# Parallel Agent Models for a Realtime Strategy Game

Bachelor thesis at the Faculty for Electrical Engineering / Computer Science
at the University of Kassel

July 8, 2008.

Submitted by:
Dennis Keßler

Supervisor:
Dipl.-Inform. Björn Knafla

Examiners:
Prof. Dr. Claudia Leopold
Prof. Dr. Albert Zündorf

Department for Electric Engineering / Computer Science
Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel

## Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

________________________________ Kassel, der 8. Juli 2008.

# Abstract

This bachelor thesis discusses two different aspects related to realtime strategy games. The first presented aspect is the application of behavior trees for controlling agents. This includes a comparison to the currently used technologies, hierarchical finite state machines and script engines. The second aspect is the parallelization of the simulation for massive multiplayer online realtime strategy games. Therefore different approaches for parallelizing of the simulation are explained. Some of these approaches were implemented in Java and are compared to each other, showing the different problems of the parallelization. As a result of these comparisons, a distributed architecture for a simulation is introduced.

# Contents

# 1 Introduction

Non-realtime games, like chess or round-based strategy games, can stop their visualization and other processor-consuming tasks to allow complex calculations for their strategy without disappointing the player. Realtime games on the other hand cannot stop or pause their visualization without disturbing the gameplay. But with the broad availability of multi-core CPUs, more spare time is available for using more sophisticated artificial intelligence routines and models.

This thesis examines different topics which are relevant to a massive multiplayer online realtime strategy game. Of particular interest are the application of artificial intelligence in realtime strategy games and the application of parallel computing techniques to the simulation of such a game. The ideas and implementation were realized within the ongoing research project jMMORTS [jMMORTS], which aims to implement a complete massive multiplayer online strategy game in the Java programming language. At the same time, this project serves as a research platform for various technologies related to realtime strategy games.

In chapter 2 the relevant background and the basic structure of a realtime strategy game are presented. Chapter 3 gives an overview of the currently applied techniques for artificial intelligence in realtime strategy games and compares them to a new approach which has not yet been realized in such a game. Chapter 4 introduces the requirements for a data model, capable of simulating a large number of entities assigned to many individual players. Different parallelized implementations in the Java programming language are discussed and compared to each other. The presented results are discussed and another solution, which uses a distributed data model, is outlined. Chapter 5 summarizes the results of chapters 3 and 4 and discusses possible further uses of the presented artificial intelligence techniques.

# 2 Overview of Realtime Strategy Games

This chapter defines the basic terms of this work and gives an overview over the game mechanics used in realtime strategy games, starting with the general idea of this genre. Based upon this, a general model for agents is presented, followed by an overview of how a simulated world for such a game can be modeled.

## 2.1 Realtime strategy games

The genre of realtime strategy games, commonly abbreviated as RTS, is relatively new compared to other genres, like round-based strategy games or adventure games. One of the first RTS games was Dune II by Westwood Studios, published in 1992 [Dune2k]. Another important example is the game Warcraft by Blizzard Entertainment[Blizzard], published in 1994.

As presented in [Troy2005, GameSpot, IGN06] common to all RTS games are two or more factions in war with each other. A typical RTS game supports two different types of gameplay. One type is the so-called single player game where the player chooses on which side to battle against computer player(s) and works through a number of missions until success. Each mission is usually played on a map of fixed size with a given entry point. The second type of gameplay is the so-called multiplayer game where typically two to sixteen players, accompanied by computer-controlled players, play on one map. The goal of a multiplayer is usually to eliminate all other participating players.

During the game, the player usually has a bird's eye view on the scenery and controls units like soldiers or tanks with the mouse. Different unit types are distinguishable by their graphical representation. Units of the same type have the same representation, except for the visualization of their health conditions.

Common for most RTS games is an abstract view on processes like obtaining new soldiers. Instead of recruiting and training a soldier, the process is described as a manufacturing process. The player uses a build button, which starts a countdown. When the countdown is finished, a new soldier appears in front of the associated building. Buildings are usually placed on the map by the player and appear without further micromanaging after a certain amount of time.

Most games concentrate on the different types of moveable units. Typically there are three functional types of units available for creation:

*Basic units*

Mostly a moveable unit that can transform to a headquarter. This building is usually a precondition to build other units or buildings.

*Support units*

Support units are used for constructing new buildings, repairing other units, or collecting resources.

*Battle units*

Battle units are the core of the gameplay, usually there are a lot of different units, e.g. various types of tanks, soldiers, ships, aircrafts, and so on.

In most cases, a mission can be divided into different stages:

*Orientation*

In this first stage of a mission the player is unaware of the map's details, oftentimes a large portion of the map is unrevealed. The player needs to find a suitable location for the base and also needs to discover where required resources are located.

*Base building*

Most games require that certain buildings have to be built in order to be able to produce more units. The production of units and buildings is limited through a shortage of required resources, like for example gold, lumber, or minerals, which the player has to find, collect, and transfer to the base. Therefore this phase usually starts with setting up the needed infrastructure, gathering resources and exploring the area surrounding the base. Defending the base against enemy units, with the given units or already produced units, is often part of this stage.

*Exploring and producing*

After the required infrastructure is ready and more units get produced, the player starts to explore more areas of the map and simultaneously accumulates battle units in and around the base. In many cases this phase includes defending the base against waves of hostile units and also the discovery of one or more enemy bases.

*Assault*

When the player has accumulated enough units for the main assault, the mission is often solved through a massive final battle between the accumulated armies, involving a lot of micromanagement of the battling units.

Often the different missions are connected with the narration of the game's background story. Possible variations from the mission prototype presented above are for example the protection of an existent base, the survival of special units, surviving on a map without a base, or fulfilling certain specific mission goals. With further progression in the game, more unit types are available for production, and the complexity and intensity of the battles increase.

In multiplayer games the players start at the same time at different locations on the map. The stages of gameplay are essentially the same, but the options - like the amount of resources on start - are often freely configurable.

Noteworthy exceptions are games like Age of Empires by Ensemble Studios [Ensemble Studios] or Empire Earth by Sierra Entertainment [Sierra], which do not have missions. Instead the game starts at one point of time in history, like Stone Age. In these games, the player is in need of long-term planning, upgrading, and expansion through the different ages of mankind including future settings. Typically the number of parties is higher in this type of game and additionally there are possibilities for diplomatic relationships between the opponents.

## 2.2 Agents

Everything in a game that is of interest to a player is referred to as an entity. The subset of entities, directly controllable by players, is called agents. Fundamental differences between agents, like the differences between soldier types and aircraft types, can be expressed by a class hierarchy for the agents.

Each agent class shares the same functions, like a function for moving or ballistic flight. Different agent types within an agent class are expressed through different basic attributes. These attributes include information like maximum speed, turn rate, or firepower. Usually all instances of one agent type are share these attributes. All tanks for example are share the same drive function, but use agent type specific values for their maximum speed. An exception are games where agents are rewarded for each survived mission or enemy, and some of the attributes are increased for a specific agent.

Another type of information are the instance-specific attributes, like current angle of view, current location, or direction of movement. These attributes are unique for every instance of an agent class.

The third type of information are the context-specific information of an agent. This type of information usually needs the class- and instance-specific attributes of an agent, like current position and angle of view. These attributes are used by context-specific functions, which calculate the context-specific information. A typical function is for example the field of vision. In this function the current position and angle of view are combined with the range of view attribute and these combined attributes are applied to the actual map for retrieval of the current visible entities for the given agent.

