PG511 - CI in Games - Final Report

Holger Danielsiek, Christian Eichhorn, Tobias Hein, Edina Kurtić, Georg Neugebauer, Nico Piatkowski, Jan Quadflieg, Sebastian Schnelker, Raphael Stüer, Andreas Thom, Simon Wessing

No. CI-252/08

Technical Report ISSN 1433-3325 August 2008

This work is a product of the Collaborative Research Center 531, "Computational Intelligence," at the Technische Universität Dortmund and was printed with financial support of the Deutsche Forschungsgemeinschaft.

TECHNISCHE UNIVERSITÄT DORTMUND
FACULTY OF COMPUTER SCIENCE
Project Group 511: CI In Games

CI in games

Final Report

Holger Danielsiek, Christian Eichhorn,
Tobias Hein, Edina Kurtić,
Georg Neugebauer, Nico Piatkowski,
Jan Quadflieg, Sebastian Schnelker,
Raphael Stüer, Andreas Thom,
Simon Wessing

August 13, 2008

INTERNAL REPORTS

Tutors:
Nicola Beume
Boris Naujoks
Mike Preuß

Contents

1 Introduction

"Computer Science is no more about computers than astronomy is about telescopes." *E. W. Dijkstra, Dutch computer scientist(1930 - 2002)*

1.1 Overview

We would like to begin by introducing ourselves and our previous and current work. In the curriculum of computer science at the Technische Universität (TU) Dortmund each student has to attend a project group for a period of two semesters. During this period students have an opportunity to improve their teamwork abilities in order to acquire skills to cope with demands they might meet at their workplaces after the graduation. So we formed a group supervised by Nicola Beume, Boris Naujoks and Mike Preuß. Because of consecutive numbering at the faculty of computer science we received the working title "project group 511" (PG 511). Our research topic was "CI in games - Methoden der Computational Intelligence zur Entwicklung von Spielstrategien" (developing game strategies using methods of Computational Intelligence).

Due to the fact that the Artificial Intelligence (AI) of non-player characters (NPCs) in computer games often features shortfalls, the PG 511 tried to improve NPCs' intelligence by means of Computational Intelligence (CI), such as Artificial Neural Nets (ANN) and Evolutionary Algorithms (EA). Our hope is that this approach would make their behaviour less predictable. The main task of the PG 511 was to figure out whether CI methods could be applied to computer games, how far CI methods can balance the game's difficulty while giving the observer an impression that the NPCs behave in a natural, human-like way.

We approached two different projects: During the first semester we worked on a clone of the arcade game *Pac-Man* by Namco [9], in which we improved the behaviour and the intelligence of the ghosts using CI methods. Furthermore we tried to determine whether a player perceives a game against enhanced ghosts as more interesting than one against the standard ghosts. Since *fun* is a fuzzy term, we agreed on the concept of *flow* from Mihály Csikszentmihályi's "Flow Theory" [Csi90] as the working definition which covers our sense of fun in games. Yannakakis and Hallam [YH04] published a measure of fun for Pac-Man clones so the PG 511 tried to check this theory by conducting a survey at the Campusfest, the open day at our university (see section 9.2).

The second semester of the PG dealt with a more complex game named *Glest* [4]. Glest is a real time strategy game. In this genre there are more options to integrate CI methods with the aim of improving the intelligence of the NPC. These options can be divided into two general parts: On the one hand there is a *global* intelligence which includes the main strategy of the NPC such as deciding on when to attack the enemy or which type of resources to gather. On the other hand there is the *local* intelligence includes the decisions of the single unit such as how to choose the correct path to the target or which target to attack. The techniques belong to the local intelligence could also improve the usability and convenience of the player by reducing the effort of micromanagement. So we divided into two subgroups which dealt with the two main topics: The *global and local AI.*

1.2 Chronological development and structure of this report

As mentioned above we had two semesters for developing CI methods for computer games. In the first semester we started with a seminar phase in which every member of our PG had to give a presentation on a special topic related to our later work. This seminar phase took place in April 2007. The major topics are described in chapter 2. After that we discussed which Pac-Man clone to redesign and we started with modifications of NJam [3] at the end of April. Our work was completed in the first week of June and we presented our progress at the Campusfest, the open day at the Universität Dortmund. We used this opportunity to collect feedback on whether our enhancements increase playing fun, which we then analyzed and the results were included in the intermediate report. We finished this report in the middle of July, right on time for the semester break. The results are described in chapter 3.

The second semester started in October 2007 with another seminar and we first discussed which game we would work on now. After we decided to modify Glest we recoded the code for about three months. Our achievements on this project are described in the chapters 4, 5, 6 and 7. We close this report with a short summary of our work on Glest in chapter 8, and with one last glance on the past year and the experiences we made during this period in chapter 9.

2 Background

"Nature does nothing uselessly." *Aristotle, Greek philosopher (384 BC - 322 BC)*

2.1 Evolutionary Algorithms

The algorithmic principle behind Evolutionary Algorithms (EA) is inspired by the theory of evolution in nature as established by Charles Darwin [Dar59] and his idea of the "survival of the fittest". EAs are used for solving search problems and operate on a multiset of *individuals*, also called the *population.* Each individual represents one possible solution in the *search space* of the underlying problem. The search space is arbitrary in principal. Some possible search spaces are the boolean ($\mathbb{B}^n$), or real ($\mathbb{R}^n$) number space, the space of permutations and trees.

The individuals and therefore the solutions are assessed a value by an evaluation function, which is called *fitness function* in the context of the evolutionary computing. A fitness function evaluates an individual with respect to the quality of the solution that it represents and ensures that the resulting values establish a partial ordering over all individuals.

The fitness value can effect the selection process. Usually the fitness function $f(s)$ maps each individual s from the multiset S to a specific value in $\mathbb{R}$ as shown in equation 1.

$$f(s):\ S\ \rightarrow \mathbb{R},\ s\ \in\ S \tag{1}$$

Since evaluating a fitness function needs the actual solution to be tried on the problem, getting the fitness value is usually a rather time-consuming process.

In order to find better solutions the new individuals are iteratively generated until a maximal predefined number of iterations or a certain fitness value is reached. Each iteration is called a *generation.* Generating new individuals is called *reproduction* and it follows the idea of reproduction in nature. An offspring is generated from one or more parents by means of *mutations* and/or *crossover.*

At two points during its progress, an EA needs to make a *selection* of individuals. Firstly, a subset of individuals (*parents*) needs to be selected for reproduction in order to generate new individuals. Secondly, it selects a set of individuals (*survivors*) from the sets of new ones (*children*) and possibly of old ones as well, which will be the starting population of the next generation. This selection depends on the fitness value. A general description of an EA is presented in listing 1, while the remainder of this section explains each step in more detail. In the following we describe only some basic operators. Remark that there are many more.

Initialization This first step generates the initial population. If few or no information about the population is available the commonly used method is to choose randomly from uniformly distributed candidates. Although any candidate solution can be chosen as starting point, it may prove better to initialize the population with promising points (in case these are known), since the algorithm can converge faster towards a local optimum[1].

[1]This, however, may also go completely wrong resulting in EA getting stuck in a local optimum, even if a better solution could be found, so using promising points in the search space is a decision which has

```
t:=0
initialization of the start population P(0)
evaluation of the population P(0)
WHILE terminating condition not satisfied
  select the parents
  generate offspring from the parents
          by the means of recombination and mutation
  evaluate offspring
  select the following population P(t+1)
  t:=t+1
END WHILE
```

Listing 1: Pseudocode of an Evolutionary Algorithm

Parent selection Selecting individuals for reproduction can be done either uniformly at random over the entire population (called *uniform selection*) or it can be related to the fitness of the individuals. The latter method assigns a selection probability $p(s)$ calculated as shown in equation 2 to each individual $s \in S$, which means that fitter individuals have higher probability of being selected.

$$p(s) = \frac{f(s)}{\sum\limits_{x \in S} f(x)} \tag{2}$$

Parent selection is highly dependent on fitness values and behaves almost deterministically in case of great differences in fitness values.

Survivor selection In order to select individuals for the next generation the same operators as in the parent selection can be used. However, it has to be decided which set of individuals to choose from. In the *plus-selection* μ individuals are chosen from the union of old individuals and the offspring. This selection is denoted as $(\mu + \lambda)$, where μ is the size of the population and λ stands for the number of offsprings. Another type of selection is *comma-selection*, denoted as (μ, λ) , that allows to choose new individuals only from the set of offspring. In both cases the choice is deterministic since the individuals with highest fitness values are chosen.

Variation Depending on whether one or more parents are needed to generate an offspring, we speak of *mutation* or *crossover*. While crossover is not used by all EAs, mutation occurs always, possibly also after a crossover. Variating with crossover means to use parts of all parents to create an offspring. These parts can be chosen randomly or by a predefined formula. Figure 1 shows how individuals in $\mathbb{B}^n$ are combined using random crossover.

When an individual is mutated, some of its features are randomly changed by a predefined function. The typical mutation operators in $\mathbb{B}^n$ are the *standard bit mutation* and the *k-bit mutation*. By means of the former one, each bit is flipped with the probability p_m. This mutation parameter is kept small and usually set to $p_m = \frac{1}{n}$, with n denoting

to be taken seriously.

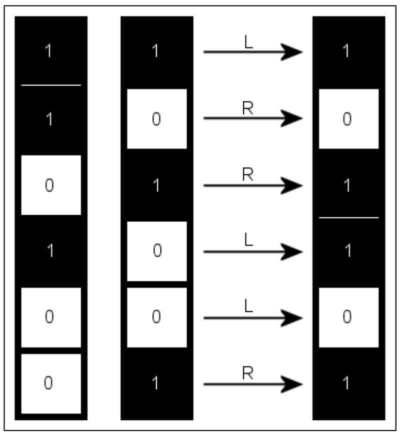


Figure 1: An example of variation by crossover. The two parent individuals (left) are in $\mathbb{B}^n$. For each bit, a value in L, R is chosen randomly, i.e. the information is taken either from the **l**eft or from **r**ight parent as the correspondent bit of the offspring (right).

the length of the bit vector. The 1-bit mutation flips exactly one randomly chosen bit. These operators can be adapted for use in other search spaces as well.

Parameters The behaviour of an EA is determined by the settings of the parameters (some of which have been described previously). A static parameter set allows no change on runtime and as consequence, the EA can progress slowly, since parameter values may vary while optimization. Thus, it can prove better to change parameter values depending on the number of generations, or generally optimization progress. This approach is called dynamic parameter control. Another possibility is to adjust parameters not only depending on time but on other features as well (i.e. fitness values or the distribution of the population over the search space), called adaptive parameter control. All these strategies are deterministic; this property is obvious for static parameter control, while dynamic parameter control operate according to predefined functions. The adaptation of parameters can also be included in an evolutionary process, which is then called self-adaptation.

2.1.1 Coevolution

A compact but sufficiently informative introduction to coevolution and all related issues that we address here can be found in [ES07]. The idea of coevolution is based on the fact that the survival of some organisms in the nature depends on other species or individuals of the same species. An easy understandable example of these dependencies is the plant pollination by insects or a predator-prey relationship. The former type of relationship is characterized by the cooperative nature of the mutual development. Insects are “helping” plants to reproduce whereas plants provide nourishment for many insects. The latter relationship is a competitive one, that is one species is gaining fitness at the other’s expense.

Drawbacks of coevolution There are some drawbacks in the coevolution paradigm. One of them is called the *Red Queen Effect*, named after a passage from Lewis Carrol's *Alice in Wonderland*[2] [Car78] (see also [DEH$^+$07]).

To make a progress in space, a Wonderlander has to move twice his highest possible speed, otherwise he can only manage to keep his current position. In the context of coevolution this means that by using a fitness value for an individual that depends on fitness values of other individuals we can never be sure of the type of development the population is going through: If a fitness value stagnates over a period of time, we cannot say whether the development of individuals is stagnating as well or increasing at equal speed.

Furthermore, there is the problem of circularity in the evolution. This effect appears due to the fact that the optimality of a strategy heavily depends on the strategy of the current opponent. With the change of the opponent, the optimal strategy might (and probably will) change. It can thus happen – and this effect was indeed observed – that, with three strategies A, B and C, we may have a dependency that strategy B beats strategy A ($A \prec B$), $B \prec C$ and $C \prec A$ (called "Rock, Paper, Scissors"-situation after the homonymous children's game). If such a situation occurs, a global optimum is not present due to the fact that there is no partial ordering of the individuals and the progress goes around in circles. These drawbacks can be weakened by means of *diversity maintenance*. The most common techniques are *crowding* and *fitness sharing*, described in the following.

Crowding is based upon the idea that the offspring are likely to be similar to their parents. To maintain diversity in a population, similar individuals should be replaced. The similarity is tested in tournaments between offspring and parents, in which the pairwise distances between parents and offspring are minimized. Resulting from the competition between similar offsprings and their parents, the subpopulations in fitness niches become equally distributed over the peaks of the fitness landscape.

Fitness Sharing is based on the idea that similar individuals have to share their fitness with each other. The shared fitness $f(s)'$ of an individual calculates as

$$f(s)' = \frac{f(s)}{m} \tag{3}$$

where $f(s)$ is the original fitness of the individual s and m is the size of the niche, in which the individual is located. It can be computed as a sum of sharing functions $\mathrm{sh}(d)$. Often a bell-shaped curve is used as $\mathrm{sh}(d)$ while d denotes the distance between two individuals. In the context of games such a sharing function is not essential, but it is possible to define two individuals as similar if they beat the same opponent. As Miles et al. [MQLL07]

[2]

"'Well, in *our* country,', said Alice, still panting a little, 'you'd generally get to somewhere else – if you run very fast for a long time as we've been doing.'
'A slow sort of country!', said the [Red] Queen. 'Now, *here*, you see, it takes all the running *you* can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"' [Car78, Page 147]

suggest, $f(s)'$ would then be calculated as

$$f(s)' = \frac{f(s)_1}{m_1} + \frac{f(s)_2}{m_2} + \cdots + \frac{f(s)_n}{m_n}. \tag{4}$$

Here $f(s)_j$ means the score of individual s against individual j and m_j means the sum of all scores against individual j. Though fitness sharing has a runtime of $\mathcal{O}(|P|^2)$, this should be insignificant compared to the time for evaluating the fitness function.

2.2 Artificial Neural Networks

The paradigm of Artificial Neural Networks (ANN) is inspired by biological nervous systems and their processing of information. Due to the complexity of their biological counterparts, these artificial networks are just an abstraction.

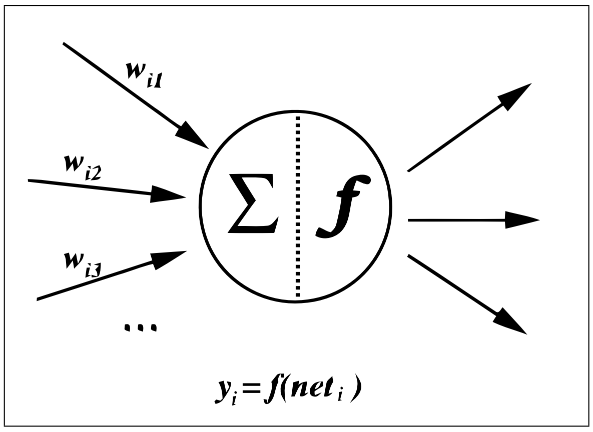


Figure 2: Illustration of an artificial neuron.

An ANN consists of a set of nodes called *neurons*(see figure 2), each representing a computing unit for itself. These neurons correspond to the nerve cells of the organic brain. The neurons are connected by directed edges bearing a weight which scales the signal flow along this edge. Although heavily reduced in their functionality compared to their biological prototype, a network of highly interconnected artificial neurons can solve complex problems. Therefore ANNs can be applied to a wide range of application like data classification, pattern recognition or function approximation.

Neurons A neuron is the basic component used to build an ANN. As stated before, its functionality is reduced. In short, it processes the incoming signals based on a *transfer function* γ and passes the results based on an activation function a. The transfer function usually sums up the weighted incoming signals (net). As for the activation function several alternatives exist:

step function

$$a(net) = \begin{cases} 1 & \text{if } \gamma(net) \geq \theta, \\ 0 & \text{if } \gamma(net) < \theta \end{cases} \tag{5}$$

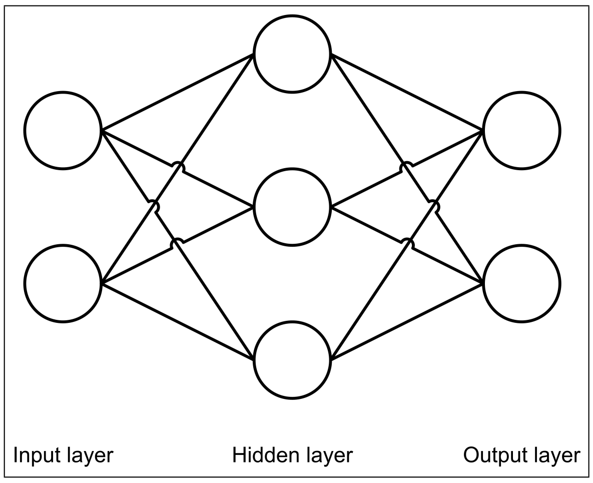


Figure 3: Illustration of a fully connected feed-forward Artificial Neural Network showing the three different layers.

linear function

$$a(net) = \begin{cases} 1 & \text{if } \gamma(net) \geq \frac{1}{2}, \\ \gamma(net) + \frac{1}{2} & \text{if } -\frac{1}{2} < \gamma(net) < \frac{1}{2}, \\ 0 & \text{if } \gamma(net) \leq \frac{1}{2} \end{cases} \tag{6}$$

θ describes a given threshold, triggering the activation of the neuron if reached, while $\gamma(net)$ describes the previously processed input.

Structure An ANN typically consists of several layers. The neurons without inbound edges are called the *input layer*, the neurons without outgoing edges are grouped together as *output layer*, the other neurons are said to be in the *hidden layer* which may consist of several sublayers. This structure is illustrated in figure 3. The signal flow path starts at the input layer following its way through the hidden layers to the output layer.

Input layer Neurons on the input layer receive the incoming signals from the ANN's environment and are already called *sensors.* Usually the inputs are, without any further modification, directed to the layer underneath (the sensors' activation function is the identity).

Hidden layer The neurons on the hidden layer(s) are used for computations based on the signals received by the input layer.

Output layer The output layer, like the input layer, works as an interface, connecting the ANN to its environment: Depending on the values calculated by the neurons on the hidden layer, neurons on the output layer are activated, passing the calculated result on to the external interface.

Several network topologies can be distinguished, two of the best known configurations will be described in the following:

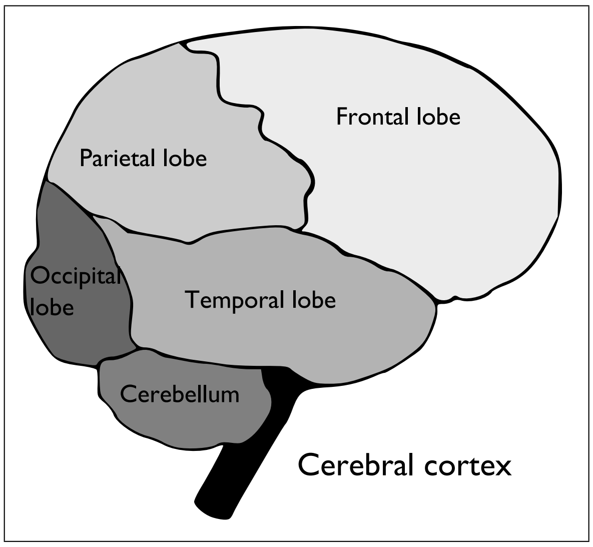


Figure 4: The human brain shown with the different areas of applications

- A net is called a *feedforward network* if each neuron is connected to each neuron on the next layer. Additionally, no connections between neurons on the same layer or connections to prior layers are allowed.
- *Recurrent networks* allow loops inside the network, i.e. connections targeting a neuron on a previous layer.

Learning process There is a variety of learning methods for ANN, ranging from supervised and unsupervised learning to reinforcement learning [RN03]. The learning process is based on adjusting the weights of the edges between the nodes inside the network.

In case of *supervised learning*, a predefined set of training patterns is used, including a vector of input values and a vector of desired output values. During the learning process the ANN processes the given input values and adjusts its configuration with respect to its results compared to the desired outcome. This process may be repeated over several iterations, resulting in a satisfying configuration of the network which computes acceptable results for a specific application. It is to be expected that during the learning phase the configuration and therefore the knowledge of the ANN becomes generalized enough to make adequate decisions for yet unknown problems.

Related research ANNs in commercial video games are a still rarely used technique with only few applications already existing. Graham et al. [GMS04] describe an ANN based pathfinding algorithm suitable for dynamic environments. Kalyanpur and Simon [KS01] use a hybrid approach of EAs and ANNs to optimize mutation and crossover probabilities whereas Yannakakis and Hallam [YH04] use a similar hybrid concept to control the ghosts in Pacman.

2.3 Self-Organizing Maps

The Self-Organizing Map (SOM) was conceived by Kohonen [Koh01] in 1982. The main application of the SOM is in the visualization of complex data in a usually two-dimensional field to create an abstraction like in many clustering techniques. The paradigm was the human being, more precisely the brain and its brain maps. It is a fact that various areas of the brain are organized according to their use. Figure 4 shows the structure and organization of the brain. More recent experiments have revealed a fine structure within many areas, where for example visual response signals are obtained in the same topographical order in which they were received at the corresponding sensors. This ordering is also called a map. There are many advantages of such brain maps:

- In brain maps the relevant functions are close to each other; the connection lengths between the functions are thus minimized.
- Due to this topology the interference between functions is minimized.

Competitive learning In the competitive learning process different actors compete against each other for the same input information. In our case we have a network where neurons challenge each other. One of those neurons becomes the "winner" with full activity, who then suppresses the activity of all the other cells by negative feedback. After the completion of the learning process each neuron becomes sensitized to a different domain of the input information and acts as a decoder for it. The SOM uses the competitive learning to define an "elastic net" of points which are fitted to the input signal space to approximate its density function. For more details about competitive learning we suggest the reader to consult [Koh01].

SOM algorithm Normally the nodes are arranged in a lattice. The lattice type can be defined as rectangular, hexagonal or even irregular. A mapping from the input data space $\mathbb{R}^n$ onto a two-dimensional space (lattice of nodes) is defined. A reference vector $r_i = (\eta_{i1}, \eta_{i2}, \ldots, \eta_{in})^T \in \mathbb{R}^n$ is associated with every node i. The r_i are initialized randomly or, if information about the input data space is available, the reference vectors are set properly. After the creation of the SOM the r_i will attain two-dimensionally ordered values. This is a key effect of self-organization.

Now we start with an input vector $x = (\epsilon_1, \epsilon_2, \ldots, \epsilon_n)^T$, which is presented by the lattice of nodes, through parallel connections to all neurons. After comparing x with all the r_i the location of the best match is defined in some metric. If we consider $x \in \mathbb{R}^n$ as a stochastic data vector, the SOM can be seen as a "nonlinear projection" of the probability density function $p(x)$ of the high-dimensional input data vector x onto the two-dimensional display. We can use any metric for comparing vector x with all the r_i. Normally, the Euclidean distance $||x - r_i||$ is used. In Glest we have chosen a weighted Euclidean distance for calculating the best-matching node (see section 6.1). Another possible matching criterion is for example the dot product of x and r_i (see [Koh01] for details).

Another key component of the learning process in which the "nonlinear projection" is formed is the fact that nodes, which are up to a certain geometric distance topographically close to each other, will activate each other to learn something from the input x. In other words, the best-matching unit (BMU), in the following indicated by the subscript c, and its neighbourhood learn in a similar way regarding the input data x. The result is a local smoothing effect on the weight vectors r_i of the neurons in this neighbourhood, which finally leads to a global ordering. We can define the neighbourhood function as shown in equation 7 with integer t as a discrete time coordinate and a so called neighbourhood function, defining a smoothing kernel over the lattice of nodes, $n_{ci}(t)$.

$$r_i(t+1) = r_i(t) + n_{ci}(t)\,[x(t) - r_i(t)] \tag{7}$$

For convergence it is necessary that $n_{ci}(t) \rightarrow 0$ when $t \rightarrow \infty$. Two common forms of the neighbourhood defined by $n_{ci}(t)$ are rectangular and hexagonal. We used a rectangular form for our SOM (see section 6.1). Now let us have a look at the definition of $n_{ci}(t)$ as defined in equation 8 with the learning rate $\alpha(t)$ and the width of the neighbourhood kernel $\omega(t)$.

$$n_{ci}(t) = \alpha(t) * exp(-\frac{||p_c - p_i||^2}{2\omega^2(t)}) \tag{8}$$

p_c and p_i are the location vectors of the BMU c and node i in the array. The functions $\alpha(t)$ and $\omega(t)$ are monotonically decreasing functions of time. The equation assures that the BMU and the neighbourhood will be changed in a similar way. Also the magnitude of modifications corresponds to the iteration count. At the beginning the modifications of the SOM will be larger than at the end. This algorithm leads to a globally ordered two-dimensional map with each neuron sensitized to a different domain of the input information. For more information about the algorithm we refer to [Koh01].

