvent*) data;
		if(cte->newteam == ai->cb->GetMyTeam())
		{
			// Just got a unit
			UnitCreated(cte->unit);
			UnitFinished(cte->unit);
			ai->MyUnits[cte->unit]->Stop();
		}
	}
	else if(msg == AI_EVENT_UNITCAPTURED)
	{
		const IGlobalAI::ChangeTeamEvent* cte = (const IGlobalAI::ChangeTeamEvent*) data;
		if(cte->oldteam == ai->cb->GetMyTeam())
		{
			// Just lost a unit
			UnitDestroyed(cte->unit, 0);
		}
	}
	L("msg end");
	return 0;
}

// THE MAIN AI UPDATE FUNCTION, THE TOP LEVEL AI CODE
void CGlobalAI::Update()
{
	ai->math->StartTimer(totalSumTime);
	int frame=ai->cb->GetCurrentFrame();
	ai->math->StartTimer(globalAILogTime);
```

```
//     L("start: " << frame);
       ai->math->StopTimer(globalAILogTime);
       ai->math->StartTimer(updateTimerGroup);

       ai->math->StartTimer(econTrackerFrameUpdate);
       ai->econTracker->frameUpdate();
       ai->math->StopTimer(econTrackerFrameUpdate);

       if(frame%60 == 1){
              ai->math->StartTimer(updateTheirDistributionTime);
              ai->dc->UpdateTheirDistribution();
              ai->math->StopTimer(updateTheirDistributionTime);
       }
       if(frame%60 == 20){
              ai->math->StartTimer(updateMyDistributionTime);
              ai->dc->UpdateMyDistribution();
              ai->math->StopTimer(updateMyDistributionTime);
       }
       if(frame%60 == 40){
              ai->math->StartTimer(economyManagerUpdateTime);
              ai->em->Update();
              ai->math->StopTimer(economyManagerUpdateTime);
       }
       if(frame > 80){
              ai->math->StartTimer(threatMapTime);
              if(frame%15 ==0)
                     ai->tm->Create();
              ai->math->StopTimer(threatMapTime);

              ai->math->StartTimer(builUpTime);
              ai->bu->Update();
              ai->math->StopTimer(builUpTime);
              ai->math->StartTimer(idleUnitUpdateTime);
              ai->uh->IdleUnitUpdate();
              ai->math->StopTimer(idleUnitUpdateTime);
       }
       ai->math->StartTimer(attackHandlerUpdateTime);
       ai->ah->Update();
       ai->math->StopTimer(attackHandlerUpdateTime);

       ai->math->StartTimer(MMakerUpdateTime);
       ai->uh->MMakerUpdate();
       ai->math->StopTimer(MMakerUpdateTime);
       //int count = ai->sh->GetCountEnemiesFullThisFrame();
       //if(count > 0)
       //     L("GetCountEnemiesFullThisFrame: " << count);
       float updateTime = ai->math->StopTimer(updateTimerGroup);
       ai->math->StartTimer(globalAILogTime);
//     L("end: " << frame << ", updateTime: " << updateTime);
       ai->math->StopTimer(globalAILogTime);

       ai->math->StopTimer(totalSumTime);

       if(frame == 1){ // Dont include the setup time of defenseMatrix in the total
              //ai->math->StartTimer(defenseMatrixInitTime);
              ai->dm->Init();
              //ai->math->StopTimer(defenseMatrixInitTime);
       }
       // Print the times every 2 mins
       if(frame % 3600 == 0 && frame > 1)
       {
              L("Here is the time distribution after " << (frame / 1800) << " mins");
              ai->math->PrintAllTimes();
       }
}
```