In terms of CPU usage the class attributes are the cheapest ones to retrieve, closely followed by the instance specific attributes. The context-specific information, also often called context functions or sensors, are the most expensive information to retrieve, as often costly calculations are involved. Additionally, other entities and their attributes might be involved in the calculations, which increases the complexity of the calculations.

## 2.3 Actions

For the modification of the agent's attributes, so-called actions are used. An action is an abstract view of the actual change and contains the changed attribute and the new value for that attribute. How the concept of actions is realized within a game is very game-specific. Usually the action is either some sort of data structure, applied during the simulation of the agent, or a simple function call to a specific function of an agent.

Based on the used attributes it is possible to define the actions for an agent and to classify the action as a basic or advanced action in terms of CPU usage:

> *Basic actions:*
>
> Basic actions use mostly class- and instance-specific information for their calculations. Examples for such actions are "move left", "turn right", or "turn field of view clockwise". The needed CPU usage is mostly low, as the involved information are usually only class- or instance-specific. The basic actions are mostly used by advanced actions and completely hidden from the player. This is a major difference to other game genres like first-person shooters, where the player has total control over one agent's basic actions.
>
> *Advanced actions:*
>
> In a typical RTS game, the advanced actions are exposed to the players. Examples for advanced actions are "destroy the given target", "move to a specified locatio", or "follow another entity". Most of that functionality is accomplished by using several basic actions and additional information of context-specific attributes. The advanced actions can be used by an artificial intelligence as well as by a player. Within the user interface of an RTS game, the player usually selects one or more of the owned agents and selects which advanced actions to use.

Within the RTS game genre, a typical abbreviation for assigning actions to an agent and the execution of these actions is commanding an agent. As the RTS genre uses in most cases a third-person view, an action is usually called a command.

## 2.4 Terrain model

A typical map is subdivided into cells. Depending on the game a cell can be occupied by one or more agents. Most RTS games use 2.5D maps. This technique applies height values to a regular grid. Additionally, information about the type of ground and visualization-specific details can be added to each point in the grid.

Another widely used technique is the utilization of height layers. Each layer is represented by a grid of values, indicating if a particular cell can be occupied by an agent. Ramps are used for moving between the layers.

The most important advantage of height layers over the employment of height values is the ability to simply represent structures like bridges, where one agent can be located under and another agent on the bridge simultaneously. With height values the use of special constructs within the simulation is needed to achieve this feature. The disadvantage of height layers over the application of height values is a reduced level of detail for the simulated game physics, because of the fixed height layers and the defined ramps between the layers. The addition of more height layers can reduce that drawback, but simultaneously increases the memory and disk usage for a map.

# 3 Behavior concepts

Since the first RTS games, some sort of AI (AI is the common abbreviation for artificial intelligence) has been used to simulate opponents for the player. Although it is commonly called AI, the games use some sort of software component for decision-making, not a real AI as defined in [Russel02].

For the different AI techniques used in RTS games, two rules or requirements can be stated:

*Believable gameplay*

The AI has to obey the same rules as the player. Therefore the agents of the computer-controlled player are required to have the same attribute values as if the player would own them.

*Continuous gameplay*

The computer-controlled player should react and act in a continuous way. A chosen tactic of the computer-controlled player should for example not be changed too frequently.

An opponent who acts believably and continuously increases the ability of a human player to adapt to the strategies used. Additionally, the replay value of a game can be enhanced, as a human player can experiment with different strategies in a well-defined environment.

To achieve such an AI behavior, different technologies have been used and mixed together. The following sections describe these technologies and explain the RTS-game-specific integration and use within an RTS game. The first two sections will introduce the script-based AI and the hierarchical finite state machine approach. The third presented technology is the behavior tree, which has not yet been used within an RTS game.

The AI technologies using scripts or hierarchical finite state machines have been sufficient for the past years in terms of reacting mostly predictably to the player's actions. But the ability to adapt to the strategy chosen by the player is still limited today. This challenge can be overcome by an increased complexity within the AI scripts or hierarchical finite state machines. At the same time the process of development, testing, and fine-tuning may become more complex, too. This further leads to more risks in the development process and may increase costs at the same time.

As a continuos example, a simple behavior for a fictional soldier is used. Like most agents, the fictional soldier has independent subparts, like the body and head. The body can walk at the same time as the head can turn. In a typical situation this soldier is ordered to defend a certain target position. This requires that the soldier moves to this position while constantly looking around for enemies. If an enemy is spotted, this enemy should be attacked, which may require moving towards the enemy. A possible implementation of this behavior is shown for each AI-approach. Based on these examples, the advantages of the behavior trees over the other two approaches are explained afterwards. The chapter concludes with the presentation of further additions to the behavior trees and how behavior trees can be used in the context of realtime strategy games.

## 3.1 Scripted AI

The scripted AI usually consists of a textual description how to act in the mission, like a screenplay. In the script, commands are used to direct the controlled entities. These commands invoke the advanced functions of the controlled entities. Within the script, the execution is typically controlled with well-known constructs, like loops and conditions. So-called triggers are often employed for implementing the conditions. A trigger is a context-sensitive condition, usually bound to a specific location of the map. An example for such a trigger might be: When a tank of the human player is at the entrance area to the computer-controlled base, start building more close-combat anti-tank units and order these to attack the tank.

With the application of a script, the computer-controlled player has only the hard-coded options within the script to react to the human player. This fully satisfies the second rule of continuous acting, but might result in a boring gameplay for the human player, especially if the more or less same base script is used for every mission in the game. A famous example for that is the AI in Command and Conquer [C&C]. A player can block every command of the AI by building a wall into the AI-controlled base. Once the entrance to the base is sealed, the AI stops to function in a meaningful way and makes overtaking the AI-controlled base very easy. This sort of gameplay is considered cheating although the human player has obeyed all rules of the game, because the AI is not able to react properly to the situation.

A script, realizing the desired behavior for the soldier example, might look like this:

```
state = IDLE
targetPosition = ( get my current body position )
```

```
while(true)
{
    switch(state)
    {
        case IDLE:
            if ( targetPosition not equals ( get my current body position ) )
             do ( walk towards targetPosition )
do ( turn my head )
            if ( i see an enemy )
                state = ATTACK
            else
                do ( turn my head )
            break

        case ATTACK
            if ( i see an enemy )
                do ( look straight at enemy )
                if ( enemy in weapon range )
                    do ( attack enemy )
                else
                    state = WALK
            else
                state = IDLE
            break

        case WALK
            if ( i see an enemy )
                if ( enemy in weapon range )
                    state = ATTACK
                else
                    do (  walk towards enemy )
            else
                state = IDLE
            break
    }
}
```

## 3.2 (Hierarchical) Finite State Machines

To overcome the limitations of the static scripted AI, the concept of finite state machines was introduced to RTS games. A finite state machine consists of connected states. A state is usually a simple script or a single command. The connections are directional. Each connection can include checks. A check is a function which returns to which percentage this connection is currently valid to be chosen. An example check is a function which returns zero percent if no enemy entities are visible and otherwise one hundred percent. At the start of a mission, a defined state is entered. When all commands within a state have finished their execution, or a command was not applicable, one outgoing connection is chosen. Therefore the checks of all outgoing connections are executed and the best matching connection is followed.

An AI built with a finite state machine has considerably more options to adapt to the player's strategy and react to unforeseen situations. Additionally the AI conforms to the requirement of strategy games to react continuously, as the same connections are chosen again, if the relevant parts of the context match.

Building and describing a large AI with states and connections can result in large and complex constructs. Therefore an extension to finite state machines was introduced. The HFSM (HFSM is the common abbreviation for hierarchical finite state machines) is a layered finite state machine.