Related research SOMs are very popular in the unsupervised learning category. Many fields of science are using SOMs for solving high-dimensional and nonlinear problems. For example Ritter et al. [RMS92] present how a SOM can be adapted for controlling a robot arm. Another application is the organization of large document files. Lin et al. [LSM91] show the use of a SOM in this area. SOMs are used in many areas, like statistics, signal processing, chemistry or medicine, and introducing it in the area of video games is a quite interesting and also new approach. Finally, we recommend the book by Kohonen [Koh01] which presents a good introduction to the topic.

2.4 Influence Maps

Influence Maps (IM) are an Artificial Intelligence (AI) technology used in the context of computer games. IMs are an areal based representation of the knowledge the AI has about the current game situation and are usually used for tactical decisions. They exploit the topographic information of the actual map to represent how the player influences different areas of the map. An IM may indicate where the units are situated or where the boundaries of the controlled areas lie. Via IM, a player may find out where enemy units are concentrated or get information about weak, exposed areas.

IMs may be used in any topology, but in the following we will talk about two-dimensional environments since these are found in most strategy games. Only a brief introduction is given in order to illuminate this technology used in section 6.3 and 6.4. A more detailed explanation can be found in our technical report [DEH$^+$07] in Appendix L. or in [Kir04].

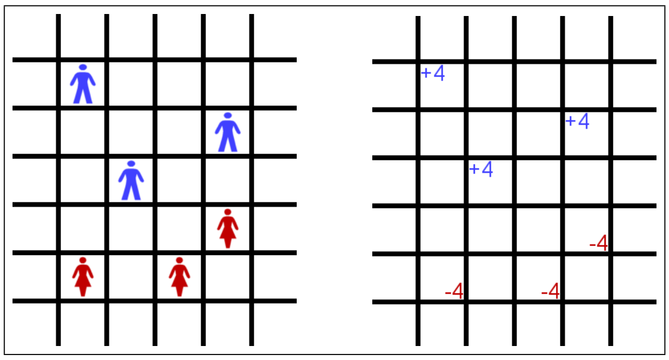


Figure 5: An example of influence maps generation. The left figure shows the units of the different faction (male, blue and female, red) placed on the map. The right figure shows the local influence of the units (+4 for the own and −4 for the other faction.)

Calculating Influence Maps Usually, a map is divided into tiles. The IM is set to the same size as the map with typically less tiles than the map. The tiles are initialized to a value of 0. To calculate the IM for one game situation, an influence value has to be calculated for every unit. This may be derived from any numeric property that unit has, for example its lifepoints or its attack strength. This influence is then propagated from the unit to a given area according to any formula imaginable, usually decreasing with increasing distance from the unit's original position.

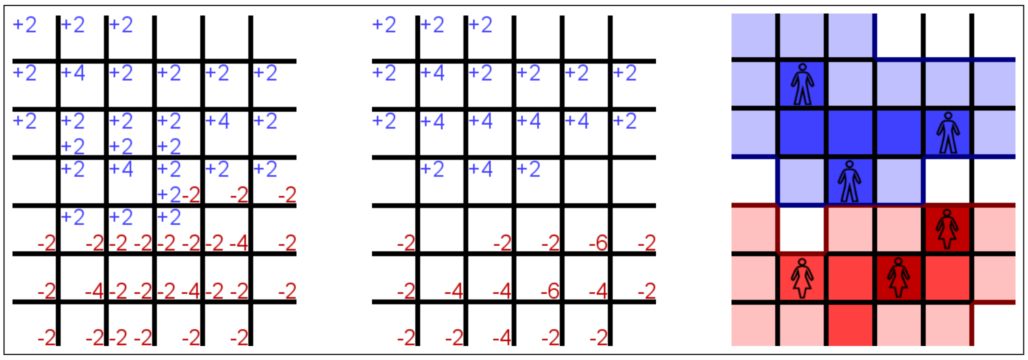


Figure 6: Continued example of Influence Map generation. The left figure shows the propagation of the influence values through the map, which are summed up in the center figure. The right figure shows the resulting influence map with the influence of the own (blue) and the hostile (red) faction plotted on the map. Darker colours show higher influence values. To clarify, the position of the original units are shown as well as the borders of the player's influence (outlined in the player's colour).

Example The following example shows how a simple IM is calculated. On a very small map (see figure 5) with a height of seven and a width of six tiles, there are two factions (blue and red). Each faction has three units on this map and each unit has an influence value of four. The influence radius of these units shall be one with a general decrease factor of $2^{-(radius+1)}$. The result of the propagation is shown in the left image of figure 6. The influence map is finally retrieved by summing up the influences of all factions. This is shown in the middle image in figure 6. It should be mentioned that the resulting influence may exceed the influence of a single unit if two friendly units stand close enough to each other. Also, the influences of two adversary units may cancel each other out. The right image of this figure shows a graphical representation of the calculated influence with the boundaries of the controlled area outlined for each player. The degree of influence is shown by shades in the player's colour.

2.5 Flocking

In computer games groups of units often behave very inflexible and each unit shows the same behaviour. A possible solution to this problem can be found in Craig Raynolds 1987 Siggraph paper "Flocks, herds and schools: A distributed behavioural model" [Rey87], in which he describes an approach to simulate an animated flock of birds. The motion of a flock of birds shows significant differences from the conventional motion of a group because every unit of the group shows a unique behaviour but the group reaches its global target. He named the individuals of his simulated flock "boids".

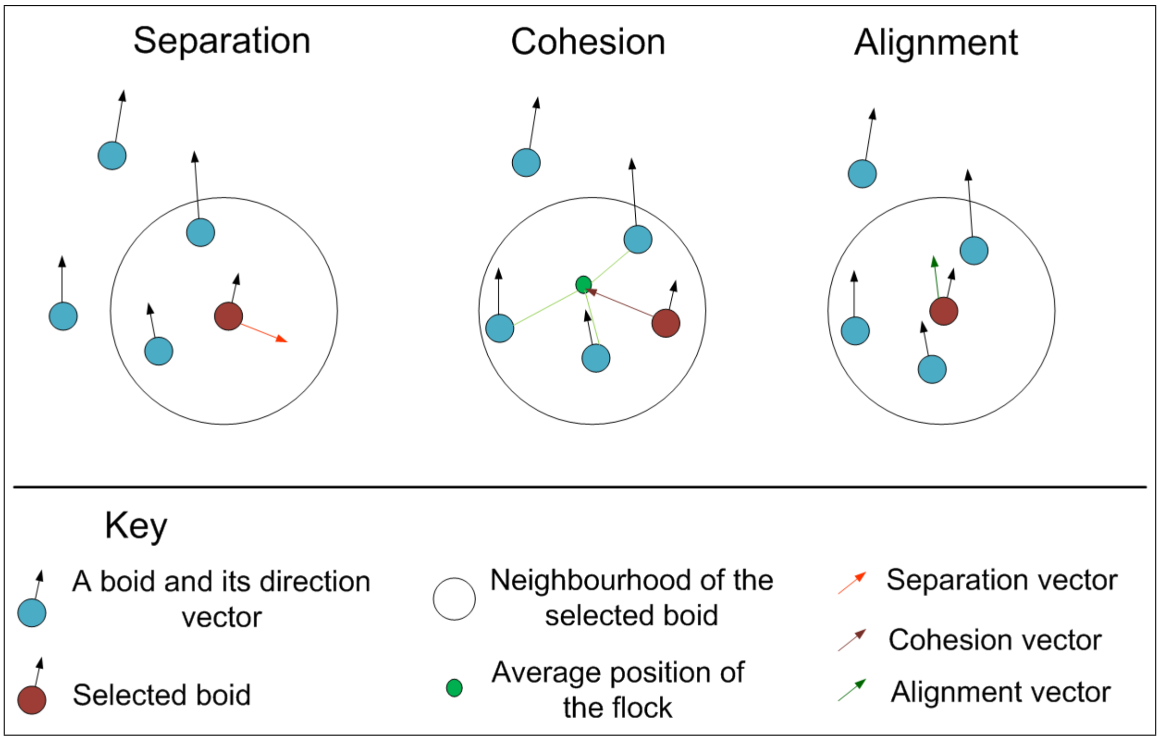

Separation
Cohesion
Alignment
Key
A boid and its direction vector
Selected boid
Neighbourhood of the selected boid
Average position of the flock
Separation vector
Cohesion vector
Alignment vector

Figure 7: The schematic diagram of the three flocking rules. The left diagram shows the resulting separation vector in this special situation. The middle diagram shows the cohesion influence for one boid and the right shows the alignment of a boid by considering the other boids in its neighbourhood.

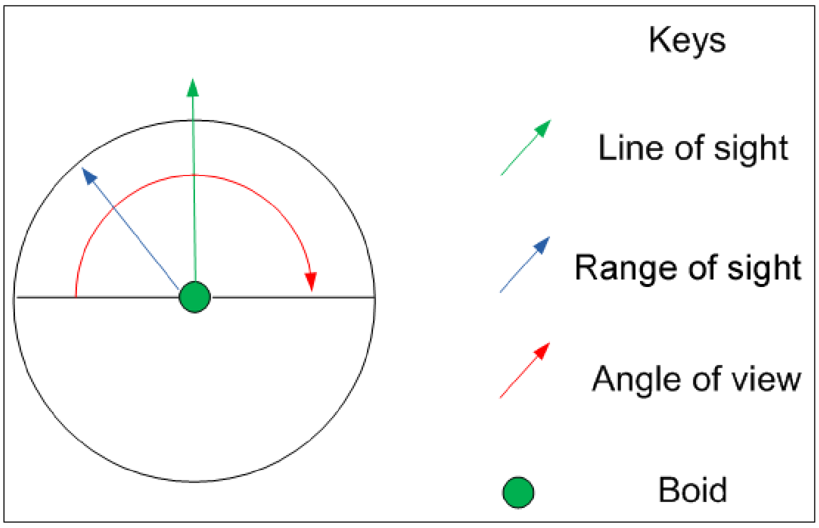


Figure 8: Angle of view and range of sight of a boid

These boids adjust their behaviour and their motion by following three simple rules which are shown in figure 7:

Alignment: Boids steer into the same direction as the other boids. Therefore, for each boid Raynolds calculates the average of the summed up direction vectors of the boids that are in the angle of view and in range of sight.

Cohesion: The ambition of each boid is to stick together with the others. The boids try to advance to the centroid of the group.

Separation: Boids try to scatter from other boids. So this rule can be seen as the opposite of the cohesion rule and represents the collision avoidance within the flock.

The movement direction of a boid is then calculated with the help of the three rules described above. To realize these three rules each boid has to know some additional information about its local surroundings especially it has to know where its neighbours are located, where they are heading and how close they are. So each boid has an angle of view and a range of sight in which the boid can recognize other boids and its environment (see figure 8). Obstacle avoidance is an important task of flocking and can be realized with this additional information so that each boid can avoid fixed obstacles. With the help of these rules and the information about the local surroundings of a boid the basic characteristics of flocking behaviour are described.

Flocking is a very useful method to simulate natural flock behaviour in computer games. Thus flocking is used to simulate flocks of birds or schools of fishes, military units or crowds. Shen and Zhou [SZ06] describe the use of flocking to steer the behaviour of military units in the futuristic ego-shooter unreal tournament.

Furthermore Davison [Dav05] mentioned that Half-Life [Val04], an ego-shooter published by Valve and Enemy Nations [10], and a strategy game created by Windward Studios uses flocking to control the movement of military units or to coordinate the formation of any units.

3 First Project: Pac-Man

"Computer games don't affect kids; I mean if Pac-Man affected us as kids, we'd all be running around in darkened rooms, munching magic pills and listening to repetitive music." *Martin Brigstocke, British comedian (1973 - . . .)*

In the first semester the project group (PG) concentrated their work on the simple, well known classic arcade game *Pac-Man* [9], developed by the Japanese company *Namco* in 1980. In Pac-Man, the player controls a figure called "Pac-Man" through a maze which is seen in bird's eye view. His task is to navigate Pac-Man so that he "eats" all of the dots (called food-pellets or pellets) lying in the tunnels of the maze, by which he wins the level and advances to the next. Pac-Man is chased by four computer controlled enemies ("ghosts") who eat (and therefore kill) him if they catch him, resulting in player losing one of his lives. If Pac-Man eats one of four special pellets called "power pills" he is able to invert the situation for a short period of time and eat the ghosts.

Since Pac-Man is proprietary, we did not use the original game but a reimplementation (clone) of the game available under GNU General Public License (GPL) [5] called *NJam* [3] by M. Babuskov in which we rebuilt Namco's original Pac-Man level as shown in figure 9. To use pathfinding algorithms, we transferred the maze used in this level into a graph. Each crossroad, or better, each tile which allows movement in more than two directions, is a vertex. The paths between the vertices are annotated with their length in tiles and a vector of directions absolute to the labyrinth (each direction being element in {"UP","DOWN","LEFT","RIGHT"}). The labyrinth with numbered crossroads is shown in figure 10, the resulting graph is shown in figure 11.

A word on spelling: we use the notation "Pac-Man" when referring to the Namco's original game or to the player-controlled figure. The notation "Pacman" refers to any clone or the player's figure in a clone.

Recent work concerning CI methods in Pacman games focuses on the control of the Pacman figure [GR03, YH04], whereas control methods for the ghosts are still rather unexplored [YH05]. In the first semester of our PG we continued the development of the ghosts.

The aim of good control algorithms for Pacman ghosts is not to optimize their ability to beat the player – which could be done easily by surrounding Pacman – but to behave so that the game is attractive for the player, provides fun and some challenge.

These emotions are of course subjective and hard to formalize. Currently, there is no general consensus about how to analyze the attraction of a computer game. Yannakakis and Hallam [YH04] as well as one of the PG's conference papers [BDE+08] propose a measure for the degree of human interest a Pacman game attracts, based on data of the game progress.

3.1 Algorithms to control Pacman Ghosts

The algorithms designed in the first period of the PG are described in the following three subsections. A more detailed description can be found in our CI report [DEH+07].

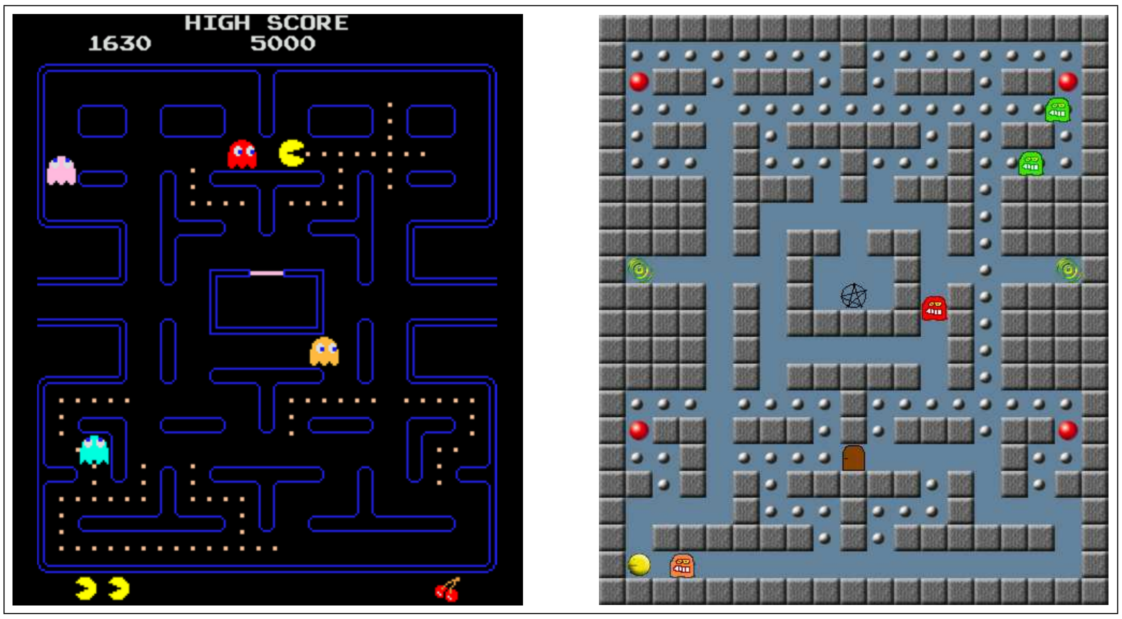


Figure 9: The original maze of Pac-Man (left) and the rebuild level in NJam (right).

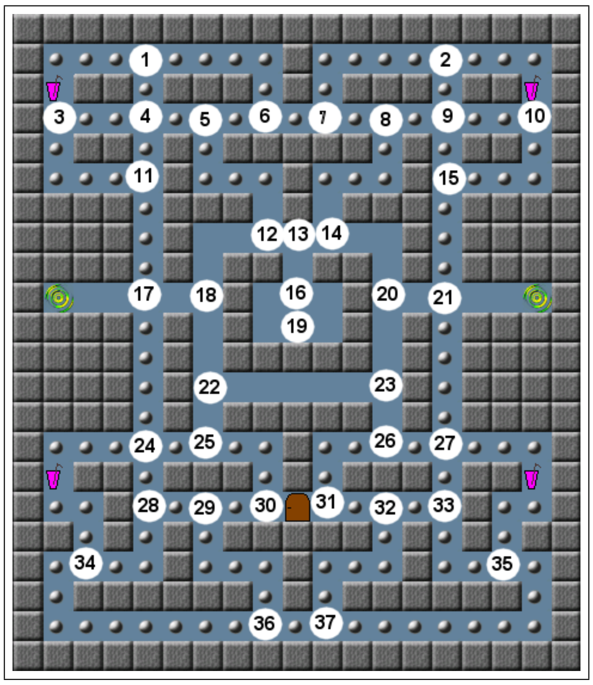


Figure 10: The rebuild maze of Pac-Man with numbered crossroads to be transfered into the gamegraph (see figure 11).

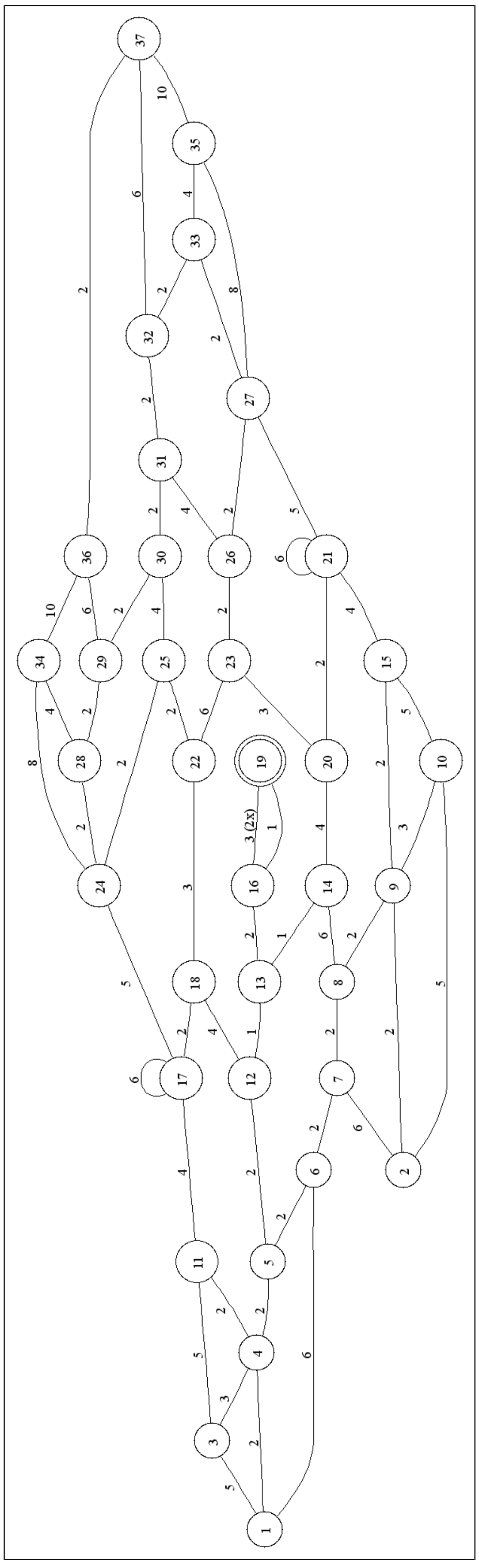


Figure 11: The graph devised from the Pacman maze; the numbering of the vertices corresponds to figure 10.

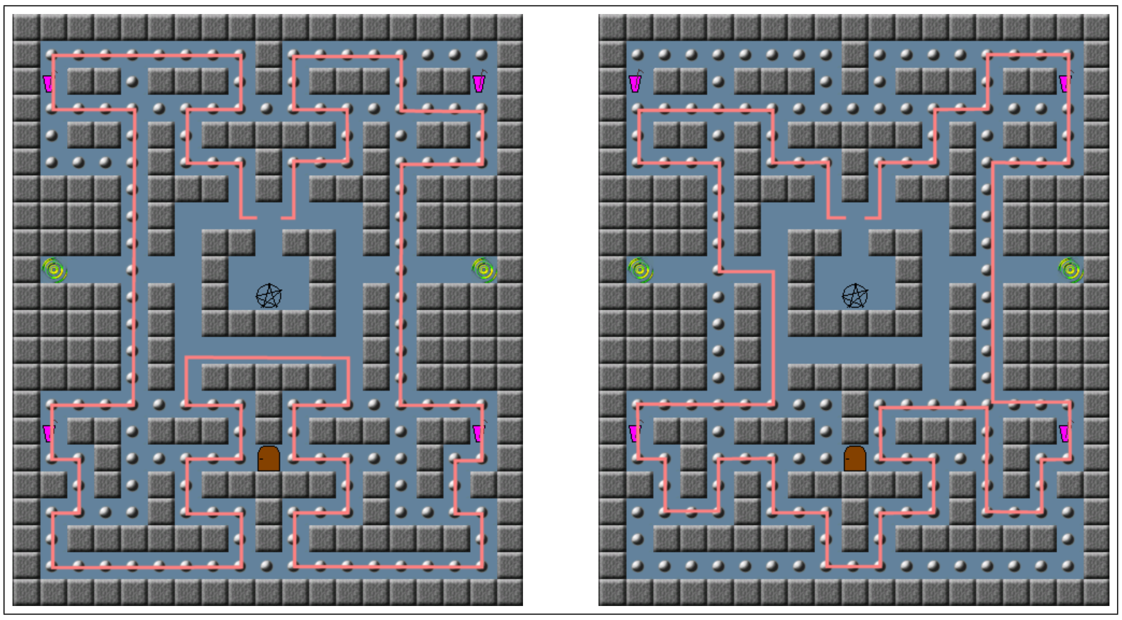


Figure 12: Routes which Firou may take through the labyrinth. The routes can be walked in either direction.

3.1.1 Ghosts controlled by Randomized-Deterministic Algorithms

Randomized and deterministic Algorithms were used as test to investigate wether AI trained by Artificial Neural Networks (ANN) or Evolutionary Algorithms (EA) are as difficult and fun-providing as manually devised algorithms. This “family” of ghosts is called Randomized-Deterministic (RD), because they are controlled by deterministic algorithms which use some randomness to make them less predictable. Apart from this, we also thought it to be natural to use randomness in the player's foe since it occurs in many board- or role-playing games like Ludo [8] or Dungeons and Dragons^{TM} [2] where a dice is used. The different algorithms we designed are described in the following.

Firou is a very simple RD ghost. He has two routes programmed which may be walked in either direction. At their starting point he chooses uniformly distributed one of the four possible routes and walks along the chosen route until he reaches the starting point where another route is chosen. The routes are shown in figure 12. He is absolutely ignorant of Pacman and can catch him only if Pacman walks directly into him.

Randegho walks randomly through the maze. Directions blocked by walls receive a probability of zero. Otherwise the probability for moving to a specific direction is only affected by the direction he moved last. This prevents the ghost from turning around every few steps. *Randegho*, same as *Firou*, receives no information about Pacman's position and does not react to him.

Old Shimer is conceived to be an old and therefore experienced hunter in the labyrinth. Due to his high skill, he has achieved a very good intuition about his surroundings and has therefore absolute information on the vertices that he has visited and their incident edges. Since he is very old, his memory is poor and he can only memorize ten vertices. When he finds the eleventh vertex, he forgets the first seen one in his memory. His field of view is the area of his memory as well as straight lines in all four directions of maximal ten tiles (Old Shimer is a bit short-sighted) if not blocked by a wall. If he sees Pacman, he pursues him until he eats him or Pacman leaves his field of view.