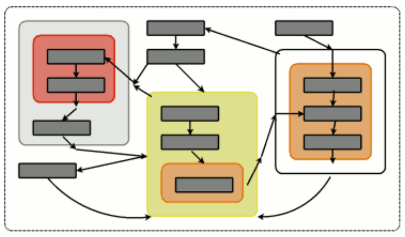

Figure 3.1: Example picture of an HFSM [AiGameDev]

In HFSMs multiple levels of finite state machines exists. As the figure 3.1 shows, states can have sub-states. A group of sub-states is represented as one state in the upper layer of the state machine. This is the hierarchical component of a HFSM. The connections can now start and end at a state or a group of states. Additionally sub-states can be connected to their surrounding state. With this technique the behavior can be grouped into related blocks like idling or attacking, and the amount of connections can be reduced drastically. Depending on the whole architecture, it may be possible to reuse some levels of one HFSM for other agents' HFSMs.

The current state in the industry, as reports on [Gamasutra] indicate, is the combination of scripts and HFSMs. Usually scripts are employed for the story-narration and overall progress in the game. For the management of entity groups and single entities, HFSMs are used.

The mixture of both techniques allows for continuous gameplay for the player and a high replay value, because of the possibility for adaption within the AI. Further, development and integration of such techniques into RTS games is easy today. Many commercial game engines already offer implementations and editors for one or both AI techniques.

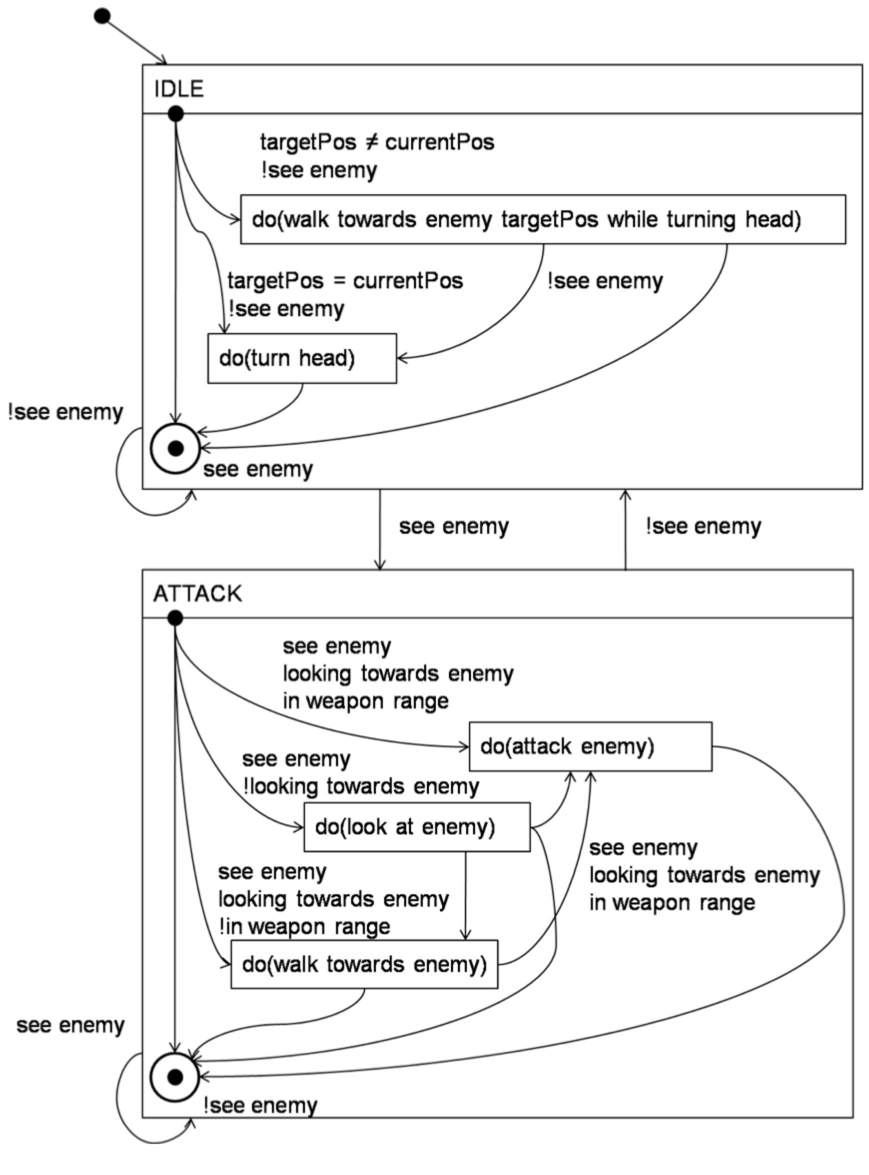

Figure 3.2: Example HFSM for the soldier example

An example HFSM for the soldier example might look like figure 3.2.

## 3.3 Behavior Trees

This introduction to behavior trees is based on the explanations of the game programming freelancer Alex J. Champandard's writings on AiGameDev.com [AiGameDev] and publications from the Damián Isla [Isla, Isla05]. An early form of a behavior tree has been mentioned by Chris Butcher and Jaime Griesemer of Bungie in [GDC02-halo] for their first-person shooter game Halo, announced in 2001. Today, the first-person shooter Halo 2 is widely used as a reference for behavior trees [Isla05, GDC07]. Various recent first-person shooter games seem to use the Halo 2 approach.

### 3.3.1 Basic structure

As figure 3.3 shows, a behavior tree consists of connected nodes. It has many characteristics in common with a hierarchical finite state machine. Whereas in HFSMs cyclic connections between state-nodes are possible and widely used, the nodes of a behavior tree are organized hierarchically. No cycles are allowed.

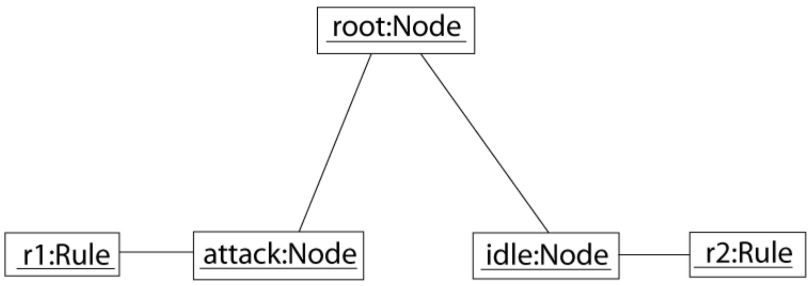

Figure 3.3: Simple behavior tree.

The origin of a behavior tree is called the root. Every other node is, either directly or indirectly, a child of this root. Nodes with the same distance from the root, measured by the number of intermediate nodes, are called a layer. A node without children is called a leaf node or in short a leaf. The leaves in a behavior tree contain the commands to be executed. The nodes located directly between the root and a certain leaf constitute the so-called path, which is important for structuring and decision-making, e.g. to decide on which leaf is to be executed next.

The nodes on a path to a leaf do not contain commands. Those intermediate nodes hold the so-called rules, with each node containing one or more rules.The components of a rule are: an attribute or sensor input of the agent, evaluated in the current context (see chapter 2.2), an operator like *equals*, *lesser*, or *greater* and the defined comparison value. Therefore a rule can be expressed as a mathematical function, either returning simply a boolean value which indicates if the rule is fulfilled, or a numerical value which indicates to which percentage the rule is fulfilled.

For the structuring of subtrees, Champandard suggests four patterns for behavior tree nodes [AiGameDev]:

*Parallel execution*

The parallel execution pattern executes all subtrees whose rules are fulfilled. In general, parallel execution does not mean a real parallel execution, but further evaluation of all fitting subtrees sequentially.