Darmok is thought to be a sly Pacman hunter. He roams randomly through the maze, imitating *Randegho*'s behaviour. But he sets traps, which are invisible to Pacman and the human player. If triggered by Pacman, all *Darmoks* in the labyrinth are informed of the player's position and then walk towards the player's position using the $A^{\star}$-algorithm [DP85]. If Pacman walks through his field of view, he chases him until he eats him or until Pacman leaves his field of view.

Puck is devised to be a shy and maybe very young hunter in the Pacman maze. He roams through the labyrinth on few, programmed paths directing him in the four corners of the labyrinth in a randomly chosen order. If, during his walk, Pacman crosses his view, he chases him until he eats him or Pacman leaves his field of view. If the chase ends without Pacman being eaten, *Puck* returns to his starting point and rolls a new order of corners.

3.1.2 Ghosts controlled by Artificial Neural Networks

As one of the CI based approaches ANNs were used to improve the behaviour of the computer controlled ghosts in Pacman. The ghosts should learn how to handle certain situations and react in an appropriate manner, without the use of global knowledge. With respect to gameplay and fun this should lead to a more natural behaviour instead of a non-beatable AI which dominates the human player throughout the entire game.

Implementation One of the first goals set was the implementation of a generic framework for ANNs which could be used and adapted in upcoming projects. The decision to start the development from scratch instead of building up on existing libraries was made pretty early. The idea was to increase the understanding of the applied technology as well as to improve the coding skills in regard to the upcoming tasks during the next year. The framework consists of a set of classes which represent the hierarchy of the network:

PacNNNeuron a class describing a single node of the network.

PacNNEdge a class describing a link between two nodes of the network.

PacNN the class of the network itself connecting instances of nodes.

PacTools a collection of command line utilities.

With the primary design goal in mind the final framework works completely detached from the game. To make use of the framework an interface was developed to connect the game to the ANNs by passing certain information describing a current game to the ANN.

The description of a situation is based on the sensor data of each ghost. The sensor data includes basic information like the location of walls relative to a ghost's position or the direction of the human player's location again relative to the ghost's position (if there is a direct line of sight). The interface collects and encodes the given information into a proper format processable by the ANNs. As a result the ANN activates the neurons on its output layer which represents the next movement of the particular ghost.

Training The process of learning is an important part of the concept of ANNs as well as of the developed framework. To keep the development on a basic level the simple backpropagation algorithm was used. As a supervised learning method it is based on expertise and requires the awareness of the given problem. Given a set of training patterns, the backpropagation algorithm follows a simple scheme:

1. A pattern is selected, its input vector is fed through the network which calculates a result vector.
2. The result vector is compared to the pattern's output vector by an error function. If the error exceeds a predefined threshold the algorithm adjusts the network backwards.
3. The adjustment is done by modifying the connections between the nodes of the network. The method of steepest descent is used to adjust the weights so that the error is minimized. As stated earlier this process starts at the output layer and propagates the modification through the entire network until the input layer is reached, therefore the name backpropagation.

The data used for the learning process was a set of encoded descriptions of possible game situations and the desired reactions, precisely the direction to move. This information had to be created manually based on the developers' expertise.

3.1.3 Ghosts controlled by Evolutionary Algorithms

As seen in Chapter 2, there are many possible ways to control NPCs by using evolutionary algorithms. We selected three different approaches to gain a broad overview about the usability of EAs to generate game strategies. Two of the EAs use individuals which describe the full ingame behaviour of a ghost, the third EA employs playing field coordinates as individuals and evolves them online every time a ghost needs to be moved.

Implementation To minimize effort we developed a common framework which is used by all three EA approaches to interact with NJam. We created the singleton class EA-Manager to gain easy access to all running EAs. This allows combined use of different EA implementations in a running game. The manager handles instances of the EAController class which represents an actual EA. Every EAController administrates some EAIndividuals which are connected to the NPC class used by NJam to control the ghosts. By doing so, we could easily access the NJam objects to gather all needed information to evaluate the individuals without changing big parts of the NJam code.

Game Theory From a game theoretical point of view Pacman can be treated as a turn-based game between the Pacman player and the Ghost player. An individual may be considered as a strategy. We developed an EA of pure strategies, where the behaviour in the game is deterministically defined, and an EA whose individuals are mixed-strategies, meaning that their behaviour is randomly affected. We consider two kinds of strategies, because the pure ones are supposed to learn faster and be easier to analyze and for the mixed ones, we expect slower evolution progress but a more dynamical ingame behaviour.

First approach - Pure Strategies We designed a $(4 + 4)$-EA approach to generate game strategies by mutating randomly initialized strategies. A strategy is a mapping from situation-codes to legal moves. A *move* $d \in D$, with $D = \{up, down, left, right\}$ is called legal if there is no wall in direction d. Each *situation-code* is represented by an integer value, which can easily be calculated by

1. gathering all needed information as seen in table 1,
2. generating a bit vector $v \in \mathbb{B}^{13}$,
3. converting v to its decimal value $k \in \mathbb{N}$.

The selection, variation and fitness calculation is done every 180 game loops, so the ghost behaviour will change during game play. The variation is done by remapping $^1/_5$ of all possible situation-codes to a new legal move. The values where derived from empirical experiments. One should notice that there are many unused codes, this was improved while working on the papers (see section 3.3).

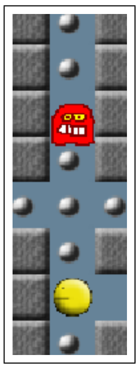


Figure 13: Example of a game situation corresponding to the situation encoded for the red ghost above Pacman in table 1.

Second approach - Mixed Strategies Our mixed strategy approach works very similar to our first approach, with the difference that it deals with probability distributions over legal moves which results in a more chaotic but unpredictable behaviour. The individual encoding stores a float value out of $[0, 1]$ with every direction. The variation is done by adding a normal distributed value out of $[0, 1]$ to one randomly selected move probability followed by a normalization of the distribution vector. Everything else works just like in the pure strategy approach. For a more detailed description see [DEH+07] and for our enhancements on this approach see [BHN+08].

Table 1: Encoding of the example game situation depicted in figure 13 by a 13-dimensional bit vector $v \in \mathbb{B}^{13}$ for the red ghost above Pacman. Information on the visible neighbourhood and a short memory are invoked.

0. Is Powerpill active	0
1. Is Pacman left	0
2. Is Pacman down	1
3. Is Pacman right	0
4. Is Pacman up	0
5. Is a wall left	1
6. Is a wall down	0
7. Is a wall right	1
8. Is a wall up	0
9. Was last move left	0
10. Was last move down	1
11. Was last move right	0
12. Was last move up	0

Third approach - Online EA The idea to implement an *Online EA* is motivated by the property of evolutionary algorithms to deliver feasible solutions to the underlying optimisation problem in each iteration. The advantage of using these intermediate solutions is that the algorithm can react fast to all occurring events. Our *Online EA* evaluates thus the calculated directions online at the runtime. Each individual consists of two coordinates and decodes one position in the maze. The mutation operator generates one new position by adding a randomly chosen number from the range $[-5, 5]$ to the individual's coordinates. Figure 14 shows a possible setting.

Since the mutation does not guarantee that generated positions are actually correct, a feasibility test is performed after each mutation.

Each individual is evaluated according to its distance to Pacman. As the ghosts are supposed to chase Pacman, this distance is generally supposed to be minimized, i.e. we are looking for the shortest path from the ghost's position to Pacman. For the purpose of calculating the shortest path, each NPC controlled by *Online EA* has a global knowledge of the maze. Apart from the distance to Pacman the fitness function takes the relative position of the ghosts to each other into account as well as the availability of power pills. The latter factor is included in the evaluation since we expect the Pacman to target the cells close to the power pill. Thus the final fitness function is defined as:

$$fitness = \kappa * \beta + 2.2 * \rho - 1.5 * \varphi \rightarrow min!, \tag{9}$$

where $\beta \in \{-1, 1\}$ denotes whether the power pill is active and $\rho \in \{0, 1\}$ takes the value 1 if there is a power pill on the current tile. The distance $\kappa \in \mathbb{N}$ denotes the length of the shortest path in the maze. The parameter φ controls the distance between the ghosts:

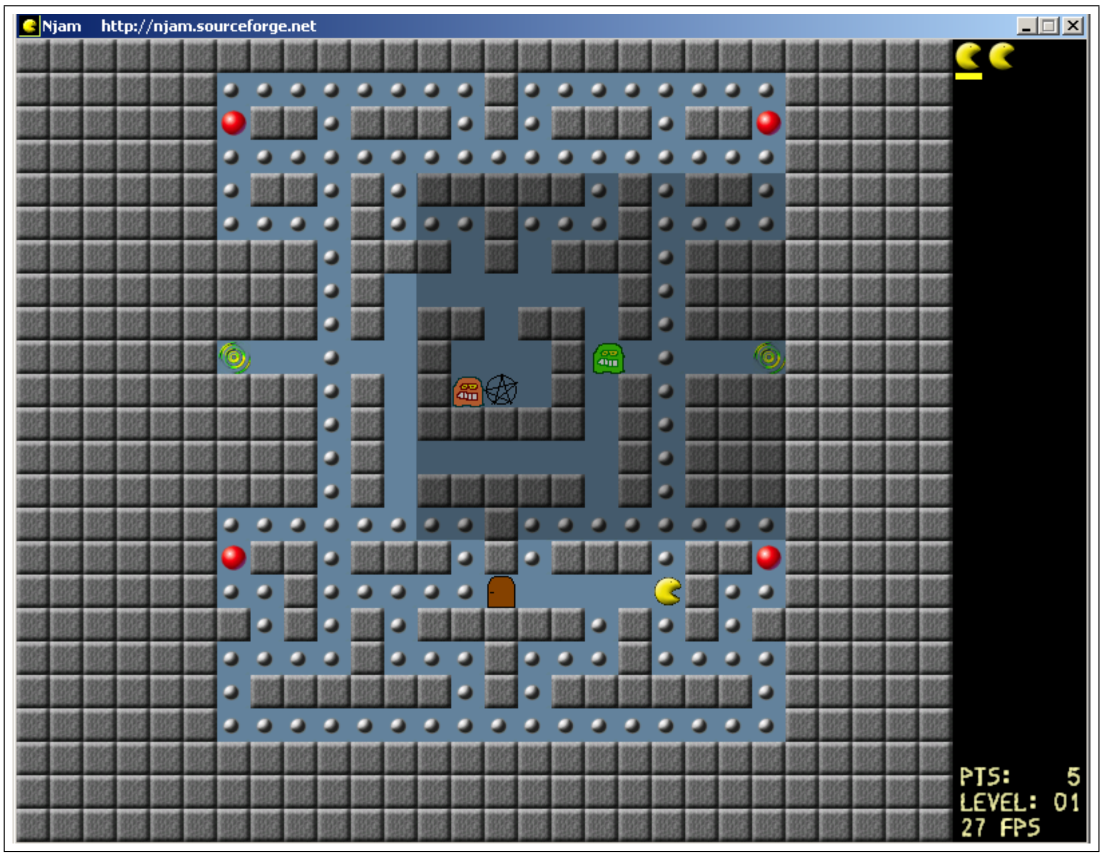


Figure 14: The green ghost can reach all fields in the shaded area.

$$\varphi = \frac{1}{4}\sum_{i=1}^{4} d_i, \tag{10}$$

where $d_i \in \mathbb{N}$ is the distance to ghost i. If the positions of the ghosts tend to approach each other too closely, this parameter will prevent such behaviour by imposing the punishment on the fitness function. The constants are determined by experimentation and common sense.

The *Online EA* proceeds as described in listing 2.

```
1. Is the ghost's position at the center of a tile?
                if yes go to step 2, otherwise END
2. start a (1+1)EA with the current individual
                (5 generations)
3. calculate the shortest path to the position
                of the best individual
4. choose the direction indicated by the shortest path
```

Listing 2: Online EA algorithm

3.2 Deterministic algorithm to control Pacman

To train the ghosts automatically, a NPC for Pacman was needed. The implementation uses a simple rule based approach to decide in which direction to move. The directions are rated according to table 2. A priority queue is used to store the four pairs of rating and direction that have to be compared in a situation. The highest rated direction is always chosen. This approach works good as long as pellets are in sight. When the maze becomes emptier, Pacman has problems finding the remaining pellets, because he has no global information.

Table 2: Rating of directions depending on the visible things and the situation Pacman is in. The direction in which he moved before receives a bonus +0.25.

Rating	Condition
5.5	Pellets in sight, the next tile has a pellet, a ghost is in sight, Pacman has the power pill
5.0	Pellets in sight, ghost in sight, Pacman has the power pill
4.5	Pellets in sight, next tile has a pellet, no ghost in sight
4.0	No pellets in sight, ghost in sight, Pacman has power pill
4.0	Pellets in sight, no ghost in sight
3.0	No pellets in sight, no ghost in sight
2.5	Pellets in sight, next tile has a pellet, ghost in sight
2.0	Pellets in sight, ghost in sight
1.0	No pellets in sight, ghost in sight
0.0	Wall

3.3 Byproducts of the Pacman project

As a voluntary effort, some members of the PG decided to further research topics related to the work carried out on NJam. This resulted in two scientific papers that are to be published on the *IEEE World Congress on Computational Intelligence* (WCCI) 2008:

Measuring Flow as Concept for Detecting Game Fun in the Pac-Man Game The first paper [BDE+08] deals with the fun players experience while playing NJam. It presents a fairly large study with human players. The assumption is that player's fun relates to the psychological *flow* concept [Csi90]. The paper's question then is whether flow is a more reliable measure than to ask human players directly for the fun experienced during gaming. In order to detect flow, it introduces a measure based on the interaction time fraction between the human-controlled Pacman and the ghosts, and compares the outcome to the results of a fun measure suggested by Yannakakis and Hallam [YH04].

To Model or Not to Model: Controlling Pac-Man Ghosts Without Incorporating Global Knowledge The second paper [BHN+08] takes a look at the methods used to implement the ghost controller for Pacman. It concentrates on the CI methods of EAs and ANNs and their attempts to improve the ghosts' behaviour. Since the approach of these methods differs - the EAs learn while actually playing the game instead of learning on an abstract model of the game like the ANNs do - performances are evaluated on a set of custom levels for further comparison. The results show to what extent trained behaviour can be generalized, thus be applied to new situations.

4 Second Project: Glest

"AI is always a minefield, and I'm always disappointed by great strategy games which appear to have the most simple, easy-to-predict AI running your enemies." *Peter Molyneux, Game Designer (1959 - . . .)*

The game we worked on as the second project is Glest, a real time strategy game published under GNU Public Licence (GPL, [5]) by Figueroa et al. [4]. This report is based on Glest version 2.0. By the end of our development Glest version 3.0 has been released with an improved gameplay and balancing. Therefore some of the results and techniques shown here might not be compatible to the new Glest release without further modification.

4.1 Real Time Strategy Games

Strategy games are usually placed in war or battle situations, where all players start with equivalent resources and have to build up a base and an army with the goal of destroying other players' bases and armies. The base usually consists of different buildings for storing collected resources which are needed to build buildings, combat units, and defensive facilities. The army usually consists of different units trained to attack a variety of weapons or attend to specified tasks (such as harvesting resources or spying on the enemy). The exact type of these buildings and units depends on the scenario in which the game is placed. In a mediaeval setting there may be knights, granaries, and towers equipped with catapults while in a science fiction setting infantrymen with laser guns, power plants, and laser turrets may be found. Strategy games challenge the player to build up an army and his base and cover the needs for resources simultaneously.

This genre falls into two sub genres: turn-based and real time strategy games.

In **turn-based strategy games**, the game is divided in rounds. Each player takes as much time as he needs to take his turn and then the outcome of his action is displayed. The actions are carried out simultaneously after each player has finished his turn (for example in the classic game *Battle Isle*[3]).

Real time strategy games do not grant consideration time to the player. The time flows continuously and each action is carried out as soon as it is triggered by the player (for example *Dune 2*)[4]

Fog of War Every player has limited vision, he can only see what is situated in the range of sight of his units. Places which are not in the sight of a unit cannot be seen by the player. This is the idea behind "Fog of War" (FoW). Usually the map is dark when the player starts a level and he gets to know its features by moving units around and exploring it. But when the game features FoW, the player cannot see what happens in the

[3]*Battle Isle* is a well known round based computer strategy game which was released by the German computer game developer *Blue Byte Software* in 1991. Battle Isle featured games of two human players at the same computer or games of one player against an AI-enemy and is considered to be *the* classical round based strategy game for computers.

[4]*Dune 2* is a 1992 released game of the US video game developer *Westwood Studios.* Its new game engine affected following game developers and, even if not being the first real time strategy game, coined the whole genre.

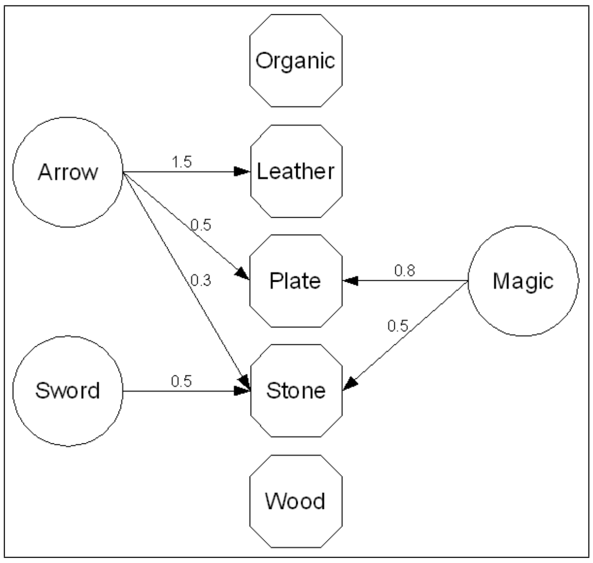


Figure 15: Factors to damage dealt by different sources to varying armour types. All three armour types (circles) dealt damage to all five materials (octagons). For clarity factors of 1 are not displayed.

explored areas unless he leaves a unit there which guards the place. This way the enemy may make movement inside already explored areas without being seen by the player (and vice versa).

4.2 Setting of Glest

Glest is settled in a fantasy mediaeval world. The players can choose to control either a *magic* or a *tech* faction. The magic faction uses wizards as basic units, which can be trained to become powerful warlocks and may conjure dragons and daemons as fighting units. The tech faction uses laborer as basic units, who may be trained to become knights, archers or engineers who are able construct (flying) battle machines. A detailed description of the units and buildings of each faction can be found in appendix B.

4.3 Rules

In order to win, each player has to eradicate all enemy units and buildings. To achieve this goal, he has to build up an army consisting of the given units. Since the economic structure in Glest is a rather simple one, the player only needs to know the techtree of his faction (described in appendix A) in order to know what is needed to build up specific units.

Damage and armour In Glest, the actual damage inflicted by an attack of a specific type of unit differs by the armour type, which the unit is equipped with. The type of armour is fixed for each unit type and given in appendix B. The multipliers for the damage are given in figure 15.

Resources The game economy is based on four resources. Gold, stone and wood are used by both factions and can be harvested by *Workers*, *Technicians* (tech faction) and *Initiates* (magic faction). These resources have to be transported to the nearest storage area, which is a *Castle* for the tech faction and a *Mage Tower* for the magic faction, in order to be used.

Mana, as a form of energy, is required by the magic faction in order to build units and is generated by building *Mana Sources*. Instead of mana, the tech faction needs food for the same purpose, which is gained by building *Farms*, *Cows* or *Pigs*. Unlike gold, wood and stone, mana and food are not consumed during the building process but during the unit's lifetime. This means that the player cannot build any units if he lacks food or mana, and also his units will die if the source of food or mana which they depend on is destroyed. An overview of the resources which are required to build a specific unit or building is given in appendix B.4.

Experienced Units Glest includes an experience model for units, which is based on the number of kills the unit has done. Each unit type has different promotion levels which are reached if a fixed number of enemies is killed, i.e. if a unit reaches certain experience in its field of action. The promotion levels and kills needed are stated in appendix B.3 for each unit type. For each level the unit reaches, its experience, hitpoints and armour class increase by a factor of 1.5, while its sight increases by a factor of 1.2. Note that the damage inflicted by the different attacks is *not* increased by advancing in levels. Units incapable of advancing in levels are *Daemons*, *Initiates*, *Summoners* and *Wicker Daemons* for the magic faction and *Catapults*, *Cows*, *Pigs*, *Technicians* and *Workers* for the tech faction as well as any building.

4.4 Units and buildings

Glest features several units, buildings and enhancements for each faction which are precisely described in appendix B. There are also tables containing the exact values used in Glest. Each unit and building has a property called *hitpoints* (HP) which is a measure of the unit's health. When a unit is attacked, the HP are decreased by the damage value of the attacking weapon. When a unit's HP reach 0, this unit dies.

5 Glest (re)coded

"In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg." *Bjarne Stroustrup, creator of the C++ programming language (1950 - ...)*

While implementing our AI enhancements, we added new core functionalities to Glest and solved some hardware compatibility issues in order to make Glest better fit our needs.

5.1 Bigger Selections

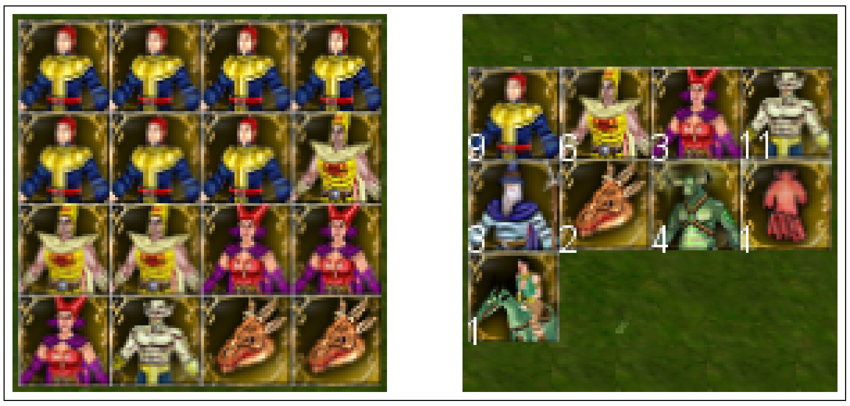


Figure 16: In the original selection of Glest the maximum size of a group is limited to 16 which are displayed as shown in the left screenshot. If there are more units in a selection they will now be displayed as shown in the right screenshot.

During the testing phase of Self-Organizing Maps (SOM), as described in section 2.3, we noticed that Glest limits the maximum number of simultaneously selected units to 16 and so a group has a maximum number of members. We raised this limit to 40 to allow bigger groups. The selected units are displayed in the old Glest style if the number of units in it is less or equal to 16 and are displayed with one unit icon and their quantity number if there are more than 16 units in this selection as shown in figure 16.

5.2 Big Endian binary compatibility

Sound, texture and three-dimensional model files used by Glest are all binary files in little endian byte order. The original Glest code only copied this data into main memory and thus only ran on the little endian processor architecture. To support Apple's PowerPC Mac as well, we ported the code to the big endian architecture. To do this, it was sufficient to use preprocessor directives to activate the adequate code during compilation of methods involved in loading data.

5.3 Cache

After first Coevolution tests, described in section 7.1, we realised that Glest reloads all needed data files on every new game started from the ingame menu. This needs quite a lot of time which slowed down the coevolution runs very much. To solve this problem, we added a static object cache to all data classes whose constructors load data from disk on initialisation. The cache heavily decreases loading time and increases memory consumption by about 30 MB.

5.4 Debugging Problems

Debugging the running game proved to be very difficult. A debug build of the game is approximately ten times slower than a release build. This makes recreating a certain situation which caused the game to crash even harder than it already is, in an application as dynamic as a real time strategy game. Another difficulty is the way a breakpoint interferes with the time measurement of the game engine. When the game is resumed, the time spent in the debugger is added to the time elapsed since the last update of the game simulation. As a side effect, units are "beamed" to their destination, buildings in construction are finished immediately, etc, all because of the long time spent in the debugger (while the game was actually halted!). All things considered, it is impossible to use a normal debugger to debug the running game.

This problem is well known by professional game programmers. The solution is to integrate debugging tools for the ai directly into the game engine, as described by Paul Tozour [Toz02]. Unfortunately, Glest lacks such ingame tools and due to time constraints it was impossible for us to write such tools ourselves, apart from the debug visualization described below in subsection 5.6, which was already a big help.

5.5 Keyboard Layout

We added new keyboard shortcuts to enable SOM grouping, visualize flocking and pathfinding features and disable GFX engine.

V to change the mode continuously:

1. **position** visualizes the actual field of each unit.
2. **cell position** is the same as position.
3. **center position** visualizes the center postion for big units requiring more than one cell because of its size.
4. **target position** visualizes the end of a move command for each unit.
5. **normal path** visualizes the original paths for each unit assigned by Glest.
6. **flock path** visualizes the *Flocking* path (detailed description in section 6.5).
7. **IM** visualizes the influences of each unit (detailed description in section 6.3).

X to stop the graphical display of the game and speeding up the game rate.

Y to continue the graphical display after pressing *X*.

F to give a *Flocking* move command to a group (detailed description in section 6.5).

G to start grouping via Self-Organizing Map (detailed description in section 6.1).

Function Keys (F1, F2, etc.) to choose the groups created by SOM.

5.6 Visualization

In order to illustrate our improvements, we added the possibility to visualize the calculated paths, flock paths and influence values of the map cells.

Paths If path visualization is activated, the cells which will be trespassed by the selected unit are highlighted in a gentle green. For groups, the path of each unit is visualized (see section 6.5, figure 26).

Flock Paths In case of a flock move (see Section 6.5), in addition to the flock path the precalculated average position of the selected group is highlighted. (See section 6.5, figure 26)

Influence Map To visualize the Influence Map (IM) (see Section 6.3), we disabled the map textures and coloured each map cell in green, in case of positive influence values and red, in case of negative influence values. The colour intensity of one cell corresponds to the absolute influence value. (see section 6.5, figure 17)

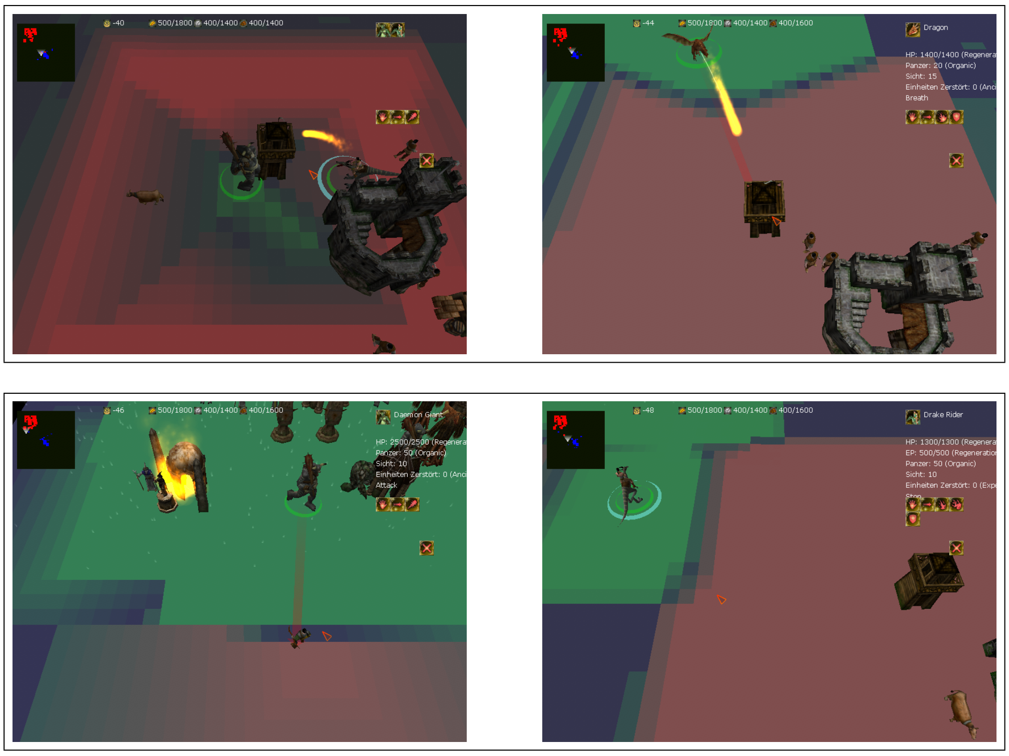


Figure 17: Activated IM visualization in four situations. The calculated influence values are transformed to colour codes and used to colour the ingame map cells. One can easily differ his own influence domain (green), the enemies influence domain (red) and the influence free areas (grey).

6 Local AI Group

"Think globally, act locally." *René Dubos, French-American environmentalist (1901 - 1982)*

The *Local AI Group* dealt with challenges which affect the human as well as the computer player. In the following chapter, we present our enhancements in the unit movement and selection which will result in a more user-friendly gameplay.

6.1 Selecting Units with Self-Organizing Maps

Controlling and grouping units in a strategy game is quite difficult, especially if the number of units increases. We address this problem by an attempt to make the grouping process autonomously. The player can concentrate on strategy issues instead of managing his units, which is also time-consuming. For developing such an autonomous process we choose an unsupervised learning approach, the Self-Organizing Maps (SOM). As described in section 2.3 using SOMs works fine in clustering processes, which is a key part in building good groups of units in strategy games. In this chapter we describe in detail the configuration of the SOM used in Glest and explain how we applied this technique to the building of groups. Finally, we conclude with an assessment of general applicability of SOMs to grouping in strategy games.

6.1.1 Architecture

The SOM's data structure was written from scratch. Although there were plans to port the ANN framework (section 3.1.2) to the Glest project for further use this idea was later discarded. Due to the relatively simple structure of a SOM there was no need to port and adapt the previous work. Furthermore this resulted in tidy code. The SOM consists of a hierarchy of three basic classes which will be described bottom-up in the following.

SOMnode is the basic class in the hierarchy. The class describes a simple node which is used to build up the SOM. Each node holds a set of attributes defining its current position in the two-dimensional grid and a weight vector as well as the respective methods used for internal calculations.

SOM as a class is based on top of the *SOMnode* and defines the actual SOM. Therefore the class manages a vector of SOMnodes beside several other attributes used for the internal computations. Additionally it offers methods for (re)calculating the SOM and grouping entities based on a given set of features.

SOMwrapper represents the interface which connects the game to instances of the previously described SOM. From an arbitrary vector of units the wrapper extracts the needed information and computes the grouping for the set of units.

```
<?xml version="1.0"?>
<som>
    <specification>
    </specification>
    <features>
    </features>
    <grouping>
    </grouping>
    <uma>
        <enemyWeights>
        </enemyWeights>
        <myWeights>
        </myWeights>
    </uma>
</som>
```

Listing 3: Empty frame of a SOM configuration file.

6.1.2 XML Extension

For easier development and debugging a facility for loading a predefined configuration has been integrated. Since Glest already makes use of the Xerces-C++ [7] XML-parser for similar purposes, the choice of a proper format was easy. A configuration includes the setup of the SOM itself as well as the features used for the evaluation. As seen in listing 4 these settings are written in a human readable XML markup and can easily be edited with simple text editors.

Listing 3 shows the structure of an empty SOM configuration. The section of each inner node has to be filled with further information regarding the final configuration of the SOM. Usually each entry uses the same syntax:

$$\langle\ <\text{key}>\ \ \mathit{value} = "\ <\text{value}>\ "\ \rangle$$

The specific keys are described in the following.

- `specification`

  `nodes` defines the number of nodes used by the SOM, the *value* must be a positive integer.

  `splitdistance` defines the split value for building groups after creating the SOM, see section 6.1.3 for more information.

  `mapradius` defines the range of the feature map which is used for calculating the influence for the neighbourhood, see also section 2.3.

  `epoch` defines the number of iterations used for the calculation, the *value* must be a positive integer.

  `startlearningrate` defines the learning rate, the *value* must be a float.

- `features`

```
<?xml version="1.0"?>
<som>
  <specification>
    <!-- size of the som -->
    <nodes value="64" />
    <!-- distance value -->
    <splitdistance value="2" />
    <mapradius value="2" />
    <!-- iterations of som calculation -->
    <epoch value="1000"  />
    <!-- learning rate -->
    <startlearningrate value="0.1" />
  </specification>
  <features>
    <feature value="XPos" />
    <feature value="YPos" />
    <feature value="HP" />
    <feature value="CanFly" />
  </features>
</som>
```

Listing 4: Excerpt of a SOM configuration file used to select units based on their current hitpoints, their location and the ability to fly.

`feature` defines a specific feature of a Unit, *value* can be of *HP* (Hitpoints), *XPos* (x coordinate of current Position), *YPos* (y coordinate of current Position), *Speed* or *CanFly* (a unit's ability to fly). Multiple features are allowed.

- `grouping`

  `weight` defines the weight value for the specific feature. The number of weights defined is analog to the number of features. Normally for grouping the weights are initialized to 1.

- `enemyWeights`

  `weight` defines the weight value for the specific feature used for the *Massive Attack* (see section 7.2). The number of weights defined is analog to the number of features. As described later in section 7.2 *XPos* and *YPos* are used to cluster the enemies. The weights are set accordingly.

- `myWeights`

  `weight` defines the weight value for the specific feature used for the *Massive Attack* (see section 7.2). The number of weights defined is analog to the number of features. As described later in section 7.2 *HP* and *CanFly* are used for grouping units for the *Massive Attack*. The weights are set accordingly.

6.1.3 Application

This part describes the SOM from the view of the player. It shows that creating groups autonomously via SOM is an interesting feature. The workflow can be divided into four sections:

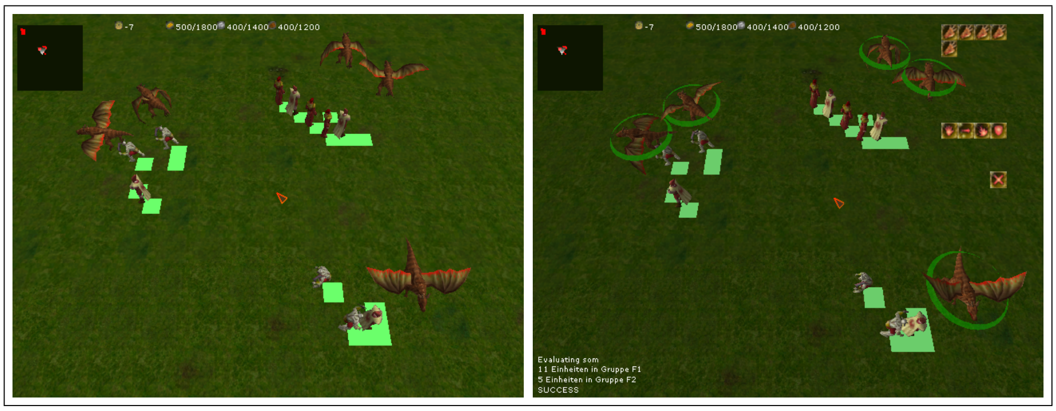
1. The player activates SOM grouping.
2. The SOM is created on the fly.
3. The groups are created.
4. The groups are presented to the player.

SOM activation In the first step the user activates the grouping function simply by pressing the key "G" (for the "group"). Note that this function allows more flexibility in the grouping of the units than the player's manual group selection. Especially during combats it is impossible to manually regroup the units, whereas it is a natural feature of the SOM based grouping.

SOM creation The core of the grouping process is the SOM creation. The unsupervised learning consists of arranging objects (units) with different features, usually multidimensional, in a two-dimensional space (see section 2.3 for more details). In our case four features (XPos, YPos, HP, CanFly) are examined to test the quality of grouping. The SOM receives all movable units the player currently controls and starts evolving. It is also important to mention that this is a randomized process, so the created SOM can differ from time to time when the player activates the grouping function. As a consequence the player may receive different groups even if he activates grouping in a similar situation.

The tested features are the position of a unit (XPos, YPos), the hitpoints (HP) and the ability to fly (CanFly). For testing purposes only one feature (XPos, YPos are handled as one feature) is weighted with 1 and the others with 0. As expected, the units were clustered according to the 1-weighted feature. When more features are considered, the grouping is a more nondeterministic task. However two observations can be made. If the units are close to each other, the HP feature will dominate the grouping process. Otherwise if the distance between the units increases, the coordinate features (XPos, YPos) will start dominating, leading to groups primarily clustered relative to their coordinates. We refer to our videos [6] for a presentation of the grouping process.

Group creation We simplify the multidimensional problem of comparing the unit features by arranging them in a two-dimensional space using unsupervised learning. Normally we would assign every unit to its best-matching unit (BMU, see section 2.3). Units that are classified as similar would then be assigned to the same BMU. Unfortunately we could not control the number of groups and therefore the quality of the grouping, because we calculate the SOM online during the game iterating over 1000 cycles only. So we cannot expect the clustering to be accurate. Similar units can thus be assigned to different BMUs, increasing the total number of clusters.



```
Evaluating som
11 Einheiten in Gruppe F1
5 Einheiten in Gruppe F2
SUCCESS
```

Figure 18: Screenshot of the SOM based selection. The left picture shows a group of friendly units, loosely gathered. The right picture shows a selected group of dragons (encircled) calculated by the SOM. You can see the debug information in the left-hand corner of the screen.

Our algorithm receives the SOM and the units as input and returns a dynamic grouping of the units. For extracting the clustering information from the SOM we use centroids in the two-dimensional space. In our case the number of centroids and therefore the number of clusters is dynamic. The algorithm iterates over all units and looks for the BMU in the SOM. It starts with one cluster, which is initially the BMU of the first unit. We introduce a parameter called *splitdistance* (see section 6.1.1), which controls the number of the groups. A small value of splitdistance results in more groups and, vice versa, a large value leads to fewer groups. The next step in the algorithm checks whether the next unit will be assigned to the existent cluster or a new cluster will be created by comparing the distance of the BMU assigned to the unit with the BMUs of the existing clusters. If the distance is smaller than the splitdistance, the unit will be assigned to the existent cluster, otherwise a new cluster is created with the BMU belonging to the unit. Because the units are already arranged in the SOM only one iteration of the SOM-algorithm is needed for obtaining good clustering results.

Group presentation Following the completion of the previous three steps, the results are presented to the player (figure 18). He receives a visual notification about the new groups and for testing purposes he can select the different groups with the keys F1-F9. As the observant reader has noticed, we can only assign nine groups in the testing scenario. For this purpose a multi start procedure was developed. The termination criterion we use is that the number of groups is between two and nine after the grouping process. This solved the problem of having only nine keys available to which to assign the groups. Due to the iteration count of only 1000 cycles for creating it, the SOM terminates sometimes in a miss, indicated by clustering all units in just one group. In this case, the creation process will be restarted. Naturally the multistart procedure could be generalized for other real time strategy games or other break conditions could be defined. After examining the workflow we see that it is a quite simple process building groups autonomously. The grouping is also accomplished efficiently with respect to the sensitive time constraints within the real-time game, so the player can arbitrarily regroup his units.

6.1.4 Outlook & Conclusion

Our efforts show that the use of SOMs in real-time strategy games has great potential. As a result of our work we could not help noticing that, surprisingly, creating groups autonomously and especially using unsupervised learning methods to handle this task is not a common approach. Although tasks such as the unit selection or grouping can already be solved with SOMs in a satisfying way, there is still room for improvement. Similar games could be a new application area for the SOM-technique. Because of this we decided to continue working on the implementation of the SOM to improve the handling in Glest. The implementation is comparatively easy, however finding a proper configuration and a working set of features might be difficult and needs some further attention. We hope to be able to write a scientific paper about our progress on this topic in the near future. To sum up, we recognize a SOM to improve the AI and thus leading to a better gameplay.

6.2 Pathfinding

Sending a unit across the map over a long distance provides difficulties for many games and of course for real time strategy games as well. Even though efficient algorithms like the algorithm of Dijkstra [Dij59] or A* [HNR68] are well known, they may not be applicable. A*, for example, operates on a graph with places represented as vertices V and the ways between these places represented as edges E annotated with the length of the way. On a map consisting of tiles, each tile is a vertex of the graph with eight ways (edges) to the adjacent tiles. The length of these ways is 1 if the target cell is trespassable or ∞ if it is blocked. A worst case time complexity of $\mathcal{O}\left(|V|^2\right)$ like the one of A* is too large, because units are moved all the time and so multiple path calculations using A* are invoked several times in the game and would slow it down very much. If, to speed up A*, less tiles are taken into account, the algorithm may fail because the path may lead to a dead end. Because of this, we developed the heuristic presented in the following which is based on clipping geometric objects, as it is done in computer graphics. Our goal is to split the paths where a unit can run into a dead end into shorter parts which can be calculated faster and then build the complete path out of this parts to avoid calculating very long paths at high cost.

6.2.1 Precalculation

When a game is started, the tiles on the map which do not allow movement are grouped into rectangular objects, as *axis aligned bounding boxes* of the objects. To find the actual objects on the map, an algorithm called mapScanner is used which works as follows:

The algorithm uses two data structures, an array of pointers, initialized empty, of the size of a map and a vector for storing obstacles, whereas each obstacle contains vectors of adjacent x and y coordinates. Semantically the obstacle vector will contain every obstacle on the map at the end of the algorithm. Each obstacle consists of one or more connected cells, e.g. a mountain range. The pointer map is used as a lookup structure and stores for every cell which is already analyzed an empty pointer or a pointer to the obstacle which is placed there.

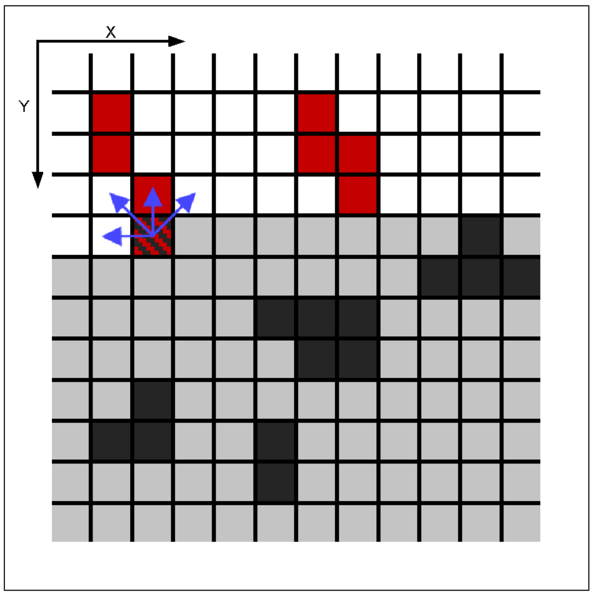


Figure 19: Algorithm for finding bounding boxes. The graphic shows a cut-out of the map, with the algorith in progress, inspecting the cells of the map row by row starting in the upper left corner. Gray tiles are still unseen with light gray cells being free and dark gray being blocked. Red cells are seen, identified as blocked and gathered together as objects, white cells are already inspected and identified as free. The shaded cell is actually being analyzed, the cells pointed at by the arrows are analyzed if they are blocked and belong to an object.

Now the algorithm works as follows: The map is scanned row by row, starting in the upper left corner with the coordinate $(0, 0)$. Each tile with coordinate (x, y) is scanned for its trespassability. If it is blocked, the cells at the positions $(x - 1, y + 1)$, $(x - 1, y)$, $(x-1, y-1)$ and $(x, y-1)$ as visualized in figure 19 in the pointer-array are checked. If one of them points to an object in the object vector, the pointer is copied to the position of the current inspected tile and the coordinates of the actual tile are added to the object that is referenced. If no adjacent object is found, a new object is created with the coordinate of he actual inspected cell and stored in the object vector. A pointer the the object is added in the pointer array at the actual position. If several objects are found, these objects are connected by the actual inspected cell and all objects that are pointed at are collapsed to one. The algorithm stops when the bottom right cell is inspected. Listing 5 outlines the mapScanner algorithm.

After the map has been scanned, the biggest and the least x (column) and y (row) coordinate are selected for each object in the object vector. From these, a bounding box of the object is saved as a *mapObject* in the game. Figure 20 shows how objects (left image) are converted into boxes (right image).

This calculation can be done once when the game is started because the terrain never changes. So the entire calculation is hidden in the loading process of the game an can be used for faster pathfinding later on.

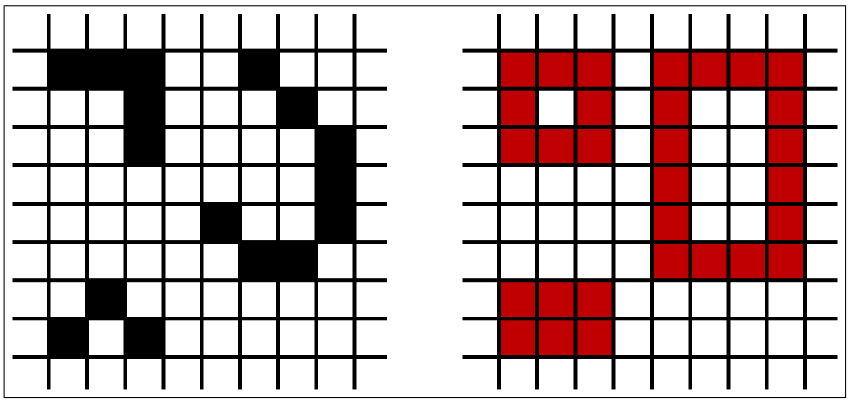


Figure 20: Generation of bounding boxes around objects: The left graphic shows a grid with tiles that are free (white) and and tiles that are blocked (black). The right graphic shows the resulting bounding boxes.

Correctness Since every cell is inspected, the algorithm identifies every blocked cell due to the nature of the look-up process. We examine direct neighbours of all blocked cells. When a block of such cells is identified and it happens to connect two previously identified objects, these are combined to a new object comprising the two old ones and the block of marked cells. So the algorithm finds the biggest possible objects.

Time complexity Every cell is visited once and for every cell, up to four lookups take place. Therefore on a map of w rows and h columns the algorithm which identifies objects has a time complexity of $\mathcal{O}(w \cdot h)$. Selecting the least and the greatest y value can be done in $\mathcal{O}(1)$. Since the map is scanned row by row, the first coordinate added to an obstacle in the precalculated obstacle vector has to be the one with the least y coordinate (being in the top row) and the last has to be the one with the greatest y coordinate (being in the bottom row). To get the vertical boundaries, we traverse the coordinates saved once, checking whether the x coordinate of the current inspected is bigger (lower) than the present found maximum (minimum) by which the maximum (minimum) is replaced with the current value. With l coordinates saved inside the object, this can be done in $\mathcal{O}(l)$. This can grow to a maximum of $\mathcal{O}(w \cdot h)$ if the whole map is blocked. As shown above, the whole algorithm has a time complexity ι of $2\mathcal{O}(w \cdot h)$ and therefore $\iota = \mathcal{O}(w \cdot h)$.

Space complexity In addition to the actual map a map of pointers has to be generated, which will consume space of $\mathcal{O}(w \cdot h)$. The objects consist of vectors saving coordinates which again consist of a single x and a single y value. Like time complexity, the worst case occurs when every tile on the map is blocked, in which case every tile has to be saved, leading to a space complexity of $\mathcal{O}(w \cdot h)$. No more memory is needed and the algorithm has a space complexity of $\mathcal{O}(n \cdot m)$.

6.2.2 Pathfinding

Pathfinding using the precalculated data is a recursive process with two major steps: The algorithm takes two parameters, the start ($\vec{s}$) and target ($\vec{t}$) position as coordinates

$\vec{s} = (x_{start}, y_{start})$ and $\vec{t} = (x_{target}, y_{target})$ and calculates the path from $\vec{s}$ to $\vec{t}$ ($\vec{s} \rightsquigarrow \vec{t}$). The linear distance between these points is checked for intersecting map objects which are already calculated with the algorithm presented in section 6.2.1.

If the direct line does not intersect with an object, the path $\vec{s} \rightsquigarrow \vec{t}$ is calculated. Our original idea was to use Bresenham's line-drawing algorithm [Bre65] for this calculation. Unfortunately, this was insufficient because dynamic objects like buildings and units might be on the calculated path which are not inside the prior calculated *mapObjects* since they did not exists in the phase of precalculation. So the path is calculated with the game's implementation of the A*-algorithm [HNR68] which, in the unmodified, original implementation of the game, uses the euclidean distance as heuristic and cost function. This, in addition, allows the pathfinding to be influenced, as shown in section 6.3.

If there are objects intersecting with the direct line, the object (ξ) with the smallest euclidean distance to the starting point is taken. Starting from this object, the corner closest to the starting position is chosen as a waypoint $\vec{a}$ with $\vec{a} = (x_a, y_a)$. From the two adjacent corners of ξ the corner which is the nearest to the target position, is chosen as the waypoint $\vec{b}$. Now the paths $\vec{s} \rightsquigarrow \vec{a}$, $\vec{a} \rightsquigarrow \vec{b}$ and $\vec{b} \rightsquigarrow \vec{t}$ are calculated recursively and combined to the searched path $\vec{s} \rightsquigarrow \vec{t}$. Figure 21 shows a found path between two points with one obstacle. In order to make this a fair as well as a fast pathfinding method this does not use global knowledge: the potential clipping objects are considered only if they are in the range of sight of the unit's faction.