*Choice*

The choice pattern recommends the choice of the best-matching or first matching subtree, depending on the details of the implementation.

*Sequence*

On completion of one subtree, the sequence pattern proposes the choice of the next defined subtree which allows the ordered execution of commands.

*Random*

The random pattern randomly selects one subtree for execution.

The most frequently used is the choice pattern, as it functions like an if or switch statement within the tree.

An exception is the root node, which contains no rule and is realized as a choice node. During the decision-making, one or more branches of the tree are selected, based on the evaluations of the encountered rules and the node types.

An example behavior tree for the soldier example is shown in figure 3.4.

### 3.3.2 Advantages over script or HFSM-based systems

Generally, the functionality of a script, HFSM, or behavior tree can be expressed by or added to each other. However, the design of the particular engine type has to be extended to support all features of the other types. Such an extension usually introduces additional complexity within the engine and can therefore increase the development time and costs. At the same time, as the complexity of the engine is increased, the complexity for the use of the engine increases, too.

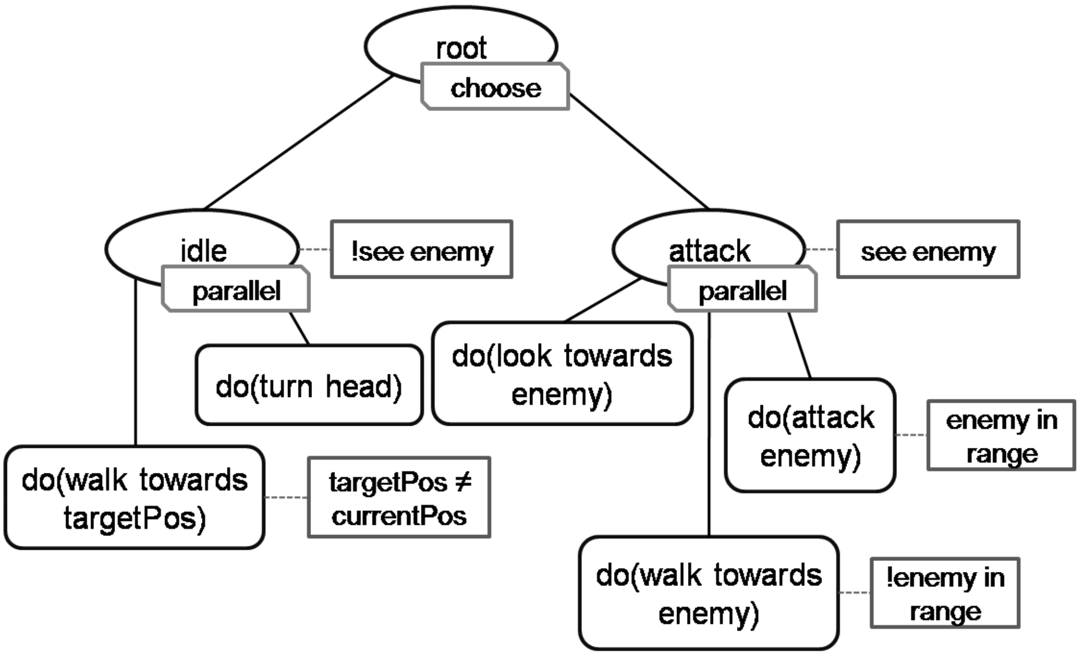

Figure 3.4: Example behavior tree for the soldier example

A comparison of the features and ease of use of each engine may look like this:

*Script engine*

The Script engine is easy to extend by adding new commands or language constructs. An extension is unlikely to break existing code. However, writing scripts is the least abstract way to define behavior. It can be compared to writing source code. Therefore, AI designers need to have at least basic programming knowledge. Changes to existing code can be challenging, too, as the understandability of the script depends on how well it is written.

*HFSM engine*

HFSMs are generally more abstract than scripts, especially when they are represented in a graphical way. During an extension of a HFSM engine it might be necessary to check all old scripts, because of the abstract representation. Extensions have to modify exiting nodes or connections or introduce new types of nodes or connections. Therefore the design of a behavior can become more challenging, depending on the provided editor, as the designer has to choose between different node types or connection types or has to set specific options on nodes or connections to achieve the desired behavior. Large behavior graphs can be challenging to edit, due to the number of needed nodes and connections.

*Behavior tree engine*

With a behavior tree the most structured representation is possible. Both representation forms, either textually or graphically, can be easily read without programming knowledge. Understanding of a complex behavior is generally easier than understanding a HFSM, as no cyclic connections are allowed and everything is strictly hierarchically ordered. Generally no extensions are needed, as most of the time all behavior can be expressed with the four types of nodes, presented in 3.3.1. The challenge for the AI designer thus lies in grouping relevant commands into subgraphs and defining the connection rules between the nodes. In the case of an extension, old behaviors should work without further change, as an extension introduces either a new node type, a new connection type, or a new rule.

One distinct advantage of a behavior tree engine over a HFSM engine lies, for example, in the parallel execution node. With this node it is possible to execute multiple subtrees at the same time, which is not a commonly available feature in a HFSM. This helps to reduce the minimal number of nodes and connections as the following example shows:

As figure 3.2 shows, every if-statement of the scripts translates into a substate within the HFSM. The substates and states are connected, guarded by the condition of the if-statements. Since a basic HFSM-implementation executes all commands within a node, optional commands result in additional substates. For example the “walk to targetPosition” in the *IDLE*-state results in the need for additional substates. One substate is needed for each combination of the two if-statements.

This complex HFSM-graph can be expressed as a much simpler behavior tree as figure 3.4 shows. With the selector and parallel nodes, every construct seen in the script is possible again. However, the designer is now required to make sure that not two opposing commands are using the same body-resource of the agent at the same time, like “walk towards initial position” and “walk towards enemy” due to a missed rule.

## 3.4 Further extensions to behavior trees

As various authors discuss [Isla05, Lent99, Guy99], the presented behavior trees might be too static, in terms of response time to rapid changes in the context of an agent. Their common solution introduces the use of stimuli.

Like the human body reacts to stimuli, this solution gives the agents’ sensorial input the possibility to trigger a stimulus. Such stimuli influence the selection probability of rules or parts of rules in the agents’ behavior tree and therefore are able to change the result of the decision-making dynamically, hiding normally chosen nodes or prioritizing previously unsuccessfully checked nodes even if the current context normally would not allow their selection. After a defined time the stimuli are either simply removed or their influences are reduced to zero over a period of time.

This technique cannot only be applied to individual agents, but also to groups of agents. One problem in group control remains the assignment of a task or command to a specific, e.g. best fitting, agent. The classical solution is to manually activate commands on the target agent. However, this approach introduces some problems within a behavior tree. In particular, how does a manual command activation influence the decision-making process of this agent? Existing answers to this question, like the use of a bypass to the AI to execute the command or risking abortion of the command execution within the next decision-making, are not satisfactory. A fitting solution is the use of blackboards [Orkin05, AiGameDev]. A blackboard is an information store, shared by all members of a group. The grouped agents can access the storage as part of their context during the decision-making. The stimuli of the agents and the group AI can write directly to the blackboard, thus enabling each agent within the group to react within a short time, but with consideration of the agent's current context to the task the group has to execute. Additionally, the selection of the executing agent is done automatically, as each agent decides if the tasks fits the current context. To ensure that only one agent at a time executes a given task, additional information might have to be integrated into the blackboard.