Known Problems

- **Teleport bug** Overlapping bounding-boxes (see figure 22) may cause the pathfinding to fail because every calculated subpath may clip one of the overlapping boxes. If this happens, the pathfinding algorithm will return an empty movement queue for the given subpath. As described above, the subpaths are combined to a complete path. This is stored in the unit's movement queue. In order to move a unit, the game engine moves the unit along this queue. If, by failed pathfinding, a subpath between two waypoints is missing, the unit gets moved instantly ("teleported") from the first waypoint to the second.
- **Wrong direction** It may happen that the corner of the bounding-box which is the nearest to the starting point leads to a longer path around the box. This is more likely the nearer the starting point is set to the box. The result is a unit walking in the opposite direction to the one needed to reach the target position in a short time. Figure 23 illustrates this problem.

6.2.3 Outlook

To avoid the problems named in section 6.2.2, the pathfinding using bounding boxes could be done using an implementation of A* with a different resolution. In the preprocessing phase for each box the ways between the corners of every obstacle can be calculated and saved inside the boxes together with their lengths. Using this information, a graph can be generated. In this graph, each corner of a bounding box would be represented as a vertex. The prior calculated paths between the corners would be represented as edges, weighted

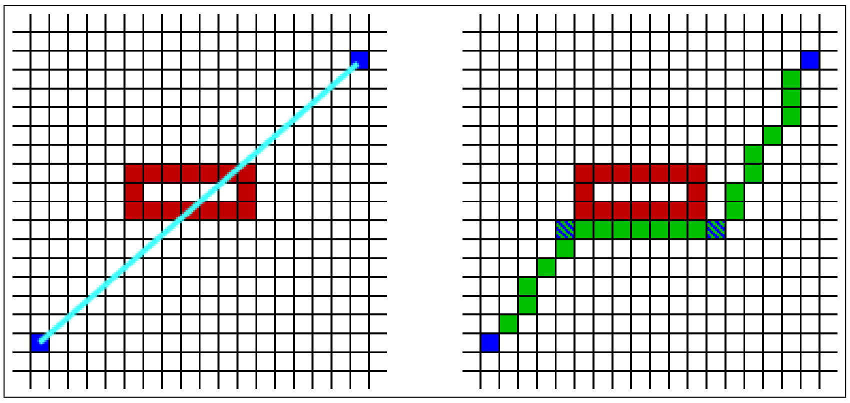


Figure 21: A path drawn by our pathfinding algorithm. The left figure shows the linear distance (cyan) between start- and endpoint (blue) of the path to be calculated, which is blocked by one object (red). The figure to the right shows the calculated way (green) with the waypoints (shaded) found by the pathfinding algorithm.

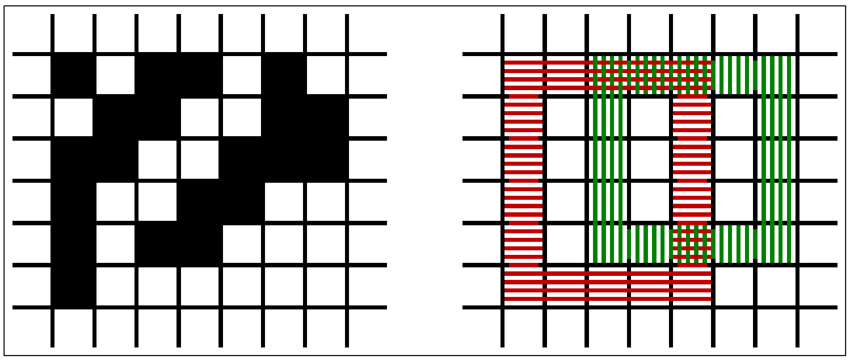


Figure 22: An example for overlapping bounding boxes: The blocked tiles (black) in the left image are combined to the two overlapping bounding boxes shown in the right figure coloured red (horizontal shaded) and green (vertical shaded).

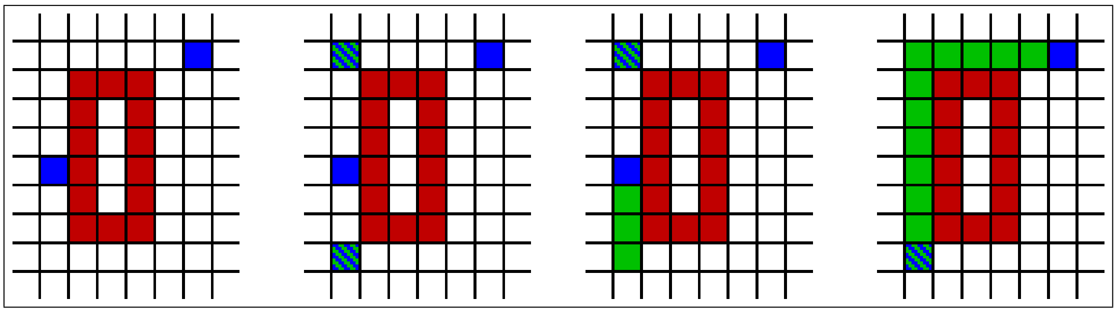


Figure 23: An example, where a unit may walk in the wrong direction because of the clipping-algorithm. The starting point (blue, left) is near the bounding-box (red). The waypoints (shaded green) are calculated as presented (middle left figure) and the way (green) elapses as shown in the middle right and the right figure.

with their lengths. Now our algorithm described above has to get the nearest corner of the obstacle closest to the starting point as well as the nearest corner of the obstacle closest to the destination. Between this two corners the shortest path can be calculated in the already constructed graph. This path is represented by a set of waypoints for which the subpaths between each pair of successive waypoints is being calculated with the A* algorithm.

The maximum number of different objects that may be found on a map with a width of w and a height of h is $\lceil \frac{w}{2} \rceil \cdot \lceil \frac{h}{2} \rceil$ and therefore the number of vertices would be four times as high, since each object is represented as rectangular bounding box with four corners; this would lead to a graph with $2 \cdot w \cdot h$ vertices with $(2 \cdot w \cdot h)!$ edges wich is two times as big as the original graph devised from the map.

Since, as described in section 4.3 playing a game of Glest includes setting up buildings of different sizes, it is to be expected a map that has large free spaces and some, big objects. So we may assume that there are less objects than $\frac{w \cdot h}{4}$, the resulting graph is smaller, which makes navigating on it more efficient. To keep this modification of our algorithm a fair one, which does not use global knowledge, the considered vertices have to be checked against the faction's knowledge of the map and only used if they are known.

6.3 Influence Map

The idea of including influence maps (IM), described in section 2.4, in the game came quite early in the design process. The first implementation of IM was done during the supplementary work specified in section 9.3, but this implementation did not suite the needs to be included into the game. Another implementation was done in the later phase of the project to enable a better pathfinding, which is described in the following section 6.4.

Influence of units Each unit u bears its own influence i_u which is spread out in quadrate-"rings" (Taxicab geometry, [Kra86]) up to its sight radius s_u. This influence is calculated from the units maximum hitpoints h_u and its combat strength c_u defined by the damage d_a which the unit deals with its most powerful attack. A unit may have up to two attacks a_1 and a_2; if a unit only has a primary or no attack, the damage dealt by the "missing" attacks is set to 0. In order to scale nicely with the radius, the combat strength is divided by 100. Equation 11 shows, how the combat strength is calculated.

$$c_u = \frac{max\{d_{a_1}, d_{a_2}\}}{100} \tag{11}$$

The influence is inversely proportional to the radius r of the ring as shown in equation 12.

$$i_u(r) = \begin{cases} \frac{h_u \cdot c_u}{r} & \text{for } 0 < r \leq s_u, \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

It is possible to define other formulas for the unit's influence and its propagation on the map. There are generally no restrictions, which formula should be used for the propagation or the type of the exerted influence, but potentially there is some extra work needed: If the influence should be calculated on the basis of dynamic attributes rather than static, some additional calls have to be made in order to modify the IM adequately.

If, for example, the actual hitpoints are being used instead of the maximum, the game engine has to be modified so that every time the unit takes damage and the influence spread out by the unit has to be recalculated. In order to do this, first the unit's influence is to be removed from the IM, the hitpoints are to be decreased, the unit's influence is to be recalculated and finally again added to the IM. Adding and removing influence to or from the IM is the subject of the following paragraph.

Calculating the map In order to use the influence map for pathfinding, it cannot be calculated in large time intervals but has to be accurate all the time. This could not be achieved by traversing through the whole map, because even the linear time complexity would slow the game down seriously.

Since influence sticks to units, the influence map has to be updated when a unit moves, but only in the small area of its sight radius. Therefore, every time a unit moves to another tile, its influence circle is removed from the unit's old position and added to the new position. By doing so, only $2 \cdot s_u^2$ tiles have to be updated after each movement and not the whole map. Equation 14 shows, how the influence of the cell $\zeta_{x,y}$ at position (x, y) is calculated with units u out of the set of all units U at the positions (x_u, y_u) and distance δ_u (equation 13) to the inspected cell.

$$\delta_u = \sqrt{(x - x_u)^2 + (y - y_u)^2} \tag{13}$$

$$\zeta_{x,y} = \sum_{u \in U} i_u(\delta_u) \tag{14}$$

When only one unit u is moved, the old influence $\zeta_{x,y}$ is updated to $\zeta'_{x,y}$ in the sight range of the moved unit. Equation 15 shows the influence added for the unit being placed on a tile; for the earlier discussed removement step of the unit the unit's influence has to be subtracted from $\zeta_{x,y}$.

$$\zeta'_{x,y} = \zeta_{x,y} + i_u(\delta_u), \quad x \in [x_u - s_u, x_u + s_u],\ y \in [y_u - s_u, y_u + s_u] \tag{15}$$

In general, it is possible to use another function than summation to add a unit's influence to the IM, but it has to be invertible in order to make the removal of the influence from the IM possible. It should also be kept in mind that this function is being evaluated all the time for each action which takes place in the game. Therefore a formula should be used which can be evaluated very fast.

For detailed analysis of each faction's strength and weakness the influence of each faction is stored separately in the IM. The IM grants the possibility to access the influence values on different levels of detail. To achieve this, the IM simulates the logical structure of a quadtree: If you query the IM you give the coordinates at which you want to know the influence together with the level of access. If you query at level one, then you get the most detailed values, namely only for the coordinates you queried. If you query for level two, the quad of coordinates to which the given coordinates belong are being calculated as well as the overall influence for this quad. We actually used simple summation to calculate the overall influence at a cell or at any higher level, but the formula used for aggregation of the different factions' influence at the different levels of the quadtree may be a different one.

Especially, it might be interesting to include the weapon and armour model used by Glest into the aggregation of the influence map. Since we were running out of time this was not included and we did not include any passibility restrictions in the spreading of a unit's influence (for example, a unit being able only to attack in melee and incapable of flying should not spread its influence across the sea). For the same reason, the global AI did not use any abstraction level of the influence map so it can be considered as future work to do some strength and weakness analysis for strategic purposes.

6.4 Pathfinding with Influence Maps

As described in section 2.4, IMs are usually used to make strategic decisions. As stated in section 6.3, our IM always shows the accurate influence for each faction on each tile at any time of the game. Because the influence consists of the inspected unit's maximal hitpoints and its mightiest attack, strong opposing influence shows dangerous areas where units are likely to be hurt or killed.

Since pathfinding, described in section 6.2, makes use of the A* algorithm, the influence can be used to calculate "safe" paths, which means paths crossing tiles influenced by the own faction or with no or only weak influence of the adversary. This can be done by introducing the influence $\zeta_{x,y}$ in the existing heuristic of the A*, which is the euclidean distance δ (equation 16) between the current cell (x, y) and the target cell (x_t, y_t).

$$\delta = \sqrt{(x - x_t)^2 + (y - y_t)^2} \tag{16}$$

In order to prevent the unit from being affected by its own influence at its current position, the unit's influence has to be subtracted from the map as described in section 6.3 and the resulting influence $\zeta'_{x,y}$ is used.

$$\begin{aligned} \varepsilon &= \frac{\zeta'_{x,y} - c_u}{180} && (17) \\ \psi(x,y) &= \begin{cases} -1 & \text{if } \varepsilon < -1, \\ 1 & \text{if } \varepsilon > 1, \\ \varepsilon & \text{otherwise} \end{cases} && (18) \\ \tau &= \delta \cdot e^{\psi(x,y)} && (19) \end{aligned}$$

Since we used the unit's health to calculate the influence, we needed to scale $\zeta'_{x,y} - c_u$ down because the amount of hitpoints of a unit varies very much. 180 has proved as a good value for scaling. In equation 18 we defined $\psi(x, y)$ as our calculated ε and added boundaries to avoid τ becoming zero or extremely high. If τ would become zero, a unit would never reach its destination because A* cannot calculate any cheapest path because every cell would have costs of zero. In the other way around to expensive costs may result in dead ends even in open areas because every sorrounding cell is much more expensive than the one the unit stands on resulting in the unit not moving any more.

This leads to the following behaviour: If the influence is lower than τ, the unit is considered to be strong enough to claim the cell under its own influence and crosses it. Basically, the unit tries to get to the target position by crossing territory with the smallest possible enemy control. If not individual units but a flock is moving, the combat strength of the flock c_f is gained by summing up the combat strength of all units u in the flock F as shown in equation 20.

$$c_f = \sum_{u \in F} c_u \tag{20}$$

Now the following question arises: "What happens if there are enemies situated somewhere on the path?" To answer this question we have modified the basic formula above to include the combat strength of the unit/flock whose path is being calculated: if a unit is weaker than its enemy and if it is possible to pass around him, then the unit will take this safe path. If the safe path is too long it becomes more expensive to take the safe path then to travel through the enemy position and our unit will take the direct path. If our unit is stronger than the enemy, our modification of the formula says: "This is already my territory because I can dominate the enemy." So the unit/flock will take the direct path and attack the enemy on this path with with all possible attacks.

During our work we encountered the problem that a unit seemed to refuse a given move command if it leads the unit into the area of strong enemy influence. This happened because the unit moved as close to the enemy's influence as possible and after that, every other movement closer to the target destination got more expensive than the current position. So the PathFinder comes to the conclusion that the path to the target destination is blocked because it cannot be reached and therefore ended the move command. To tackle this problem we weighted the own unit's influence the more the closer the unit gets to its target position and got the influence ζ''_{xy} as shown in formula 21:

$$\zeta''_{x,y} = \zeta'_{x,y} - \begin{cases} \delta^2 \cdot c_u & \mathit{if}\ \delta < s_u, \\ 0 & \mathit{otherwise} \end{cases} \tag{21}$$

This works fine most of the time but does not solve the problem. Some other possibilities should be taken into consideration but due to time constraints we were unable develop any other formula.

6.5 Flocking

In computer games, NPCs often have to move in cohesive groups rather than independently. In the real time strategy game Glest the player has the possibility to select groups of units and give every unit of this selection the same target position that it should move to. In this case, the player wants the units in a selection not to split but to stay together. Obviously, they should achieve their target as a group and the behaviour of the selection should be realistic, resembling the birds in a flock. We tried to implement this type of movement for a group of units. In the following subsections we describe how the movement is implemented in Glest, our implementation of the flocking rules and their integration into Glest. Finally, we describe limitations of our implementation and give an outlook on further work on the subject.

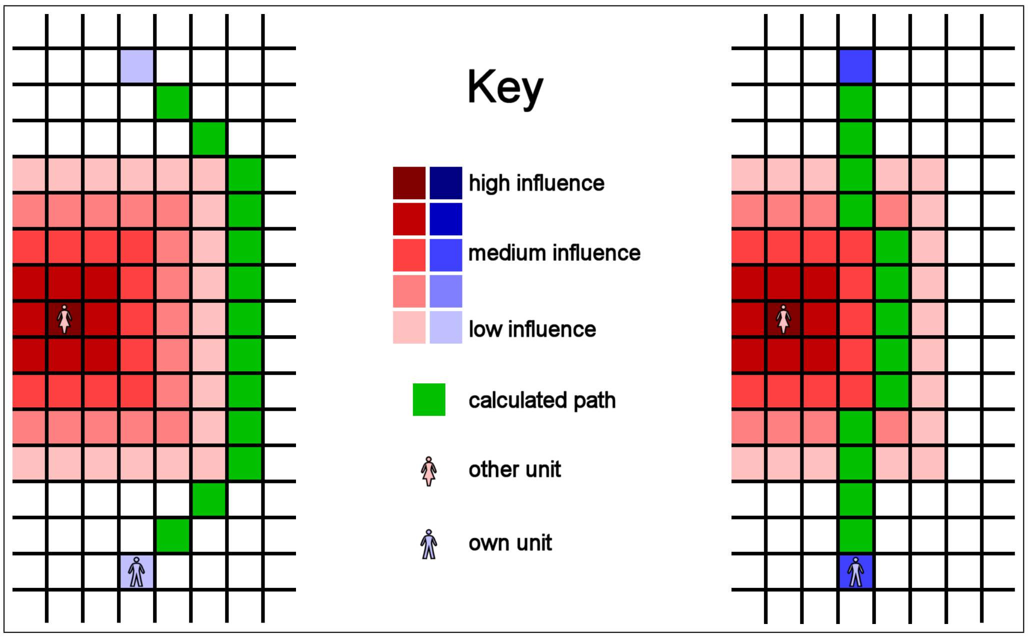


Figure 24: Pathfinding using Influence Maps. The left figure shows how a weak unit completely avoids enemy influence, whereas a unit with medium strength, shown in the right figure, moves through enemy controlled cells with an influence value less than its own.

6.5.1 Movement in Glest

This section describes how the movement of units is implemented in Glest. A basic understanding of this is necessary to understand our implementation of the flock move and some our design decisions.

Map The map of Glest is divided into discrete cells. Only one ground and one air unit can stand on a cell at any time. Most units occupy one cell, only a handful of units have a size of two and therefore occupy a quad of $2 \cdot 2$ cells (see appendix B). A consequence of the discrete coordinates is the fact that a unit can only move in eight directions (see figure 25).

How Glest processes move commands The Glest engine keeps track of its current state by an instance of a Glest::Game::Programstate object. The handling of all user input is delegated to this object and its update and tick methods are called regularly to update the current state.

During a running game, an instance of Glest::Game::Game, which is derived from Glest::Game::Programstate, serves as the state object. The Game class itself delegates most user input to an instance of Glest::Game::Gui, including the mouse click event, which initiates a move command, if a suited unit or group of units is selected. If the position the user has clicked on is empty[5], the handling is further delegated to the

[5]If the position is occupied by an enemy, an attack command is generated.

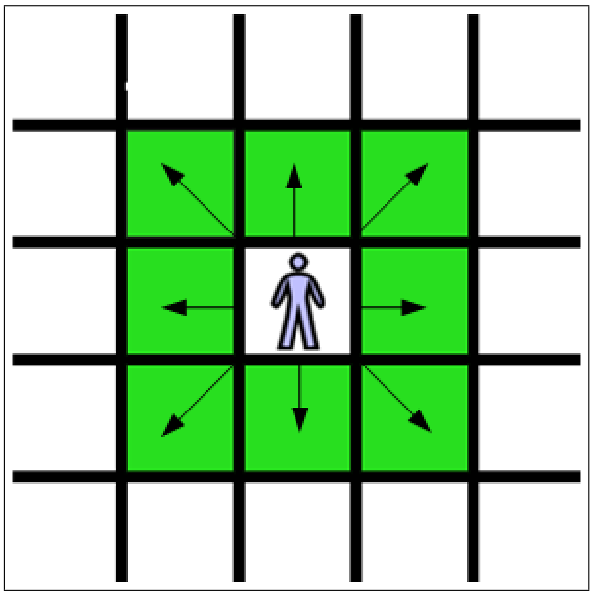


Figure 25: Possible directions for a unit to move.

Glest::Game::Commander class, where a move command is created and added to each unit's command queue.

The command of each unit contains the target position the unit should move to. If a group of units is given a move order, each command created contains a different target position to avoid collisions. Only one unit moves to the position requested by the user, the target positions of the other units are located around this position. The command does not contain a path; the path is calculated during the update of a unit (see below), which takes place independent of the other units. Since the information that a group of units got the same move order is lost during the creation of the move command and due to the fact that each unit gets updated and moves independent from each other, the visual appearance of a group of units moving across the map is unsatisfying: Faster units do not wait for slower ones and in a worst case scenario, instead of the whole group reaching the target position at once, the units arrive one after the other, thus becoming cannon fodder for enemies nearby.

The actual movement of a unit takes place during its update. During each update of the game simulation, Glest::Game::UnitUpdater::updateUnit is called for every unit of every faction in the current game. The update of a unit consists of two steps: First, UnitUpdater calls the update method of the unit. A unit keeps track of the progress of its current action and if the current action has not yet been finished, the update method returns false, to indicate that the UnitUpdater does not need to do anything. In the case of a move command, the action of a unit is to move one step, from its current cell in the map to the next one. Moving from one cell to another includes the rotation of the unit, if necessary. If this action has been finished, Unit::update returns true and the UnitUpdater also performs the second step of the update cycle.

In the second step of the update of a move command, the UnitUpdater compares the current position of the unit with the target position. If the unit has reached its destination, the command is finished and removed from the units command queue. Otherwise the unit is told to move to the next position of the calculated path. If the path is empty, a new path is calculated using the Glest::Game::Pathfinder class. This is the case when the unit gets updated for the first time after it has been given a move command.

6.5.2 Our implementation

Our implementation is based on the idea that we do not want to have a number of boids moving randomly and eventually forming a flock, as described in Craig Reynolds original work (see section 2.5). Instead, the group of units called the flock already exists (the player selected some units) and the members of the flock should reach a given target (also selected by the player). A path to the given target can be obtained using the A* algorithm already implemented in Glest. The flock should than follow this path.

It is a common concept to separate the global pathfinding, which finds a path free of static obstacles (like buildings) from the local pathfinding, which is the movement along the found path, avoiding dynamic obstacles (like other units) as described in [Mat02]. In this context, we reimplemented the local pathfinding of Glest, using slightly altered flocking rules. At first we used the global pathfinding in its original form, later in form of the advanced version described in section 6.4.

Global Pathfinding The Glest engine provides the class Glest::Game::PathFinder, which calculates the movement path for each unit as explained in detail in paragraph *How Glest processes move commands*. A distinction is drawn between a move command for a single unit and a move command for units which are centralized to a group. In the first case, the path starts at the current position of the unit and ends at the target position. In the second case, each selected unit calculates a path which starts at their respective current positions. But the target position of each unit of the selection is slightly different from the chosen target position so that every unit of the selection gets an individual target position and an individual path. To calculate this path for a unit, Glest uses the A* algorithm due to the fact that it can calculate a path between start and target position and the algorithm is quite robust as described in section 6.2. But how do we get the start position for the group of units when every unit is placed at a different position?

We compute the mean position of the group of units and take that position as the start position for the path calculation. Because of the fact that units within a flock should act like individuals in a group, this calculated movement path can be understood as a reference path, which the units of the group should stick to as described in the section below.

Instead of using the average direction vector of the neighbours for the alignment rule2.5, we use a segment of the calculated path, which we call *reference vector* (v_{ref}), to make the flock follow the given path. Since the path is just a list of n positions on the map, it is a natural choice to use the difference of position i and $i+1$ as one reference vector, starting with the vector $v_{ref_0} = pos_1 - pos_0$ when the flock starts to move and ending with $v_{ref_{n-1}} = pos_n - pos_{n-1}$ just before the flock reaches its destination. We call the i^{th} position *last position* and the $i^{th}+1$ position *current position*. As the flock moves along the path, the current and last position have to be updated, to obtain the next reference vector. We do this, whenever the average position of the units comes closer to the current position than a given threshold called *PathUpdateBound*.

Debug visualization The missing ingame debug environment provoked us to realize some visualization to illustrate and debug our improvements. Figure 26 shows some additional information about the flocking realization. First of all we visualize the flock path the units of the group should correspond to. Every cell of this flock or reference path is highlighted in green. In addition the average position of the group is pointed out in a blue coloured cell. At least we have to mention the current path segment which is coloured in red. So we can reconstruct the process of the flock. Figure 26 also shows the standard calculated path for each unit in the group.