Another extension to further refine the concept of behavior trees is the possibility, mentioned in [AiGameDev], to combine a behavior tree with any other form of AI code like HFSMs or scripts. This extension is simply done through the introduction of specialized leaf-nodes, containing the necessary information for the underlying AI-engine to execute properly. For an AI designer, the use of these specialized nodes is transparent within the behavior tree.

## 3.5 Behavior trees in RTS games

In a complete RTS game, different layers of decision-making exist. Typically there are four layers of decisions necessary:

- *Story*

  If the game supports a dynamic story, based on the results of the different missions, this component is responsible for storytelling and mission selection or generation.
- *Mission*

  Within a mission, the computer-controlled player(s) have to decide which general actions to perform, like building a base, scouting the map, or attacking the player.
- *Entity group*

  In most cases, every player has to group the controlled entities for the different tasks within a mission. For example different attack and defense groups, which have to be controlled independently from other groups or single entities. The AI for an AI-controlled player therefore has to decide which entities have to group together, which entities should be built, and which tasks a group has to fulfill.

- *Entity*

  Every single entity needs individual commands, even if the entity belongs to a group. This so-called micromanagement has to be done often and is influenced by the task of the group if the entity is grouped.

From the top to the bottom the decisions within that layer have to be made more frequently and in a more detailed way. This results in more complex scripts or HFSMs for each layer. At the same time the process of decision-making has to be less resource-intensive in terms of CPU utilization. Additionally the resource demand should be lower on each layer, as the game has one AI-instance for the story, one AI-instance for a mission, one AI-instance for each group and maybe one instance for each individual entity.

Currently, articles on [Gamasutra, AiGameDev] indicate that for the story and mission scripts are used. HFSMs are commonly used for the group and entity AIs. As shown in section 3.3.2, behavior trees can replace all the currently used techniques on all layers. However, the last two layers are of particular interest, as their complexity is higher.

For the group control the application of a behavior tree engine can solve the continuous gameplay problem, mentioned in 3, as well as the adaption problem. The group AI writes tasks or information which lead to tasks to the blackboard. The agents within the group execute the tasks or react to the information on the blackboard as far as it fits the individual context. At the same time, both group and agent AI have the ability to react to rapid changes within the game.

As shown in 3.3.2, the process of designing and fine-tuning a behavior tree can be much easier for an AI designer than it would be with scripts or HFSMs.

# 4 Parallel simulation models

This chapter introduces the basic concepts of the simulation for an RTS game. Based on this presentation different possible approaches for parallelizing the simulation and the data model of such a simulation are shown. The second section of this chapter explains the challenges encountered while implementing various approaches with the Java programming language. Afterwards, the results obtained from the different implementations are compared and some conclusions mostly specific to Java are drawn.

Basically, the most interesting parts for parallelization within an RTS game are the simulation and the visualization. As this thesis is about technologies for an MMORTS game, a server application is used for the simulation. The clients are connected to this server and send their actions/commands to the server, while receiving updates on the currently running simulation. There are at least two methods to achieve parallelization in the simulation. The first method, parallelizing the calculations of the simulation, is discussed in the next sections. Another method is the distribution of the different simulation aspects to different, communicating applications, not necessarily located on one server system. A possible implementation for such a distributed approach is outlined at the end of this chapter. The aspects of parallelizing the visualization is not part of this thesis.

## 4.1 Basic concepts

An RTS game can be broken down into different components, such as simulation, graphics and sound. Each component interacts with other components and can further be broken down. In a typical RTS game all of the components are included in the game application. In a massive multiplayer RTS game the components are usually distributed to different applications running on different systems, like the game client, which runs on the player's system, or the server application, which manages the whole simulation and typically is located at the game provider's location.

The fundamental component of an RTS game is the simulation. Every entity within the game is part of the simulation. Generally the simulation is in one of the three states: initialize, run, or shutdown. Initialization and shutdown are left out in this thesis, as they contain mostly only the creation and cleanup of other necessary components. The run state, also known as the main loop, drives the whole game. In a classical setup, with all components running locally, an iteration of the main loop consists of three steps:

- *Input*

  During the input step all input sources are polled for new events. Input sources are usually mouse and keyboard, but also the network, if the game supports multiplayer game modes. Generally the input is not presented as raw mouse clicks or key hits, but rather in the form of an event. The events are applied to the simulation which might result in changes, or the creation or deletion of entities.

- *Process*

  After the input handling the entities of the simulation have to be updated. This updating of an entity may include decision-making, pathfinding, moving on the map, interactions with the surroundings, moving, or even firing bullets. Most of the actions have to be done individually for each entity, some, like pathfinding, might be done for groups of entities. Collision detection must be done for all entities, as this involves all of them.

- *Output*

  When the process step has finished, the changes, like movements, destructions, or creations of entities have to be visualized to the player. In a multiplayer game, changes must be shared with other players, too.

This main loop has to be run at a certain rate, often 10 to 30 times per second, to achieve an immediate feedback to the player. For multiplayer games the problem of synchronization can be solved either directly through a defined master system which holds the valid state of the world, or indirectly through an agreement of the participating applications about the valid world state.

In both solutions, the amount of calculations might be too high to be performed by each participating system, especially if the game is a massive multiplayer game which implies large armies for each player and many players on a map. Most current games impose many restrictions, like the amount of controllable entities per player and the maximum number of allowed players on a map. Further, most of those maps are designed to confront all players with constant action, thus further limiting the amount of entities. Moreover, the maps are generally optimized for pathfinding, as pathfinding is one of the most resource-consumptive tasks in such a game.

One approach to the problem of simulating a lot of entities is the use of a central server. This server accomplishes the required calculations and communicates with the clients, receiving their events and sending them updates for their visualization. This approach reduces a lot of the overhead, but still is limited to one CPU executing the main loop which takes care of the whole simulation. Therefore, the next step is the parallelization of this main loop. To discuss the possible approaches for parallelization, the parts of the simulation which might be parallelized have to be determined.

In chapter 2.2, the concept of entities and their specialization, the agents, was introduced. Based on this, the entities and agents are revisited and are now described as a hierarchical construct of components:

*Sensors*

A sensor is a component which reads from the context of an entity and makes the information available internally. This may include the construction of the necessary data which form the context, or the evaluation of previously unused data, or simply the returning of an already present value.

*Actuators*

Actuators are functional components of an entity which change externally relevant attributes, like the position of the entity, or generate new simulation entities, like a gun releases a bullet upon firing.

*Advanced functions*

Advanced functions aggregate functionality of actuators and sensors to higher level functionality. An example for such functionality may be a “shootAt” function of a tank agent, which takes care of being in firing range of the target, turning the gun towards the target, and the actual firing, including the ballistic calculations for the shot. An example for an expensive planner, in terms of CPU use, is a “moveTo” function, involving a pathfinder.

*AI*

The AI is the highest level component of an entity, usually only found in an agent. Regardless of the actual AI implementation, the AI uses the functions of the other components for decision-making and commanding the entity.

Mostly a simulation in an RTS game can be seen as a composition of two elements:

*Agents and entities*

Most entities, especially the subset of the agents, need to update frequently. Within this update, the entity executes its current commands. Usually, if the entity is controlled by an AI, the AI’s decision-making process is executed during the update, too.

*Physical simulation*

The physical simulation in RTS games is mostly about collision detection. Each entity in the simulation possesses attributes like the position or the bounding volume of the entity. During an update, entities are moved in the simulated world and therefore must be checked for collisions, including the impact calculations in the event of a collision and the resulting changes within the collided entities. The collision detection needs to compare all the entities' positions and bounding volumes after each change to reliably detect collisions. As this would result in a search problem with the computational costs of $O(n^2)$, many techniques for optimizations exists [Bobic00].