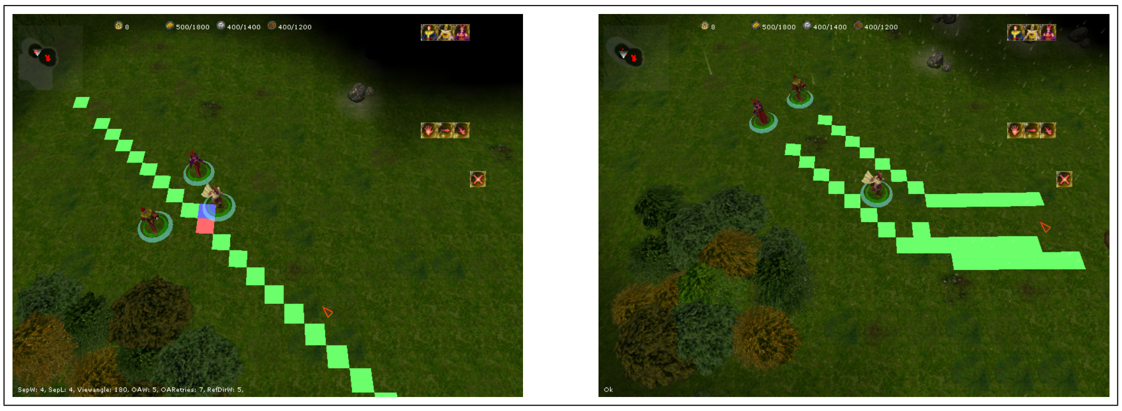


Figure 26: The left picture illustrates the visualization of the flock path. The path of the flock is highlighted in green, the average cell position of the group is pointed out in a blue coloured cell and the cell of the current path segment is coloured in red. The picture on the right shows the standard pathfinding, which calculates a different path for each unit of the group.

Movement After computing the movement path for the group of units the local pathfinding with the flocking method set in. For the local movement it is important to know that the map in Glest is divided into discrete cells. So each unit is located on a cell and can only move in eight different directions to the cells around as you can see in figure 25.

The move direction v of one unit is calculated as follows

$$v = w_{ref} \cdot v_{ref} + w_{coh} \cdot v_{coh} + w_{sep} \cdot v_{sep} \tag{22}$$

where v_{ref} is the reference vector described above, $v_{coh} = pos_{avg} - pos_{unit}$ is the vector from the unit's position to the average position of the flock, v_{sep} is the separation vector and w_{ref}, w_{coh} and w_{sep} are the corresponding weights. If an obstacle is found in the direction v, which is closer than the boundary *MaxSearchDistance*, the obstacle avoidance is invoked. We decided to use a simple heuristic, which is based upon the fact that the reference path contains no obstacles. Therefore, the unit should steer towards the reference path in hope that it will avoid the obstacle. This leads to a new direction v calculated as

$$v = w_{ref} \cdot v_{ref} + w_{coh} \cdot v_{coh} + w_{sep} \cdot v_{sep} + w_{oa} \cdot v_{oa} \tag{23}$$

where v_{oa} is the vector from the unit's position to the current position of the path and w_{oa} the corresponding weight. The weight w_{oa} is inverse proportional to the distance of the obstacle and is multiplied with *ObstacleAvoidanceWeight* from the ini file (see below)

$$w_{oa} = \frac{\text{MaxSearchDistance} - \text{dist}_{\text{obstacle}}}{\text{MaxSearchDistance}} \cdot \text{ObstacleAvoidanceWeight}. \tag{24}$$

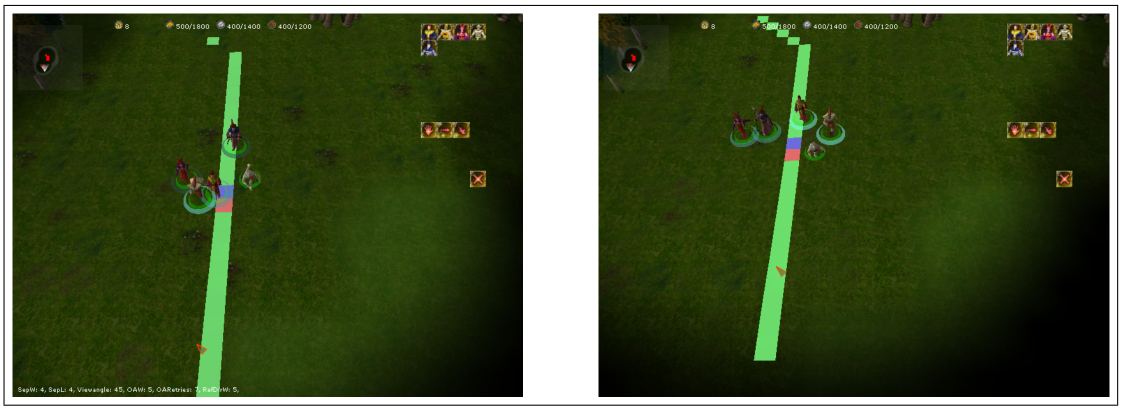
If the new direction is blocked by another unit of the flock, it will wait for a random amount of time. If the unit is blocked by something else, it tries to move to a randomly calculated direction.

To make the moving flock appear even more natural, we also alter the speed s each unit moves along with, depending on its distance from the average position of the flock. This *speed modification* is calculated as follows:

$$s_{new} = \frac{s_{old}}{|pos_{avg} - pos_{unit}|}. \tag{25}$$

Parameters Due to the fact that flocking depends on a lot of parameters that influence the *flock behaviour* of every unit of the flock, a convenient manipulation of these predefined configurations for better testing and debugging purposes has to be provided. Therefore we set up a configuration file named *glestFlock.ini* so that the important parameters can be sourced out and modified easily and we do not have to compile the whole project in order to test the parameters.

As shown in listing 6 theses settings are written in a human readable configuration file and easily be edited with simple text editors and are finally applied in Glest::Game::MrFlock. The specific parameters and how each influences the movement of the flock group is described in the following.



SepW: 4, SepL: 4, Viewangle: 45, OAW: 5, OARetries: 7, RefDirW: 5,

Figure 27: Parameters of the left picture: SeparationWeight = 4; SeparationDistance = 4; ViewAngle = 45. Parameters of the right picture: SeparationWeight = 4; SeparationDistance = 4; ViewAngle = 180. The angle of view of the units causes a different move behaviour of the group. In the left picture the smaller ViewAngle value causes that the group hangs together and each unit follows its unit in front. In the right one the group shows a more scattered behaviour because of the bigger ViewAngle value.

WaitTime: In some situations a unit of the group cannot move to its target position because the position is blocked by another unit. In this case the unit switches into a *wait state* and waits a specified wait time which is calculated by:

$$\text{waittime} = \text{randRange}(0, 15) + \text{WaitTime} \tag{26}$$

The randomized summand makes sure that units normally do not wait the same number of cycles and hinder each other. After that time the unit checks if its new calculated target position is free or is still blocked. If the target position is free the unit will move to this position. In the other case a new WaitTime is calculated. The WaitTime is measured in update cycles and is set to 5 cycles by default.

TimeOut: A timeout is necessary in situations a unit cannot move to the rest of the group and has to be thrown out. TimeOut is measured in update cycles and set to 2000 by default.

PathUpdateBound: Whenever the average position of the units comes closer to the current position than a given threshold, the PathUpdateBound, the current position is set to the next position of the reference path. PathUpdateBound is set to 1.5 cells by default.

ReferenceDirectionWeight: The reference vector as described in section *Global Pathfinding* 6.5.2 reflects the direction of the calculated path for the group. Each unit of the group steers into the direction the calculated path pretends. The ReferenceDirectionWeight offers the possibility to assess this reference direction. The ReferenceDirectionWeight is set to 5 by default.

SeparationWeight: The separation vector as described in section 2.5 reflects the aim of each unit to avoid hitting its neighbours. A higher weight improves this direction. Manipulating this weight causes a significant change of the behaviour of the units in a group. SeparationWeight is set to 4 by default.

SeparationDistance: This value characterizes which units around the current unit are considered for calculating the separation vector. Only the units within the SeparationDistance are considered. The SeparationDistance is set to 4 by default.

CohesionWeight: The cohesion vector as described in section *Global Pathfinding* reflects the direction of the unit to the average position of the group. Each unit steers to the mean of the group. The CohesionWeight is set to 1 by default.

ObstacleAvoidanceWeight: The ObstacleAvoidanceWeight offers the possibility to assess the avoidance direction which is given by the avoidance vector as described in section 2.5. The ObstacleAvoidanceWeight is set to 4 by default.

ObstacleAvoidanceRetries: ObstacleAvoidanceRetries is necessary in situations, where the real obstacle avoidance with the help of the avoidance vector (see ObstacleAvoidanceWeight) has failed and a unit of the group is blocked by a static obstacle. So this unit tries a random direction which is given by a random direction vector. ObstacleAvoidanceRetries controls how often a flock unit tries to move with the help of a random vector. The ObstacleAvoidanceRetries are set to 5 by default. If the maximal trials using the random vector are reached, the unit is removed from the flock.

```
=== Properties File ===

WaitTime=5
TimeOut=2000
PathUpdateBound=1.5
ReferenceDirectionWeight=5.0
SeparationWeight=4.0
SeparationDistance=4.0
CohesionWeight=1.0

ObstacleAvoidanceWeight=4.0
ObstacleAvoidanceRetries=5
MaxSearchDistance=7

ViewAngle=180
```

Listing 6: Illustration of the configuration file used to set the different parameters that influence the behaviour of the group.

MaxSearchDistance: MaxSearchDistance constricts the distance a unit can look in front to search for obstacles. The MaxSearchDistance is set to 7 by default.

ViewAngle: With the help of this parameter a change can be made between different kind of fields of view. For example, a narrow field of view or a wide field of view. In the first case, the group behaves like a snake and each unit follows its unit in front and in the other case the group shows a more scattered behaviour. The effect of different ViewAngle values can be seen in figure 27.

6.5.3 Integration into Glest

This section describes how we integrated the ideas described above into the existing Glest Code.

Expanding the move command To be able to quickly compare the old movement behaviour and our new flock move, we started by defining a modifier key: If the player holds down the 'f' key while issuing a move command, a state flag is set in the Gui class. This flag is passed to the Commander class, which then creates a flock move command instead of the ordinary move command.

For several reasons, we decided to expand the original command class and use the existing move skill of the units, instead of writing a new command. It is our understanding that introducing a new command would have required a new skill for each unit. The consequence of introducing a new skill would have been that every data file belonging to a certain unit type had to be modified (the new skill had to be added) thus becoming incompatible with the normal, unmodified Glest version which we deemed undesirable. Using the normal move skill and command class also kept the code changes to a minimum:

We added a flag *isFlockMode* and a query method *isFlockMove* to the command. By doing this, we are able to execute a different code path, when the handling of a flock move differs from the handling of a normal move, keeping the original movement behaviour intact.

Several helper functions were added to the Command class, for example to calculate the average position of the flock, and others. The movement using the flocking rules is implemented in the UnitUpdater class. If a normal move command has to be updated, the old method updateMoveCommand is called, otherwise our new method updateFlockMove is invoked.

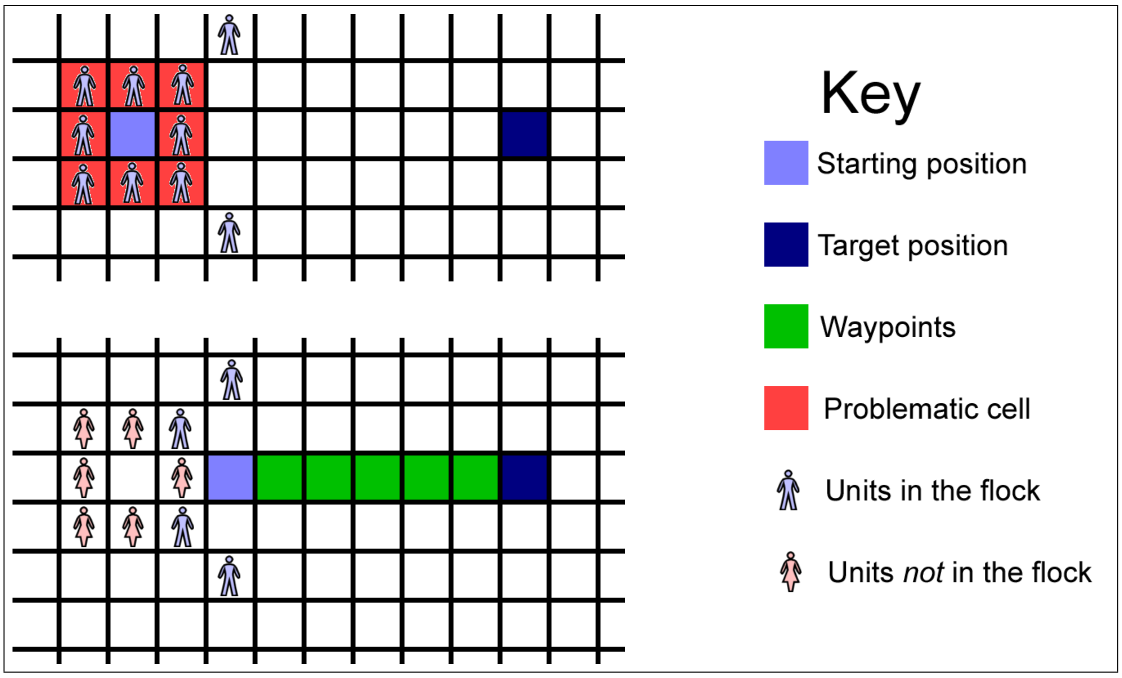


Figure 28: A situation, where the calculation of the flock path fails: The group in the upper picture completely encapsulates its average position, which is the starting point for the pathfinding. Since the units are regarded as obstacles, no path is found. The bottom picture shows the same situation after some of the units have been updated and got removed from the flock, because the pathfinding failed. Since some units have been removed from the flock, the average position changed and a path is found. The remaining units start to move, leaving the other units behind.

Known problems Finding a path for the flock still imposes a problem in certain situations: As described above, the pathfinding starts at the average position of the whole group. If the units stand near each other (see figure 28), the pathfinding might fail, because the average position is surrounded by other units and these are regarded as obstacles. If no path around the obstacles can be found, the pathfinding fails. In this case, the unit during whose update cycle the pathfinding for the flock group failed is removed from the flock group. Since no path has been found, the next unit which gets updated is responsible for calculating the path for the whole group, which might fail as well. As one or more units are removed from the group, the group's average position alters, the path is finally no longer blocked and a way is found (see the bottom graphic in figure 28). The group starts to move, leaving the units removed from the flock behind.

Obviously, the units within the flock should not be regarded as obstacles. We tried to alter the methods which are called by the A* algorithm to determine if a cell in the map is blocked by an obstacle. If a cell is occupied by a unit which belongs to the flock, the modified methods return "no obstacle found". After these changes, the behaviour became better but the problem did not disappear completely. We assume that we actually did not modify every method necessary but due to time constraints, no further work was spent on the issue.

6.5.4 Conclusion and future work

Our present work show that steering groups of units with flocking can provide a natural behaviour in real time strategy games. But there is still some work for us because some special cases, as described in the previous section, are not perfect. Furthermore our results would look better if the units can move to continuous coordinates and not only in eight different directions so that the flocking behaviour looks more realistic and fluent.

Although the Glest code is readable the missing documentation causes that everything needed to be reverse engineered and the missing ingame debug environment makes testing an development hard.

But after all there is still room for improvement especially getting a selection of units to move as a natural flock, cohesive can be seen as the basic task to be solved. As mentioned above further time could be spent on finding better parameter sets.

In the end we have to mention that some refactoring concerning our implementation might be useful because the current code is the result of a lot of experimentation and should be reorganized. Some bugs remain to be fixed but due to the limited possibilities to debug the game at runtime we do not intend to do so.

```
objectVector : VECTOR<mapObject>
maxX : WIDTH_OF_MAP
maxY : HEIGH_OF_MAP
pointerArray : ARRAY<Pointer<mapObject>>[maxX, maxY]
                                        // init each as NULL

FOR (y = 1 TO maxY){
  FOR (x = 1 TO maxX){
    IF(isBlockedCell(x,y)){
          adjacendObjects : VECTOR<Pointer<mapObject>>
      FOR EACH ([x-1, y], [x-1, y-1],[x, y-1],[x+1, y-1])
               AS adjacendCell{
        IF(NOT_NULL (pointerArray[adjacendCell])){
          PUSH pointerArray[adjacendCell]
               INTO adjacendObjects
          addCoordinateToObject([x,y],
                                &pointerArray[adjacendCell])
          pointerArray[x,y]=pointerArray[adjacendCell]
        }
      }
      IF ( SIZE_OF(adjacendObjects) == 0 ){
            newMapObject : NEW mapObject
        addCoordinateToObject([x,y], newMapObject)
        pointerArray[x,y] = newMapObject*
        PUSH newMapObject INTO objectVector
      }
      IF( SIZE_OF(adjecendObjects) > 1 ){
        joinObjects(adjacendObjects)
        updatePointerArray()
      }
    }
  }
}
```

Listing 5: Pseudocode-outline of the of the mapScanner-algorithm

7 Global AI Group

"It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something." *Franklin D. Roosevelt (1882 - 1945)*

The "global" subgroup dealt with those issues of Glest's AI concerning global decision making and strategic behaviour. The Glest AI is a deterministic rule-based one with a fixed set of parameters controlling the activation of each rule (see listing 7). The main drawback of determinism in this case is the predictability of the opponents behaviour, which significantly lowers the fun after playing only a few games.

Having identified several sources of suboptimal decisions which the AI makes during a typical game, we decided to tackle two of them. On the one hand we tested whether there are learnable configurations of parameters in the rule base. An evolutionary approach was chosen to accomplish this first milestone in the hope that the winning strategies will survive over less well performing ones. On the other hand the simplicity of the attack rule led to rather naive – though at first efficient – attack strategies. The basic attack principle was to send all available military units to fight as soon as any opponent's unit is in sight. This strategy, also called *rush*, was improved by a novel idea of using self organizing maps (see section 7.2.3).

7.1 Coevolution in Glest

Using coevolution (background information on coevolution is given in section 2.1) to evolve players' strategies follows the evolutionary processes in nature where individuals develop depending on other individuals in the environment. They can either evolve together by cooperating with each other or compete against each other. A successful strategy in a game is the one that beats the others, thus we can evaluate the quality of a strategy by comparing it to others. Coevolution allows to deal with the population of game strategies in exactly this manner. Since tactical games are competitive by design, we evolved the AI strategies using coevolution like described in section 2.1. Due to time constraints, the coevolution has been run for the tech faction only.

```
Precondition:
  * At least one idle unit exists,
       which can mine resources

Perform:
  * Select randomly one idle unit
  * Select resource,
       which is currently least available
  * Send unit to nearest resource mine
```

Listing 7: Sample rule RefreshHarvester

```
;=== Properties File ===

CoEvo=0
CoEvoGen=2
CoEvoLambda=5
CoEvoMinimap=0
CoEvoMu=5
CoEvoGameEnd=50000
CoEvoTest=0
```

Listing 8: Parameter configuration for coevolution

7.1.1 EA implementation

Coding the population For simplicity we decided to leave the set of rules unchanged. This limited us to coevolving the rules' parameters. Each individual represents two valid parameter sets of the rule base. We divided the game into two phases: The first is the "building" phase and describes the beginning of the game. It is characterized by the high amount of units built (both characters and buildings) and the concentration on harvesting resources, without taking any notice of the opponent. The second phase begins with the first battle, since the main concern of the AI afterwards is to gain military strength. We expect that the AI focuses on different goals (resource harvesting in the first one vs. building combat units in the second) and thus behaves differently.

Variation New individuals are generated by a crossover between two parents. The crossover is implemented as an n-point crossover, with full sets of parameters for each rule chosen interchangeably between the parents. The resources are mutated by shifting some amount from one resource to another, maintaining the property of a probability distribution. In all other cases a variable is mutated with a probability of $\frac{1}{n}$, where n is the number of variables. Integer values are mutated by adding a uniformly distributed number from $\{-1, 0, 1\}$. Floating point numbers are mutated by adding a normal distributed number with a mean of 0 and a variance of 0.04.

Selection The μ Individuals with best fitness are selected for the next generation. In case of equal fitness values an individual is selected randomly.

Configuration All parameters that are relevant for the coevolution runs are specified in the file named "GlestExtd.ini" (see listing 8). The usage of this file simplifies varying the parameters of the EA (number of parent individuals "CoEvoMu", number of offspring "CoEvoLambda" and number of generations "CoEvoGen"). Additionally, we have specified the flag "CoEvoGameEnd" whose value stands for the number of update cycles after which the game will be forced to end. This forced ending prevents games from getting stuck and running for an arbitrary long period of time without any of the opponents being able to win the game. The remaining flags ("CoEvo", "CoEvoMinimap" and "CoEvoTest") have been used for testing purposes only and their value is irrelevant.

Initialization We decided to use a (5+5)-EA with elitistic selection and fitness sharing. The choice of $\mu = \lambda = 5$ is a compromise between runtime and population size. In a first attempt, the starting population contained individuals with parameters of the standard Glest AI. For the sake of higher diversity, we then decided to randomly initialize the individuals.

Fitness Function The fitness of an individual corresponds to the outcome of the game. If an individual's AI wins the game, its fitness is increased by 1. Since this fitness assessment leads to circularities among the individuals (an effect described in section 2.1), fitness sharing is employed at the end of each generation, like shown in equation 4. The basic idea is to reward individuals that won against seldom beaten ones in order to maintain high diversity.

7.1.2 Results and Discussion

We expected that the amount of harvested resources is a good indicator of the individual's performance. Furthermore, we expected that the variable *building ratio*, which controls the amount of triggered build-rules, differs between the two game phases (before and after the first attack).

The results are obtained by running automated games for a predefined number of generations. Since one evaluation of the fitness function requires a completed game, running the EA is very time consuming. The effect of a certain parameter setting is evaluated subjectively by an observation of games against a standard Glest AI as the reference, because an appropriate measure of an individual's quality is not obvious.

Building The parameter *building ratio* is used by one of the AI's rules to trigger the construction of new buildings. It works simply by checking if the proportion of buildings in the total of all units is below the building ratio threshold. Alternatively, the rule can be triggered by the absolute number of buildings falling below a threshold. If one of those is fulfilled, a building type is determined by another mechanism and finally built. The order to produce buildings is as follows: First the defensive buildings, then the buildings for warrior productions and finally the storages are built.

As stated earlier, we distinguish between two different game phases. Figure 29 shows how an individual's building ratio thresholds for the two phases developed during an evolution over 34 generations after they have been randomly initialized. At the beginning of the game there are only a few buildings. This is of course the expected state, since it's the player's task to build more. The evolution results in a low threshold in the first phase, encouraging the AI to quickly expand its base. The high value of the threshold in the second stage of the game indicates that most of the buildings in the base are present. Thus, the AI spends the available resources on the production of units rather than buildings.

Harvesting For the sake of isolated observation of the harvesting behaviour we have excluded all parameters but the harvesting distribution from the evolutionary process. All results related to harvesting have been obtained from a randomly initialized population

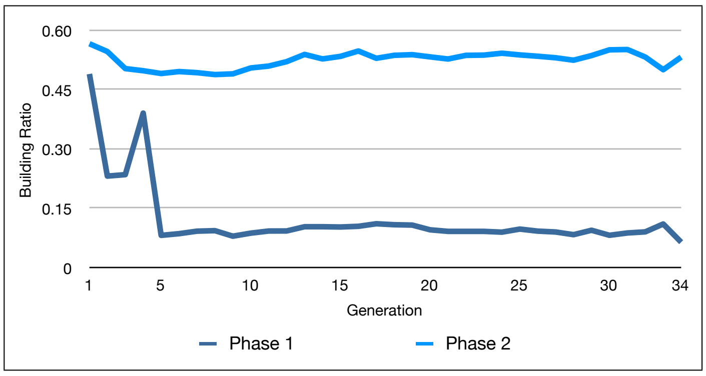


Figure 29: The favoured values of the building ratio threshold are different between two stages of the game. A low value leads to more initiated building tasks, whereas a high ratio suggests that building is less important. The graph shows how the randomly initialized thresholds reach a rather steady state after five generations only.

evaluated over 27 generations. To understand the obtained results, it is important to explain how the learned parameters are being processed. Originally, the AI was simply harvesting the current most urgently needed resource. Trying to introduce some planning, we added information about how much of a resource should generally be kept in reserve by the player. This information would then be learned by evolution and was represented as a probability distribution over the three affected resources gold, stone and wood. We also set it off against the current distribution of stored resources to consider the real demand. In equation 27 is shown how the learned distribution l and the distribution of current available resources c are combined to a resulting distribution r. All three are probability distributions over the three resources, meaning $l_i, c_i, r_i \in [0,1]$ and $\sum_{i=1}^{3} c_i = \sum_{i=1}^{3} l_i = \sum_{i=1}^{3} r_i = 1$.

$$r_i = \max\{0, l_i - c_i\} \tag{27}$$

After calculating the r_i, they have to be normalized to fulfill the condition $\sum_{i=1}^{3} r_i = 1$. Then the resource to harvest can be drawn randomly from the probability distribution r and a worker is assigned the task to harvest it. The learned distribution over the resources is somewhat counter-intuitive. Figure 30 shows a tendency to keep stone or wood in reserve rather than gold. We would, however, expect that gold is the preferred resource, since it is needed in a significantly larger amount than the other two (see also the resource requirements described in techtrees appendix A).