The shown breakdowns of the internal structure of an entity and the simulation lead to different strategies for parallelizing.

- Parallelizing the entity updates within a simulation turn:

  During a turn, all entities are updated in parallel. The physical simulation can then be done either after all entities have updated, or concurrently whenever an entity changes its physically relevant attributes.

  If the physical simulation is executed after all entities are updated, an additional step for all entities, affected by the physical simulation, is needed for incorporating of these changes.

  If the physical simulation is executed after the update of every entity, the order in which the entities are updated may change the results of the simulation. For example, during one simulation turn, a bullet will hit a tank critically, at the same time the tank shoots one of its bullet. If, on the one hand, the bullet is simulated first, the tank will not be able to perform the shot, because of its destruction. On the other hand, if the tank is simulated prior to the bullet, it can perform its shot before being destroyed by the bullet. This sounds like a critical problem, but in the context of games this behavior might be tolerable in opposition to a scientific simulation.

- Parallelizing the different entity components:

  This approach is based on the idea of data locality. The data locality is achieved with the concept of distributed entities. Instead of a single structure, containing all information about an entity, the entity exists within the different simulation components. Each simulation component contains thereby only the relevant data for each contained entity.

  For example, each entity consists at least of a colliding component, which takes care of the physical interaction between entities. Therefore, the physics simulation component owns the bounding volume and location for each entity within the simulation, but shares this data with other components, like a move component which influences the position.

  A simulation turn then consists of executing first the move component, which internally calculates the movement for all entities in parallel, before executing the colliding component, which then calculates the resulting collisions for all entities in parallel.

The ease of use of the different approaches is determined by the used programming language and libraries for achieving parallel execution.

## 4.2 Implementations in Java

As this thesis is based on practical work, done in the Java programming language, different prototypes for the approaches presented above were implemented:

1. *Sequential reference implementation*

   This implementation was the easiest one, from the programmer's point of view. All entities were simulated sequentially, afterwards the physical simulation was applied and the resulting changes sequentially applied to the entities. As an optimization a hierarchical tree was used for spatial sorting of the entities and obtaining collision information. During the update of each entity the attached behavior was executed, resulting in new events for the entity. Then those events were applied to simulation. Sensorial data, like visible enemy entities, was retrieved from the hierarchical tree, used for spatial sorting. The overall speed of this implementation depended on the used data model for spatial sorting. After conduction several experiments, the fastest data model was the so-called quad-tree. This hierarchical data structure has one node, which overlaps the whole map. This node is automatically subdivided into four child nodes, as soon as the amount of entities contained in this node reaches a fixed upper limit. This procedure is repeated until a hierarchy of nodes exists, each containing at most the fixed amount of entities. The procedure is reverted as soon as the subnodes of one node contain less than a lower limit of entities. The advantage is the fast access to nearby entities, but the disadvantage the costs of the insertion, moving, and removal of an entity, as these can trigger many changes to the tree structure. However, the conducted experiments have shown that the fast access to nearby entities compensates for the disadvantages.

2. *Parallel execution of all entities, followed by the physical simulation*

   Based on the first implementation, the difference was the encapsulation of each entity as a single Java runnable. The distribution to the available threads was done through Java's built-in thread pools. Many changes to the collision detection implementation were necessary to support parallel access. The hierarchical tree could not be satisfyingly modified to support parallel access. Therefore, it was replaced by a regular grid, dividing the map into cells which allowed individual locking for write access. The sensorial functions were successfully modified for this new spatial sorting structure. Different experiments with the diverse implementations of Java's thread pools showed that a thread pool with a fixed number of available threads had the best results in terms of time needed to complete a test. Other alternatives to this fixed thread pool are the single thread pool and the cached thread pool, which generates new threads on demand, but reuses previously created threads.

3. *Parallel execution of grouped entities, followed by the physical simulation*

   This implementation is mostly based on the previous one, but instead of submitting each single entity to the thread pool, this implementation grouped entities by their location on the map. This grouping allowed some simplifications in the collision detection implementation, while introducing new complexities in the group management of the entities. Different strategies for grouping were tested, the best results were achieved while dividing the map into the number of available threads stripes along the x-axis and dividing these stripes into the number of groups per thread rectangles along the y-axis. The basic idea for this approach was a result from the previous locking of the cells. The main problems were to resolve collisions across cell borders and moving of entities between cells. Other tried implementations used mainly different strategies for forming the entity groups. These included dynamic grouping, based on the entity density on the map, and different static implementations, like dividing the map into a double number of stripes as threads are available. In this implementation first all odd numbered stripes were executed by the threads, then the even numbered stripes, which should remove the necessity to synchronize the different stripes. However, the experiments showed that the distribution of threads to the thread pool has a larger impact on the runtime than the synchronization between the threads.

4. *Parallel execution of grouped entities with concurrent physical simulation*

   The implementation tried to further simplify the collision detection. Therefore collision detection was done in the local group. The intended simplification was not possible, due to the further increased complexity for the handling of the borders of a group and the resulting interactions with other groups. Additionally, the above-mentioned side effects of non-deterministic execution order occurred more frequently than in the other implementations.

## 4.3 Experiments

All different approaches were examined for their ease of implementation, overall performance, and scalability. To measure the ease of implementation the overall time to complete the data model and the complexity and amount of interactions between the different model elements was used. The used settings for the comparisons consist of a flat map. The map is filled with tank agents, each with the same behavior. All tanks are placed with equal distance to each other, forming a grid on the map. The behavior consisted of only two sub-behaviors:

- If another tank is in my field of vision, shoot at this tank.

- If no other tank is in my field of vision, move to the origin of the map and turn the field of vision clockwise.

The different implementations were tested with an increasing initial count of tanks and increased number of used threads. The main focus of the examination lay on the maximum usable count of threads. Therefore the test was run with different numbers of tanks and threads. For each number of threads the maximum count of tanks was determined at which the implementation was able to achieve at least 10 updates of all entities per second. As a maximum the number of used threads were declared, beyond which additional threads did not further increase or even decrease the number of tanks the simulation was able to handle with the same speed. All tests were performed on different systems, including different Java versions and vendors. For comparison, the numbers below are from measurements on a quad CPU with each dual core AMD Opteron system, running Linux and a Sun Microsystems reference Java implementation.

The obtained results were:

1. *Sequential reference implementation*

   As intended, this implementation was limited by the available power of the used CPU core.

2. *Parallel execution of all entities, followed by the physical simulation*

   This approach limits the maximum useable amount of threads through the sequential overhead of distributing the agents to the available threads. This distribution uses a lot the synchronization features of the Java language. The costs of this synchronization, in terms of used time, is considerably high in comparison to the actual executed behavior and necessary calculations for a single entity. The maximum number of usable threads were, with little difference over one thread, two threads.

3. *Parallel execution of grouped entities, followed by the physical simulation*

   The use of groups of entities reduced the overhead of the distribution. However, the synchronous access to the physical simulation, for updating the position of an entity, emerged as a new limit for the usable number of threads. The maximum number of usable threads was three. This might be a result of the applied spatial sorting, however, profiling results showed that most time was still used for synchronizing between the different threads.

4. *Parallel execution of grouped entities with concurrent physical simulation*

   The further reduction of needed synchronization resulted in a maximum usable number of up to four threads for parallel processing. But the resulting simulation results are not fully comparable to the other implementations due to the side effects of the concurrent physical simulation.

Due to the complexity of the implementation, no prototype using a fully component-based approach was implemented. The obtained results indicates that any form of using numerous, at least partially synchronized threads is not the best way for building a parallely executed simulation in Java.

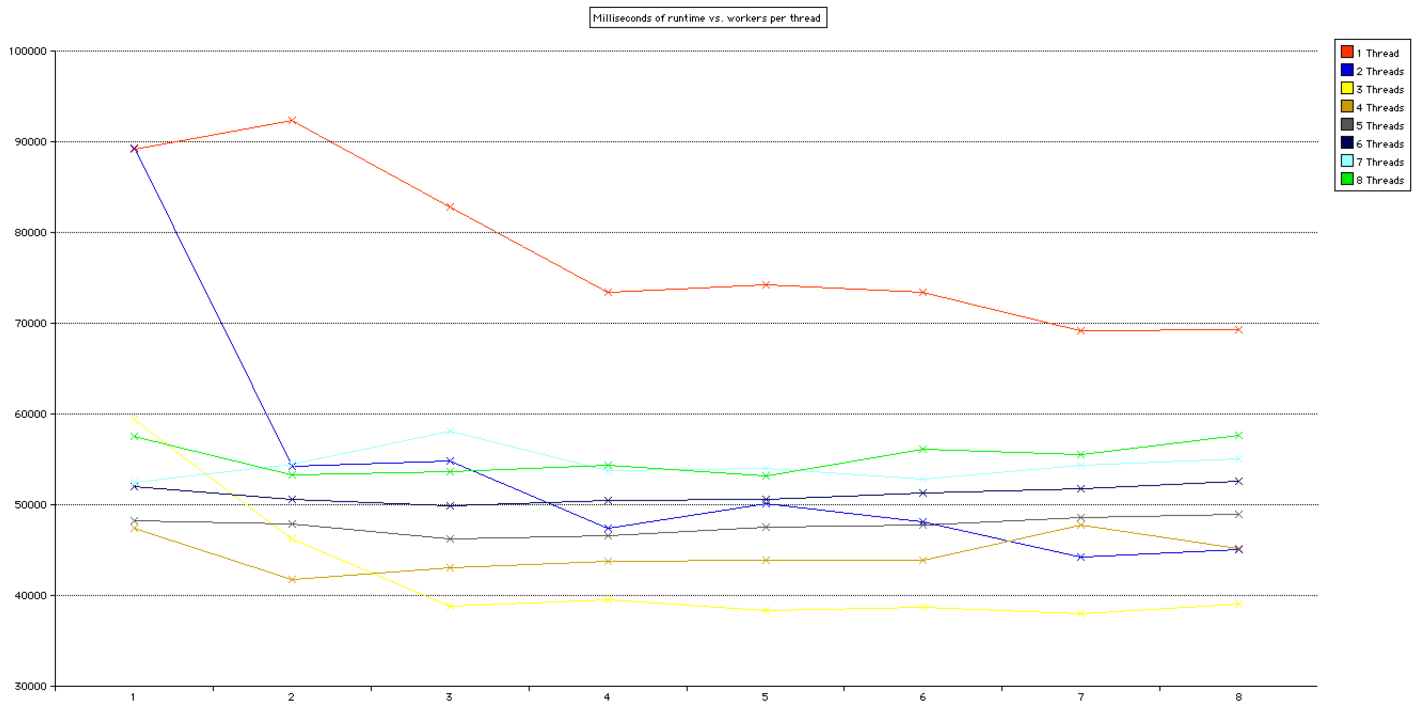

Figure 4.1: Speed measurements with different numbers of threads and workers for the third prototype.

As an example for this conclusion a result of a test run is shown in figure 4.1. This result is from the third implementation and the setup contains 110,889 tanks, forming a grid of 333 times 333 tanks. Each line represents a number of used threads. The measured values are the amount of used entity groups per thread versus the needed time in milliseconds until one or zero tanks survived which approximately occurred after 10,000,000 frames. Thus a lower time is a better result. More than eight threads were not tested, because of the expected overhead of the needed context switches due to the availability of eight cores. The figure refers to worker groups which are groups of entity groups. As the map is divided into cells, each cells can contain a group of entities. The cells are of fixed size. To achieve an overall fair distribution of these groups to the available threads, the distribution is done randomly every single run of the simulation during the creation. Therefore, all setups have been automatically tested several hundred times. One worker group is assigned to one thread. The results of one to four threads are of particular interest.

The red line, showing the results of using one thread, shows a slightly higher execution time while using one group than the sequential implementation. The even worse result, while using two groups, can be explained with the additionally necessary maintenance overhead for regrouping of the moving tanks. With the use of three threads the overhead of regrouping contributes even more to the execution time, but the resulting time is lower due to the simpler collision detection over the much smaller groups.

The blue line, showing the results of using two threads, illustrates the influence on the execution time of regrouping the entities with one group per thread and the overhead introduced due to the needed time for synchronization for each regrouping. This overhead is drastically reduced while using two or more groups per thread. With two groups per thread, the probability that an entity changes into the other group of the same thread, thus needing no synchronization, is one third.

The yellow line, showing the results of using three threads, is the optimal setting for this implementation, if at least three groups per thread are used. With one group per thread, the results are much better than using one or two threads as the needed regrouping, and therefore synchronization, is considerably lower. With three groups per thread, nine groups exist overall. If an entity needs to be regrouped, one of eight groups are possible with two 2 of 8 in the same thread. This results in overall similar times for thread synchronization and collision detection.

The brown line shows that with four threads the overhead of the synchronization between the threads increases if more than two groups per thread are used. The contribution of the synchronization to the total execution time is constant, even if more groups per thread or more threads are used.

The results can be slightly shifted towards more threads, if more tanks are used, but at the same time, the influence of the Java memory handling on the total execution time increases, too.

## 4.4 Distributed computing approach

As this thesis aims to show an approach, useable in a massive multiplayer online RTS game, the use of a central server system for the simulation of a game world is necessary. The aim to support a massive amount of players leads to the need to simulate a huge number of entities in realtime. The presented approaches in 4.2 are not able to scale with an increasing amount of simulated entities due to the limits of the synchronization and memory handling within Java.

To avoid the drawback of thread synchronization and reduce the overhead of the memory handling of Java, an approach using distributed computing can be used. Within this approach, most of the time intensive calculations are done in another application besides the simulation server. The simulation needs only the position and bounding volume of each entity. Another application, the so-called proxy, manages the entities. When an entity is created, the proxy notifies the simulation and the other proxies of the new entity. As an owner of the entity, the proxy calculates in a separate simulation loop the commands for each self-created entity and sends relevant updates to the simulation server, when a change has occurred. The changes the server receives can be limited to:

- A create notification for a new entity.
- A move vector for an entity, which is applied in each update to the entity's position.
- A remove notification for a deleted entity.

At each simulation turn, the move vectors are applied to the entities and collision notifications are sent to the proxies. Additionally, the create and remove notifications are forwarded to all proxies.

This may sound like a much slower approach than the above presented ones, but the implemented prototypes did not include the necessary communication between simulation and participating clients for the players. As this communication is crucial for a real game, the overhead of communicating the changes of the simulation to the clients has to be considered. In this approach, the proxies are able to calculate the necessary changes for the client. There is no need for a communication between the clients and the simulation server in this approach. The load of calculating the behavior for each individual entity can therefore be distributed from the simulation to the proxies. At the same time, the structures of the simulation can be more optimized, as every entity has only a position, bounding volume, and in the case of movement a movement vector. Additionally, the restriction of using only one simulation thread allows the application of highly optimized collision detection algorithms without consideration of synchronous access problems.