Since the learned distribution contradicts the common sense expectations, we investigated, if the amount of harvested resources is affected by the learned distribution at all. As already mentioned, we expect the winning strategy to harvest more, in general. Figure 31 shows a comparison between the amount of each resource of the winner and the loser AI. Generally, the winner is harvesting more resources. Related to this effect, we have to mention, that if the end of a game is caused by our forced timeout, we declare the AI with more resources to be the winner. The impact of these games is, however, not significant because it only occurs in four games out of 45 on average. The figure also

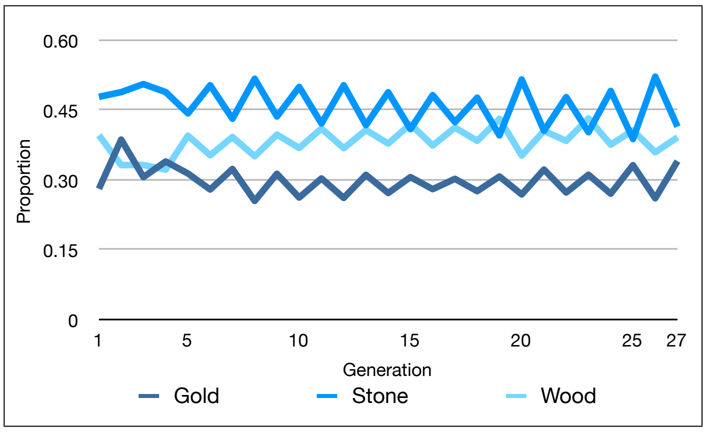


Figure 30: The distribution for harvesting of resources shows no learning progress over time. The values are averaged over the survivors in each generation respectively.

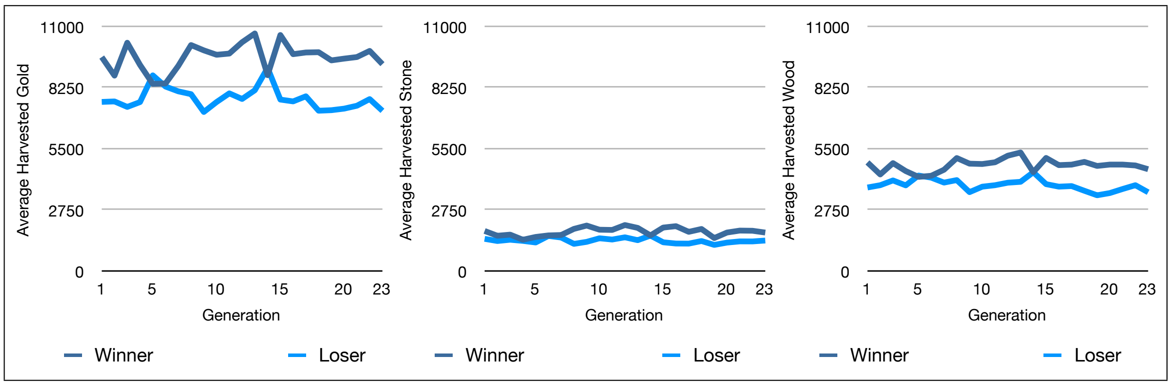


Figure 31: Average amount of harvested resources. The winning AI is generally harvesting more resources than the losing one. The diagrams also show that gold is needed most, followed by wood and stone.

shows that the amounts of harvested resources do not change much over generations. So, we conclude that the learned distribution has no real effect on the player's behaviour. If this is due to a programming error or the frequent execution of the harvest rule by the AI remains an open question.

Observations The ultimate test for the coevolved AIs is obviously its performance in a game against a reference AI. The reference is, logically, the standard AI as implemented in Glest, since we want to show the improvements of our development. In the test setting, we let one of the individuals from the last available generation play against the standard AI, assuming it to be the best available one. The experimental method we employed was "test by direct observation" and included observing the games between the standard and our coevolved AI. The games have been evaluated subjectively by 7 PG members. We observed all possible combinations of tech and magic factions. Generally speaking, the trained AI used as tech faction wins most games against tech factions. Since we only trained the AI as tech against tech faction, we expected this outcome. The observations showed that the trained AI shows different behaviour at the first stage of the game. The trained AI first tries to collect as much resources as it can and mainly develops the base during this period. Military forces are developed after the game has been running for a while.

The learned strategy is a non rushing strategy, whereas the standard AI follows a rushing strategy so that in test games the first attack is always performed by the standard AI. Defending this first rush is a difficult task for the trained AI, but always worked out in the end. After the rush, the trained AI had a better starting position in terms of resources and a further developed base. This defines a turning point in the game and the trained AI beats the standard AI. The weak beginning with respect to the military strength makes it difficult for the trained AI to defeat the magic faction.

The magic faction has the ability to create many daemons rather fast and without exhausting resource consumption by the summoners. This leads to very fast rushes by the magic faction, which cannot be beaten by the trained AI. It was discussed by the group if this behaviour is a bug in the original game because we would rather expect well balanced factions. The remaining test cases are those in which the trained AI (trained as a tech faction) plays the role of a magic faction against both the standard magic and tech faction. Again nearly all games against the magic faction are won, which can be explained by the building-stage. In games against the standard tech AI the trained AI using the magic faction has much more problems. Our interpretation is that the magic faction is more powerful in rushing strategies, but less effective on long term. As soon as the techs have developed an adequate base, they win the game due to their technological dominance.

Discussion One main problem is to understand the meaning of the resulting parameters learned. The rule base with all parameters mutated is a very complex subject and the resulting behaviour can be hardly described theoretically. In order to reduce complexity we also trained the AI only by mutating the distribution of the resources. Observing the behaviour of such AIs shows that resource distribution is the main factor of improving AI behaviour. The test games are won in the same way as with all parameters changed.

7.1.3 Future Work

One major unhandled issue is to adapt the performing strategy to the enemy behaviour. This implies an analysis of the game state, which includes rating the enemy state and the own state. For simplicity we distinguished only two situations, which are separated by the amount of existing units. Of course, this static rule provides no interaction with the current game state. Increasing the number of distinguished game states alone will have no influence on adaptation anyway. Instead, an online learning process needs to be established. As coevolution uses no information about the underlying problem, it would probably be to slow for such a task.

7.2 Ultra Massive Attack

The standard AI implemented and used by all opponents in Glest consists of 13 rules. Every rule contains a time interval specifying the frequency at which it is being tested. So, this approach is analogous to *polling* (see [Tan01, page 285]) in electrical engineering and computer science. (In section 7.2.3, we propose to use an event driven approach instead to simplify enhancements to the game.)

We tried to improve one of these rules called *Massive Attack*, which worked as follows: As soon as an opponent's unit gets within the range of vision the *Massive Attack* rule is triggered. The system then decides whether the home base is stable or not, i.e. whether the player has enough warrior units to defend the home base. The second factor to be considered is whether the opponent is seen close to the home base or not. So there are different outputs depending on these two factors.

- The base is stable $\Rightarrow$ All combat units are sent to the field where the opponent's unit was seen.
- The base is not stable and the opponent is seen out of the base range. $\Rightarrow$ Nothing happens.
- The base is not stable. The opponent is seen inside the base range $\Rightarrow$ All units including workers give up their current command and start to defend their home base.

To trigger this rule the original AI usually sends a *Scout Patrol* to the known opponents' starting positions all over the map to scan the areas they reach for the adversary units. The scouts are thus bound to find an opponent rather soon. As the human player is able to memorize the deterministic starting positions after few games, too, we do not consider this knowledge of the AI as cheating.

7.2.1 Modifying the Attack Rule

We started to rewrite the *Massive Attack* rule to create an ***U**ltra **M**assive **A**ttack (UMA)* and therefore had to modify the *Scout Patrol* rule as well. The *UMA* focuses on the attack on the enemy base only. For defending issues the old *Massive Attack* is used and it only triggers in case an enemy attacks within the base radius.

The UMA is closely interrelated to the scouting. We changed the way of sending scouts to find the adversary base. First, we increased the number of scouts sent out. We expect to receive more information on the strength of the enemy and the distribution of his units by sending out more scouts. This information is used for further decisions in the UMA. In order to improve the scouting only the fastest units are selected as scouts, because fast units are able to collect more information in less time. Additionally, scouts are not sent all to the same target position but rather to slightly different positions. This strategy leads to a broader covering of the enemy base and each scout gathers different information.

As previously stated, the UMA is closely coupled with the scouting. Due to the aforementioned lack of an event system, the UMA is only triggered if all scouts are killed, being the only available information. Another aim behind this method is that we try to gather as much information as possible in order to decide how to perform the attack. The UMA is restricted to the old *Massive Attack* in the way the attack is performed. Up to now all units with an attack skill were sent to the target position of the *Massive Attack*. They showed no trace of structured or intelligent behaviour. Neither was the enemy analyzed in any way nor an intelligent attack position was targeted. The new UMA tries to take this aspects into account.

The main idea behind the UMA is that the enemy units sighted by the scouts are dynamically clustered by their position. Clustering is realized by a Self-Organizing Map (SOM, see section 2.3). The clustered groups can then be analyzed in detail and each group stands for a possible attack position. The goal of the clustering is to classify enemy groups by a certain feature (for example strength), so that an appropriate group of our own units can be assembled, which is expected to be able to defeat the considered adversary group.

Due to time constraints and fast approaching deadlines, we were only able to implement a simple, manually designed mechanism to partition the combat units appropriately to face the adversary's units. The clustering of the UMA's units is again realized by a SOM. Unlike the hostile units, the UMA's units are clustered by their health points and their ability to fly. A fitness function then assesses the appropriateness of a group against an enemy group. As shown in figure 15, there are five different types of armour in the game. Firstly, the percentage of health points that belongs to flying units y in the group is calculated. The fitness value f is designed to give information which enemy group is best matched to a specific own group. It is calculated as

$$f = \sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{s=1}^{S} (m_{ts} \cdot a_{us} \cdot f_s). \tag{28}$$

In this equation, U is the number of units of the own group, T is the number of armour types in the game, S is the number of attack skills unit u possesses, m_{ts} is the damage multiplier between armour t and attack skill s (see figure 15) and a_{us} is the attack strength of attack skill s of unit u. Fitness value f_s, defined by equation 29, reflects the probability that an attack skill is useful against a specific enemy group.

$$f_s = \begin{cases} 1 & \text{if } s \text{ is a field and air attack,} \\ y & \text{if } s \text{ is an air attack only,} \\ 1-y & \text{if } s \text{ is a field attack only} \end{cases} \tag{29}$$

In the matching process, each group is assigned an enemy group. Hence, like the original *Massive Attack*, an UMA once triggered always takes place, no matter how hopeless it is. After matching each enemy group, the assembled own groups are sent to attack their matched enemy group. This offers the opportunity to attack at specific positions with grouped units, which are able to defeat the resident hostile units.

7.2.2 Implementational Issues

Implementing the new UMA did not require many changes. Firstly, we had to restrict the existing *Massive Attack* so that the *Massive Attack* only triggers in case of defending the own base. Further on, only the function *sendScoutPatrol* was modified. The changes extend the function in a way that more scouts are sent out to random positions. The main problem was that we had to define a point in time, when the *UMA* is triggered. Glest contains an observer design pattern for units that can be used to monitor the lifecycle of a *Scout Patrol*. A “kill-event” was used to make a snapshot of the last seen hostile units. When the last scout is dead, the UMA is started.

7.2.3 Future Work

A problem that still remains untouched is to decide if an attack is promising at all. This could probably not be done without redesigning the entire attack rule. Another problem is that the information used to design the attack might be outdated when the groups arrive at the battlefield. This is of course mainly due to the hostile units' mobility. It is difficult to track the progress of an action, because Glest lacks a proper event system and the AI lacks any kind of memory. Every given order is immediately forgotten. Especially the omission of the event system complicates adding functionality a lot. Also, it negatively affects the game's runtime because in every game cycle the AI has to iterate over all units. Instead, a data structure containing all active units should be maintained. This would allow to leave out all idle units.

One important general issue of a *Massive Attack* is the group behaviour. It is not very intelligent to attack the enemy by sending the own units to specific positions. The attack should always be performed as a group. This implies that the units act as a group and move together to the attack position. Otherwise units arrive at the enemy base one after another and can be killed easily.

As previously discussed our project group introduced a method named *Flocking* (see section 6.5). The integration of the *Flocking* into the *UMA* did not work smoothly. The idea was to move the assembled groups with a flock move. This procedure did not work properly so that further work has to be done. Main problems were that the *Flocking* method has problems with grouping units, which are locally dispersed, and moving several flocks through tight terrain.

8 Conclusion

"A year spent in artificial intelligence is enough to make one believe in God." *Alan J. Perlis , American computer scientist (1922 - 1990)*

In the second project carried out by the PG511, the usability of the CI methods was tested in the complex real time strategy game Glest. The originally available AI was modified both on overall strategic level (described in chapter 7 in detail) and on local level of single or groups of units (see chapter 6 for details).

Although we are quite happy with the results, there are a few issues, both positive and negative, worth mentioning now that we are able to look back and reflect on the entire project. To begin with, it has to be stated that our version of Glest includes some interesting new features. The player now has the possibility to group his units automatically by using SOMs (see section 6.1) or to see different influence areas of his and adversary's units by visualizing the influence map (see section 6.3). Thanks to the implementation of flocking for the control of motion over the map (see section 6.5), the groups of units are now moving to their shared target position as a group rather than each unit at its individual speed (thus possibly and probably splitting the group and leading to a man by man annihilation). These concepts are rather new in the range of real time strategy games.

Our modifications of Glest were implemented without using available libraries. This was a deliberate decision since we had the impression that the overall learning effect both in improving programming skills and in understanding the CI methods would suffer. Additionally, by using libraries like the boost library [1] we used in the first term would have unduly increased the projects weight.

On the downside, there is a whole list of difficulties which we identified during the work with Glest. Firstly, there are insufficiencies in the implementation of the game itself. On the whole we can say that the usage of CI methods in any aspect of the implementation needs to be considered within the original design. Especially the runs of the coevolution (see section 7.1) were influenced by the lack of possibility to manipulate the game timers and thus reduce the duration of the games for some significant period of time.

Further on, we badly missed proper debugging functions. If a game is started inside a debugger and halted on a breakpoint, the ingame clock is not stopped but continues running which makes debugging of ingame methods quite difficult. We tested our methods mostly by starting a game in a modified setting and by careful observation of the behaviour. The setting at the beginning was chosen to suit the intended test: the number and types of units were varied as well as the outlines of the maps. Additionally, a variety of visualization modes was implemented in order to make new properties in motion control (6.5) and calculations of the influence map (6.3) visible and thus enable some debugging.

Our version of Glest shows some deficiencies concerning its performance. Especially when all new features are active, the performed calculations are taking too long to complete. The code is not optimized regarding the runtime of single functions, that is because we accepted some computational overhead of quadratic or cubic runtime when better runtime would have probably been possible. This problem has not been solved, partly because of the approaching end of term and partly because the necessary modifications would require most of the game code to be restructured completely. It would have been easier, however, if Glest was equipped with a proper event system that kept the history of events during the game for individual units.

Although the implementations were tested and demonstrated for each group individually – and we find that they show some impressive behaviour – the dependencies between all new features has not been tested thoroughly. The short tests we ran and the performance problems we met led to the conclusion that the code in its current version needs lots of fine tuning. Some functions that work fine in their original setting tend to disturb other functions. If, for example, a unit reaches the enemy terrain and its influence map shows that it is easily overpowered by the enemy, the unit will strive not to enter the enemy territory. But if this unit happens to be a scout, patrolling the map for information on an enemy, we want it to get killed in order to trigger the UMA (like described in section 7.2). Opposing tendencies of this type are difficult to track since it requires time consuming observations and figuring out all possible test scenarios.

To sum up, we consider our work as enrichment to the existing game. Although CI methods do not play a big role in commercial computer games yet, the methods presented in this report are well suited for their tasks. After dealing with Glest for one semester we strongly discourage further work on this game if it is only for a short period of time. Our impression is that Glest offers great potential for improvement and that implementational efforts should be granted sufficient testing time.

9 Review

“Parting is such a sweet sorrow.” *William Shakespeare, “Romeo and Julia”, act 2, scene 2*

As stated in chapter 1, a project group is a course that every student of computer science in Dortmund has to take. One has to apply to offered PGs ordering them by personal preference and submitting his academic transcript of the intermediate examinations. From the applicants, eight to twelve students, which are considered qualified to cope with the demands of the particular topic, are accepted. If possible – which usually is the case – the students' personal prioritization is respected. By this, a group of students is cobbled together by their marks and preferences. This may lead to a group not being able to work together because of different characters – a problem which luckily has not occurred during our project phase.

9.1 Seminar Phase

In the first part of the PG, seminars are held by the participants in order to introduce the PG to the topics related for the upcoming work. The written compositions were submitted to the tutors for review. Since there was no cross-validation of the reviews, the results differed a lot, depending on the actual reviewer. While some issues were tolerated by one tutor, they were chalked up by another. However, the overall rating was quite positive and a good exercise for the upcoming work during the following year.

The actual seminar phase was held during a two day stay in the Universitätskolleg Bommerholz, which is the conference/guest house of the TU Dortmund. During this phase, the presentations were held. Every presentation was followed by a feedback round in which the participants and the tutors commented on the style of presentation and slides. A tense situation at first, since the estimated expectations were pretty high, but it eased up during the course of the day. The group finally started to warm up at the end of the first day at the casual gathering in the in-house bar. Unfortunately the tutors did not take part which would have been appreciated by the participants, nonetheless it was a great evening. Due to the residential seminar the participants got to know each other better and the otherwise time-consuming process of seminars was shortened a lot. We think this was a great advantage over other PGs holding their seminar phase at the University over a couple of weeks.

9.2 First Semester

As already mentioned in chapter 1, the PG phase was split into two projects planned to last for one semester each. In the first semester, we evolved Pacman's adversaries. A detailed description of our work is given in chapter 3 and our CI report [DEH$^+$07].

Working in the first phase seemed a bit over-directed. Reports on the progress had to be given in every meeting, which were scheduled twice a week. At the end of every meeting a feedback was given to the person moderating the session with comments on the overall course of the meeting and its outcomes. This practise, which was thought to teach us better skills in actually running meetings, sometimes seemed to be criticism for criticism's sake (it was said, that "the climate was bad" or "the meeting was boring"). While our skills in meeting moderation improved (a lot, actually), the feedback round shortened and was later dropped in favour of more urgent issues.

In the beginning, our tutors regarded it as an obligation for the students to be present at every meeting. When one member was unable to take part in the very first meeting (for good reasons), he was given a formal warning, that this behaviour would not be tolerated again. That struck us as being a bit disproportionate, since everyone might be unable to attend a meeting at one point or the other during a one year period, be it for personal reasons, illness or work. Luckily, one member of the PG intervened and reminded our tutors of the fact that a PG should regulate itself as long as possible without outside interference. An agreement was settled, that a member can be absent from a meeting with valid excuse, if he notifies the group of his absence in advance and the PG does not veto the planned absence.

During the semester, we gained the impression that one participant's work as well as his motivation were far below the others'. We conferred about this and scheduled a meeting without the tutors to discuss the matter between ourselves, giving this person a clear warning that his efforts will have to intensify significantly because we did not feel it is fair for eleven students to do the work of twelve. He apologized and assured to do more stressing that he had to sit several examinations during this term but would do much more in the second semester. The apology was accepted, the topic therefore dropped and we returned to work to achieve the given goal as a group of twelve.

We completed our modifications in time and showed the upgraded game on the *Campus-fest*, the university's open day. The visitors who played against our ghosts were asked to fill out a questionnaire by which we tried to verify the formula of Yannakakis and Hallam [YH04] which measures the perceived fun in a game. The verification did not work out because the data contained to much noise, we blamed this effect to our questionnaire and the disturbing surroundings of our stand. Some participants of the PG decided to make another try at the verification and submit the outcomes to the IEEE World Congress on Computational Intelligence[6], while another subgroup wrote a paper about the techniques used which was to be submitted to the same conference. Section 3.3 deals with those papers both accepted for the conference.

The last task of the first semester was to write an intermediate report about our work, which was published as CI report [DEH+07]. The semester finished with an agreement to start the second semester with a kickoff meeting at Bommerholz and to report our results of the first semester in front of the colloquium at our chair.

9.3 Second semester

In the second semester, the PG modified the real time strategy game Glest. As decided, we started with a two day residential in the Universitätskolleg Bommerholz where we split into two subgroups, each dealing with the local or global AI. These are described in the chapters 6 and 7. During this meeting, one of our participants annoyed the PG as well as the tutors by not having prepared his part of the presentation for the colloquium at our chair (which was to be rehearsed during the residential) as well as not keeping the decided

[6]`http://www.wcci2008.org/`

schedule and being absent from a part of the meetings, which lead to an extra task of writing the minutes for the next five meetings and a formal warning.

Although not being a draconic punishment, the PG had problems with this decision because our problematic participant mentioned earlier did not, as promised, work more but rather went on with his slack participation. So this problem was put to a meeting schedule. The PG as well as himself was given an opportunity to put forward their opinion on his working manners and he was given the task by the tutors to write down his share of the work in the first semester to check whether his efforts were sufficient or not. This report was discussed with the PG, which showed he did significantly less work than the others.

In order to assess his achievement individually, he was given the task to implement an *Influence Map* for Glest using Fuzzy Logic. His work was timed to last two weeks and at the end he had to report about his achievements. These were noted as not being adequate to compensate his lack of commitment and initiative in the past so the tutors decided that it is not possible for him to pass this course any more, which resulted in him leaving this PG and starting in a new one.

Considering the events it might have been our fault to some part as well. Trying to settle occurring problems internally without the help of our tutors was the wrong decision. Their role within the PG is supporting us if needed. Addressing the problems in a public discussion might have been less time-consuming and less stressful.

During the first two months of the second semester, we worked on the tasks discussed during the residential, and some participants voluntarily worked on the mentioned two papers (see section 3.3), dealing with the techniques used to control the ghosts and the problem of measuring and calculating fun or flow during a game against the modified adversaries. Although it was a very interesting work, doing research for writing a conference paper and the actual writing was a new experience for all participants and consumed quite a lot of time. Our tutors' support proved invaluable: they reviewed the text, gave leads on doing the actual research and used their reputation at the university to help us. Nevertheless our concentration and time was split between the two tasks and the time used for the submissions was taken at the expense of the PG's project of enhancing Glest.

In addition, the subgroup dealing with pathfinding wanted to use the influence map in their work. This was slowed down by the extra task of the special student for about three weeks because implementing it in the subgroup would have made the task obsolete. Sadly, movement was crucial for the other groups, so the whole modification aimed on getting units from one tile to another slowed down the work of the flocking group as well.

At the end, the pathfinding was completed with delay and the subgroups could integrate it in their methods. Due to the problems described in section 6.2, the movement, and therefore the behaviour of the units, was far away from being perfect.

9.4 Expectations and experiences

The expectations before the project started were quite similar among the participants. When asked about the personal expectations and reasons for taking part the most common answers included the high interest in methods of CI and their application in computer games and gaining insight for further studies or, if possible, a master thesis. Everyone was highly curious about the next year and the achievements to come, although we already knew a lot of work had to be done.

Taking a look back on the past months most of us agree that our expectations have been delivered. Working on a pretty new area of research and solving certain problems in an interesting and unique style the project was a huge, yet personal success for everybody. However comparing the pro and contra arguments for the PG as an educational course, the opinions sometimes differ among the participants. We had a lot of fun while working on our project which has been supported by the PG itself. Of course we had no influence on the composition of the group and its constellation must not always work out that well, yet we had the luck that we soon became a team of colleagues working on a common goal. Improving the soft and social skills like organization, leadership, scheduling tasks and moderating discussions and meetings are just some of the things a PG offers. However the time and work that is needed to achieve the goals set must not be underestimated. Twenty hours a week in addition to the official meetings were quite common, even more prior to pending deadlines. It has to be considered that there is not much time left for other courses beside the PG.

So we close saying that this course is a difficult and hard one but useful in the academic training and thank ourselves for a year of interesting and learnful work and our tutors for providing this experience for all of us.

A Techtrees

"Character cannot be developed in ease and quiet. Only through experience of trial and suffering can the soul be strengthened, ambition inspired, and success achieved." *Helen Keller, US blind & deaf educator (1880 - 1968)*

Techtrees describe which unit, building or technology has to be available to build a specific one or, the other way round, which advancement is made and which units are enabled by constructing the given building or unit or by researching some subject. The techtrees of the two factions in Glest are shown in figure 32 for the magic and in figure 33 for the tech faction.