One problem in this approach, which might need more considerations, is the synchronization between the clients. It might happen, for example due to a slower reacting proxy application, that one client falls behind other clients, therefore seeing only the past of the simulation and giving commands based on past information. However, an implementation of this approach was not possible within this thesis. Therefore, no real experiences regarding this or any other problem are available.

# 5 Conclusion

This thesis has presented two aspects of realtime strategy games. Chapter 3 has shown the currently applied technologies of script-based and hierarchical finite state machine based decision making systems, the so-called artificial intelligence for realtime strategy games. This chapter furthermore defined the two rules of continuous and adaptive gameplay and introduced behavior trees as another technology for decision making. All three technologies were compared to each other and the ease of use and extensibility of behavior trees were explained. Additionally some extensions, mainly the stimulus for faster reaction to changes and the blackboard for group coordination, were introduced. The chapter concluded with an overview of how the technology of behavior trees fits into the genre of realtime strategy games.

Chapter 4 started with a summary of different possible parallelization approaches for a realtime strategy game's simulation. Some of the outlined approaches were implemented as part of this thesis and the employed technologies and ideas were explained. These implementations eventually led to the conclusion that implementing a parallel simulation in Java is not the preferable approach. This conclusion is drawn from the obtained results from the different implementations. As an alternative, another approach, using a distributed computing approach, was presented and the possible improvements were discussed.

The conducted experiments with the integration of behavior trees were, in opposition to the parallel simulation approach, a full success. The implementation for the experimental prototypes, as described in chapter 4.2, was considerably easier to use and implement, than an alternatively implemented HFSM engine, while providing the same set of functional features. Based on the gained knowledge and good results while using behavior trees, the integration within the current research project jMMORTS is planned in the near future. In addition, the planning for the development of a tool for behavior design has also been started. As a vision for the jMMORTS project, computer-aided gameplay is currently under investigation. With an extended utilization of behavior trees it might be possible to give normal users of the game an easily useable tool for creating their own behaviors for their entities. The goal is to provide a system, freely configurable by the player, which is able to continue a certain level of player-specific gameplay, even when the player is not currently using the game and therefore making the game more realistic, as the entities of a player do not vanish, when the player leaves the game.

# Bibliography

[AiGameDev] Understanding Bahavior Trees - AiGameDev.com
`http://aigamedev.com/`

[AiGameDev] Game Design 2.0: Learning from Social Networks by AiGameDev.com
`http://aigamedev.com/design/social-networks`

[AiGameDev] Using a Static Blackboard to Store World Knowledge by Alex J. Champandard, July 18th, 2007
`http://aigamedev.com/architecture/static-blackboard`

[Blizzard] Blizzard company profile by Blizzard Entertainment
`http://www.blizzard.com/inblizz/profile.shtml`

[Bobic00] Advanced Collision Detection Techniques by Nick Bobic, March 30, 2000
`http://gamedevelopment.com/features/20000330/bobic_01.htm`

[Brun06] Brun, J, Safaei, F & Boustead, P, Fairness and playability in online multiplayer games, 3rd IEEE Consumer Communications and Networking Conference (CCNC 2006), 8-10 January 2006, 2, 1199-1203.
`http://ro.uow.edu.au/cgi/viewcontent.cgi?article=1230&context=infopapers`

[Chen08] Bibliography of Network Games Research by Chun-Yang Chen, May 22, 2008
`http://www.iis.sinica.edu.tw/~cychen/ngbib.html`

[C&C] Command and Conquer: cheat codes, hints, and help
`http://www.gamewinners.com/Cheats/index.php/Command_And_Conquer_(PC)`

[Dune2k] Dune Fansite by Jesse Reid
`http://dune2k.com/Duniverse/Games/DuneII`

[Ensemble Studios] Ensemble Studios - Age of Empires
`http://www.ensemblestudios.com/Games/AgeOfEmpires/Default.aspx`

[Gamasutra] Gamasutra . The Art & Business of Making Games
`http://www.gamasutra.com/`

[GameSpot] Gamespot presents: A history of Real-Time-Strategy Games by Bruce Geryk
`http://www.gamespot.com/gamespot/features/all/real_time/`

[GameSpot] About GameSpot's Reviews and Ratings
`http://www.gamespot.com/misc/reviewguidelines-old.html`

[GDC02-halo] The Illussion of Intelliegence by Chris Butcher and Jaime Griesemer
http://halo.bungie.org/misc/gdc.2002.haloai/talk.html

[GDC07] Three Approaches to Halo-style Behavior Tree AI
https://www.cmpevents.com/GD07/a.asp?option=C&V=11&SessID=4704

[Guy99] A Modular Framework for Artificial Intelligence Based on Stimulus Response Directives by Charles Guy
http://www.gamasutra.com/features/19991110/guy_01.htm

[IGN06] The State of the RTS by Dan Adams April 2006
http://uk.pc.ign.com/articles/700/700747p1.html

[Isla] Damián Isla
http://www.bungie.net/Inside/MeetTheTeam.aspx?Person=isla

[Isla05] Damian Isla: GDC 2005 Proceedings: Handling Complexity in the Halo 2 AI
http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml

[Java SE 1.5] JavaTM Platform, Standard Edition 5 Overview
http://java.sun.com/javase/5/docs/technotes/guides/index.html#jre-jdk

[Java Tutorials] Executors (The Java Tutorials > Essential Classes > Concurrency)
http://java.sun.com/docs/books/tutorial/essential/concurrency/exinter.html

[jMMORTS] Research Group Programming Languages / Methodologies: Projekt Spieleentwicklung MMORTS
http://www.plm.eecs.uni-kassel.de/plm/index.php?id=711

[Lent99] Developing an Artificial Intelligence Engine by Michael van Lent and John Laird
http://ai.eecs.umich.edu/people/laird/papers/GDC99.pdf

[Lua] Lua, the programming language
http://www.lua.org/

[LWJGL] lwjgl.org - Home of the Lightweight Java Game Library
http://lwjgl.org/

[Netbeans IDE] Welcome to Netbeans
http://www.netbeans.org/

[Orkin05] Agent Architecture Considerations for Real-Time Planning in Games by Jeff Orkin, 2005
http://web.media.mit.edu/~jorkin/aiide05OrkinJ.pdf

[Russel02] Artificial Intelligence: A Modern Approach (2nd Edition) by Stuart Russell and Peter Norvig, 2002
http://aima.cs.berkeley.edu/

[Sierra] Sierra Entertainment - Empire Earth
`http://www.sierra.com/en/home/games/game_info.prod-L2NvbnRlbnQvc2llcnJhL2VuL3Byb2R1Y3RzL2VtcGlyZV9lYXJ0aF9nb2xk.html`

[Sun microsystems] Sun microsystems
`http://www.sun.com/`

[Troy2005] RTS Design by Troy Dunniway 2005
`http://www.dunniwaydesign.com/rts_design.htm`

# List of Figures

# A Acknowledgment

*Björn Knafla*

Many thanks for the last two years! I'm looking forward to keep in touch with you and to continue the countless discussions about our thoughts not only about game programming. Many thanks too, for the huge amounts of written and verbal annotations and the guidance into a better direction of this work.

*Simone Krug*

Countless hours of talking, motivating, and much more hours of correcting made this work readable and understandable. Without you, this would not been finished yet! Many thanks and love to you my dear.

*Michael Zaton*

Far away but helpful, too. Many thanks to the language authority overseas! Hopefully we meet again soon.

*Monika Krug*

Many thanks to you, too. Blazing fast feedback and large emails of error corrections and annotations helped a lot to raise the language-level of this work.