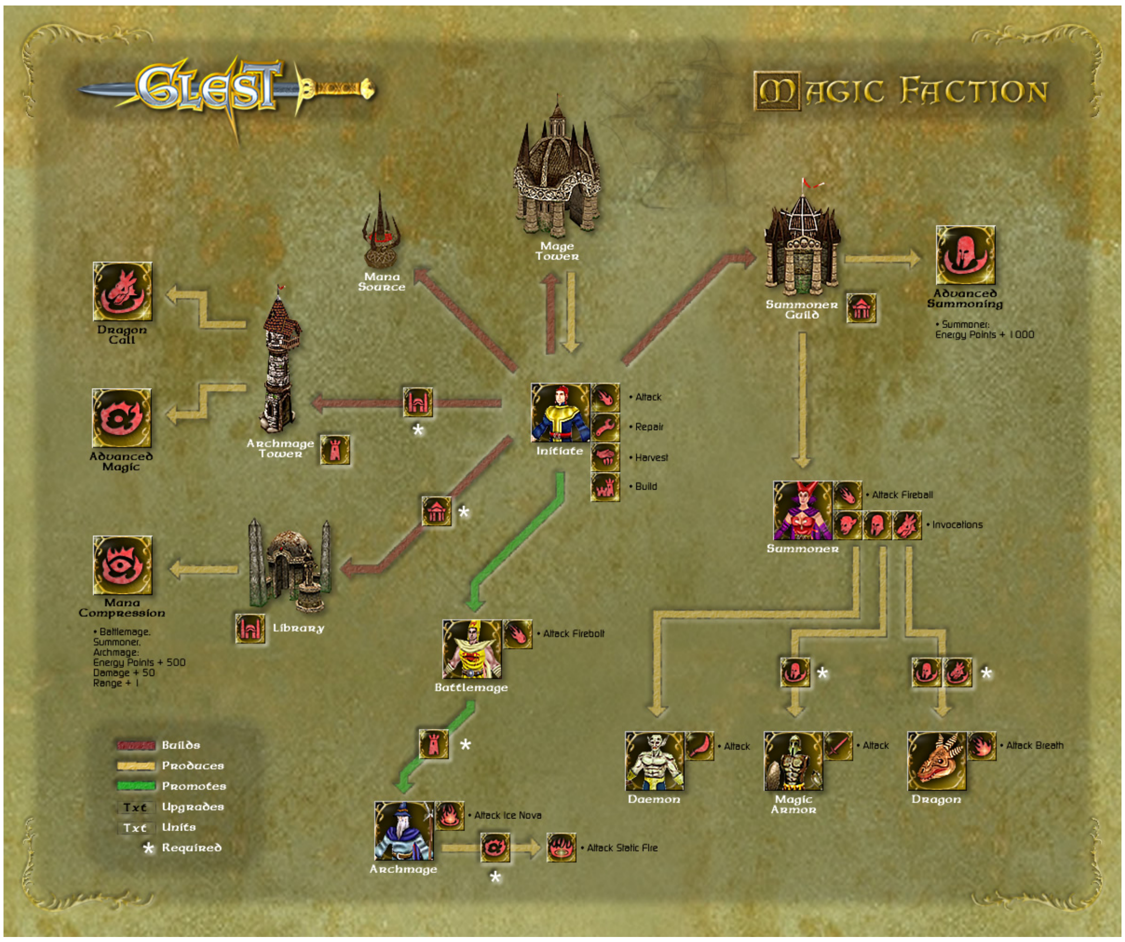


Figure 32: Techtree of the magic faction. The graphic shows the techtree of the magic faction, it was taken from [4]

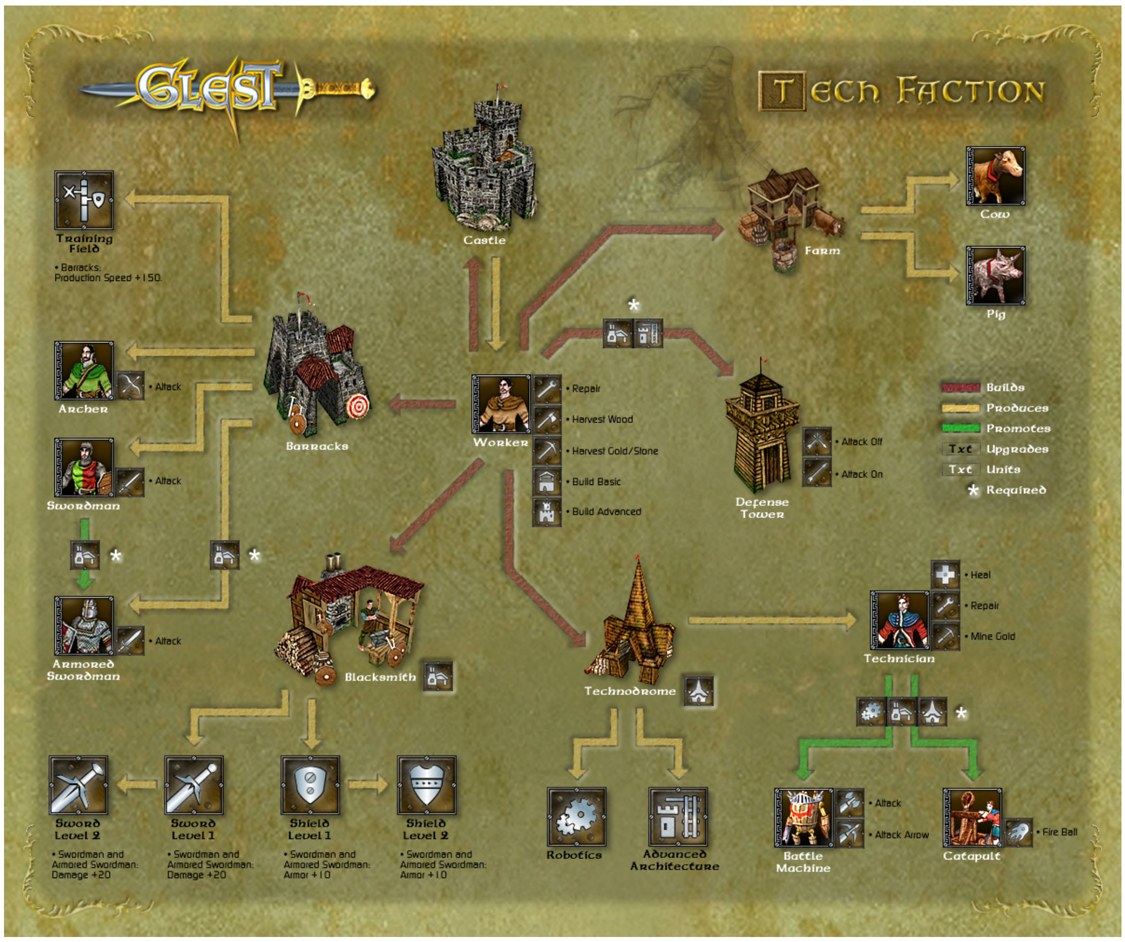


Figure 33: Techtree of the tech faction. The graphic shows the techtree of the tech faction, it was taken from [4]

B Glest Units

“You can’t say that civilization don’t advance, however, for in every war they kill you in a new way.” *Will Rogers, US humorist (1879 - 1935)*

B.1 Unit description

B.1.1 Magic faction

The magic faction can build 9 units, all of which can be used as fighting units, 6 buildings and 1 unit that is considered as a building (the *Golem*).

Buildings

- ***Archmage tower*** The *Archmage Tower* can be build if the faction is in control of a Library. *Archmage Towers* enable the *Dragon Call* and the *Advanced Magic* enhancement.
- ***Golem*** For game purposes, the *Golem* is considered as a building, though it is capable of moving. Its movement is slow and energy points consuming, and a *Golem* might only take 10 steps of continuous movement before it is forced to stop and regenerate. It attacks by throwing stones at opponents’ ground units, making it a moving defense device with quite high damage. In mediaeval terms, the *Golem* can be compared to a siege tower.
- ***Libraries*** are needed for the mana compression enhancement and to build *Golems.* In order to build *Libraries*, a *Summoner Guild* must have already been built.
- ***Mage Tower*** The *Mage Tower* is the central building of the magic faction. In towers, resources are stored. *Mage Towers* are needed to build *Initiates.*
- ***Mana Source*** Mana is the special resource needed to build any magic unit. Each *Mana Source* grants the player 14 points of mana.
- ***Summoner Guild*** The *Summoner Guild* is the building which produces *Summoner* units. Here the enhancement *Advanced Summoning* may be researched. A *Summoner Guild* is needed to build *Libraries.*
- ***Tower of Souls*** The *Tower of Souls* is the magical *Defense Tower* dealing quite high damage against flying units. Since flying units are achieved later in the game, the *Tower of Souls* is too expensive to be build in the early game but may be useful if the game lasts longer. A *Library* is needed to build the *Tower of Souls*, as well as a *Summoner* and therefore the *Summoner Guild.* If the player controls a *Tower of Souls* he may upgrade his *Daemons* to *Daemon Giants.*

Units

- ***Archmages*** are powerful fighting units. Equipped with *Advanced Magic*, they deal the highest damage of all units in the game both at flying and at ground units. Their regular attack damage is quite high as well. Their only weakness is the low hitpoints value, so if an *Archmage* is attacked, it will be killed very soon.
- ***Battlemage*** The *Battlemage* is the second best fighting mage and can be evolved from the *Initiate* without further buildings or enhancements needed. With *Libraries* build, he may be promoted to *Archmage.* His attack deals damage to ground and flying units as well and deals mediocre damage.
- ***Daemons*** may be conjured by *Summoners.* They are basic fighting units dealing about the damage of an *Initiate* but having more hitpoints.
- ***Daemon Giant*** The *Daemon Giant* is an evolved *Daemon* and quite a strong unit for face-to-face combat. He hat thrice the hitpoints of the *Daemon* and deals about twice the damage.
- ***Dragon*** The *Dragon* is one of the most powerful units of the magic faction. It is capable of flying and deals high damage.
- ***Drake Rider*** The *Drake Rider* is technically a *Summoner* riding a saurian. He bears two attacks, one dealing mediocre damage to ground units at medium range and one dealing medium damage to flying units at long range (which is identical to the *Summoner's* attack).
- ***Initiate*** The *Initiate* is the basic unit of the magic faction. *Initiates* are used to harvest resources, construct or repair buildings and can be upgraded to become *Battlemages. Initiates* themselves are very weak fighters.
- ***Magic armour*** Unlike it is conjecturable by the name, the *Magic armour* is no enhancement but a fighting unit. In the story of the game, it is a magically animated armour for hand-to-hand fighting. It does not deal a great round of damage, but its quite high hitpoints together with the cheap summoning costs make it quite a good infanterist.
- ***Summoner*** The *Summoner* is a weak fighting unit, only having slightly more hitpoints and dealing slightly more damage than the *Initiate.* His strength lies in the ability to conjure monsters like the *Daemon* or the *Dragon* or becoming a *Drake Rider.*

Enhancements

- ***Dragon Call*** is, together with *Advanced Summoning*, needed for *Summoners* to conjure *Dragons.*
- ***Advanced Summoning*** is, together with *Dragon Call*, needed for *Summoners* to conjure *Dragons.*
- ***Advanced Magic*** grants the ability for static fire attack to the *Archmage.*

- ***Mana Compression*** is an enhancement for *Battlemages*, *Summoners* and *Archmages*. It increases their energy, damage and attack range.

B.1.2 Tech faction

The tech faction can build 8 units, from which three do not have any fighting ability, 7 buildings and 8 enhancements.

Buildings

- ***Airodrome*** An *Airodrome* is needed to build *Ornithopters* and *Airships*.
- ***Barracks*** are training grounds for *Swordman* and *Archers*, the basic fighting units of the tech faction. They may be upgraded with a *Training Field* to produce at greater speed.
- ***Blacksmith*** A *Blacksmith* is needed for the *Barracks* to build *armoured Swordman*. Also the enhancements *Shield* Level 1 and 2 and *Sword* Level 1 and 2 can be researched by a *Blacksmith*.
- ***Castle*** The *Castle* is the central building of the tech faction. In *Castles*, resources are stored. *Castles* are needed to build *Workers*.
- ***Defense Tower*** A *Defense Tower* is capable of firing arrows at walking and flying units at great distance with slightly higher damage than an *Archer*.
- ***Farm*** A *Farm* provides food for the units as well as may produce *Cows* and *Pigs* which also provide food.
- ***Technodrome*** A *Technodrome* is needed to produce the *Technician*. In a *Technodrome*, *Advanced Architecture* and *Robotics* can be researched.

Units

- ***Airship*** The *Airship* is the strongest fighting unit of the tech faction. It is capable of flying and fires air to ground missiles which deal the second highest damage in the game.
- ***Air Ballista*** The *Air Ballista* is the tech faction's air raid defense, firing arrows at flying units even farther than the *Defense Tower*. Its high hitpoints and high speed make it a dangerous unit. The only drawback is, that flying units are build very late in the game.
- ***Archer*** The *Archer* is one of the two basic fighters of the tech faction. High speed, good reach, armour and the capability to hit ground and flying units as well make him a cheap and interesting fighting unit in the early time of the game. The quite low damage, however, reduces his use later in the game.

- ***armoured Swordman*** The *armoured Swordman* is the enhancement of the *Swordman* and can be produced if the player is in control of a *smith.* He is a relative cheap unit with mediocre hitpoints and damage and below average speed. The capability to be upgraded with *Shield* 1 and 2 and *Sword* 1 and 2 makes him an interesting unit for the mid of the game or hand to hand combat against any magic unit. If he manages to close up, he even proves as match for the *Archmage.*

- ***Battle Machine*** The *Battle Machine*, available with a *Technodrome*, *Robotics* and *Blacksmith*, is a strong fighting unit. With speed like the *armoured Swordman*, good armour and high hitpoints, he proves as hard to kill threat against enemy forces dealing high damage in hand-to-hand combat and also being attacking at range with medium damage to weaken approaching enemies. His only drawback may be the high production costs of 300 wood that exceed even the tech's building costs.

- ***Catapult*** As the *Battle Machine*, the *Catapult* needs *Robotics* researched and a *Blacksmith* and a *Technodrome* under control of the player in order to be build. It has quite high hitpoints, below average speed and fires missiles that deal good damage against ground based units.

- ***Cow*** The *Cow* is a unit for food production only. The player may move *Cows* around, but they are incapable of doing anything else.

- ***Horsemen*** can be produced if the player is in control of a *Blacksmith* and a *Farm* and has researched the *Stables* enhancement. He is the fastest unit of the tech faction.

Enhancements

- ***Advanced Architecture*** can be researched in a *Technodrome* and allows the construction of *Defense Towers* and *Airodromes.*

- ***Shield*** Level 1 and 2 can be built by a *Blacksmith* and increases the armour class of *armoured Swordmen.*

- ***Sword*** Level 1 and 2 can be built by a *Blacksmith* and increases the damage of *armoured Swordmen.*

- ***Training Field*** A *Training Field*, built by *Barracks*, increases the production speed of the *Barracks.*

- ***Robotics*** The *Robotics* enhancement, researched in the *Technodrome* is needed to build *Battle Machines* and *Catapults.*

- ***Stables*** may be built if *Advanced Architecture* is researched and allow training of *Horsemen.*

B.2 Unit statistics

B.2.1 Mage units

Hitpoints, armour and attack statistics for the units and buildings of the magic faction. If an attack is applicable only to ground units or only to air units it is noted with a subscript g (ground) or a (air). If nothing is stated, damage is dealt to walking and flying creatures as well.

Unit or building	Hitpoints	Armor		Attack			
				primary		secondary	
		Type	Class	Type	Damage	Type	Damage
Archmage	350	Organic	0	magic$_g$	190-370	magic	650-750
Archmage Tower	10 000	Stone	0	—	—	—	—
Battlemage	700	Leather	20	magic	100-180	—	—
Daemon	700	Organic	3	sword$_g$	60-140	—	—
Daemon Giant	2 500	Organic	50	sword$_g$	300-350	—	—
Dragon	2 500	Organic	20	magic	210-290	—	—
Drake Rider	1 300	Organic	50	magic$_g$	135-235	magic	120-220
Energy Source	1 400	Stone	0	—	—	—	—
Golem	2 000	Stone	45	magic$_g$	50-450	—	—
Initiate	450	Organic	0	magic$_g$	60-140	—	—
Library	5 000	Stone	0	—	—	—	—
Mage Tower	900	Stone	0	—	—	—	—
Magic Armor	900	Plate	2	magic$_g$	110-190	—	—
Summoner	500	Organic	0	magic	120-220	—	—
Summoner Guild	7 000	Stone	0	—	—	—	—
Tower of Souls	6 000	Stone	0	magic$_a$	250-350	—	—
Wicker Daemon	1 400	Stone	0	—	—	—	—

B.2.2 Tech units

Hitpoints, armour and attack statistics for the units and buildings of the tech faction. If an attack is applicable only to ground units or only to air units it is noted with a subscript g (ground) or a (air). If nothing is stated, damage is dealt to walking and flying creatures as well.

Unit or building	Hitpoints	Armor		Attack			
				primary		secondary	
		Type	Class	Type	Damage	Type	Damage
Aerodrome	6 000	Wood	0	—	—	—	—
Airship	1 700	Wood	20	magic$_g$	250-550	—	—
Air Ballista	3 000	Wood	20	arrow$_a$	225-375	—	—
Archer	700	Leather	20	arrow	50-150	—	—
Armored Swordsman	900	Plate	20	sword$_g$	80-140	—	—
Barracks	6 000	Stone	0	—	—	—	—
Battle Machine	2 200	Wood	45	sword$_g$	370-430	arrow	125-275
Blacksmith	6 000	Wood	0	—	—	—	—
Castle	9 000	Stone	0	—	—	—	—
Catapult	1 500	Wood	10	magic$_g$	300-600	—	—
Cow	500	Organic	0	—	—	—	—
Defense Tower	7 000	Wood	20	arrow	70-170	—	—
Farm	3 000	Wood	0	—	—	—	—
Horseman	1 300	Plate	50	sword$_g$	190-240	—	—
Ornithopter	1 200	Wood	15	arrow	100-200	—	—
Pig	300	Organic	0	—	—	—	—
Swordman	700	Leather	20	sword$_g$	60-120	—	—
Technician	550	Leather	0	magic$_g$	25-375	—	—
Technodrome	6 000	Wood	0	—	—	—	—
Worker	600	Leather	0	—	—	—	—

B.3 Unit Advancement

	Level 1		Level 2		Level 3	
Unit	Name	Kills	Name	Kills	Name	Kills
Archmage	Expert	5	Master	15	Legendary	30
Battlemage	Expert	3	Master	10	—	—
Daemon Giant	Ancient	10	—	—	—	—
Dragon	Ancient	10	—	—	—	—
Drake Rider	Expert	5	Master	10	—	—
Golem	Ancient	10	—	—	—	—
Air Ballista	Elite	10	—	—	—	—
Airship	Elite	12	—	—	—	—
Archer	Elite	5	—	—	—	—
Armored Swordsman	Elite	4	—	—	—	—
Battle Machine	Elite	10	—	—	—	—
Archer	Elite	5	—	—	—	—
Horseman	Elite	6	—	—	—	—
Ornithopter	Elite	10	—	—	—	—
Swordsman	Elite	3	—	—	—	—

B.4 Unit costs

In order to construct, train or summon buildings, units or creatures the player has to expand resources in the amount given in table B.4.1 for the magic and table B.4.2 for the tech faction.

B.4.1 Mage unit costs

Costs for the constructions of units and buildings of the magic faction in Glest. Units followed by an asterisk (*) are buildings (the Golem is considered as building for game purposes). Entries followed by a diamond (◇) are enhancements. Negative values indicate the production of a resource.

Unit or building	Resources			
	Gold	Wood	Stone	Energy
Archmage	200	—	—	2
Archmage Tower*	200	150	300	—
Battlemage	125	—	—	1
Daemon	50	—	—	1
Daemon Giant	250	—	—	2
Dragon	350	—	—	2
Drake Rider	125	100	—	2
Golem*	250	—	250	2
Initiate	75	—	—	1
Library*	100	150	250	—
Mage Tower*	175	150	400	—
Magic Armor	175	—	—	1
Mana Source*	150	—	100	−14
Summoner	175	—	—	1
Summoner Guild*	100	50	100	2
Tower of Souls*	200	—	250	3
Wicker Daemon◇	—	200	—	—

B.4.2 Tech unit costs

Costs for the constructions of units and buildings of the tech faction in Glest. Gold, wood and stone are costs that accrue only once for the production of the unit, while food is a permanently consumed resource. Negative values indicate the production of a resource. Units followed by an asterisk (*) are buildings.

Unit or building	Resources			
	Gold	Wood	Stone	Food
Airodrome*	150	200	—	—
Air Ballista	250	200	—	1
Airship	500	250	—	1
Archer	100	50	—	1
Armored Swordsman	150	75	—	2
Barracks*	100	150	50	—
Battle Machine	150	300	—	1
Blacksmith*	100	150	—	—
Castle*	200	150	350	—
Catapult	100	200	—	1
Cow	95	—	—	−10
Defense Tower*	250	250	—	—
Farm*	150	150	150	−10
Horseman	250	100	—	3
Ornithopter	275	200	—	1
Pig	50	—	—	−5
Swordman	75	25	—	1
Technician	150	—	—	1
Technodrome*	100	250	—	—
Worker	75	—	—	—

List of Figures

List of Tables

C References

"When you steal from one author, it's plagiarism; if you steal from many, it's research. " *Wilson Mizner, American playwright, (1876 - 1933)*

[BDE+08] Beume, Nicola ; Danielsiek, Holger ; Eichhorn, Christian ; Naujoks, Boris ; Preuss, Mike ; Stiller, Klaus ; Wessing, Simon: Measuring Flow as Concept for Detecting Game Fun in the Pac-Man Game. In: *Proceedings of the 2008 IEEE World Congress on Computational Intelligence*, 2008

[BHN+08] Beume, Nicola ; Hein, Tobias ; Naujoks, Boris ; Neugebauer, Georg ; Piatkowski, Nico ; Preuss, Mike ; Stüer, Raphael ; Thom, Andreas: To Model or Not to Model: Controlling PacMan Ghosts Without Incorporating Global Knowledge. In: *Proceedings of the 2008 IEEE World Congress on Computational Intelligence*, 2008

[Bre65] Bresenham, Jack E.: Algorithm for computer control of a digital plotter. In: *IBM Systems Journal* 4.1 (1965)

[Car78] Carrol, Lewis: *Through the Looking-Glass.* Neuauflage 2003. PENGUIN Group Inc : London, 1978

[Csi90] Csikszentmihályi, Mihály: *Flow: The Psychology of Optimal Experience.* HarperCollins : New York, 1990

[Dar59] Darwin, Charles R.: *On the Origin of species by means of natural selection : or the preservation of favored races in the struggle for life.* London, 1859

[Dav05] Davison, Andrew: *Killer Game Programming in Java.* O'Reilly Media, Inc., 2005. – ISBN 0596007302

[DEH+07] Danielsiek, Holger ; Eichhorn, Christian ; Hein, Tobias ; Kurtić, Edina ; Neugebauer, Georg ; Piatkowski, Nico ; Puchowezki, Michael ; Quadflieg, Jan ; Schnelker, Sebastian ; Stüer, Raphael ; Thom, Andreas ; Wessing, Simon: Zwischenbericht der PG 511 / Technische Universität of Dortmund. 2007. – Technical Report of the Collaborative Research Centre 531 Computational Intelligence (236/07). – in German

[Dij59] Dijkstra, Edsger W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik. 1*, 1959, S. 269–271

[DP85] Dechter, Rina ; Pearl, Judia: Generalized best-first search strategies and the optimality af $A^\star$. In: *Journal of the ACM* 32 (1985), S. 505–536

[ES07] Eiben, Agoston E. ; Smith, James E.: *Introduction to Evolutionary Computing.* Springer-Verlag : Berlin, 2007

[GMS04] Graham, Ross ; McCabe, Hugh ; Sheridan, Stephen: Neural Networks for Real-Time Path Finding In Computer Games. In: *Institute of Technology Blanchardstown Research Conference* (2004)

[GR03] Gallagher, Marcus ; Ryan, Amanda: Learning to play pac-man: An evolutionary Rule-based Approach. In: *Proceedings of the 2003 Congress on Evolutionary Computation (CEC'03)*, 2003, S. 2462–2469

[HNR68] Hart, Peter E. ; Nilsson, Nils J. ; Raphael, Bertram: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *Transactions on Systems Science and Cybernetics SSC4 (2)*, 1968, S. 100–107

[Kir04] Kirmse, Andrew: *Interdisciplinary Systems Research.* Bd. 26: *Game Programming Gems 4*. B&T : Basel, 2004

[Koh01] Kohonen, Teuvo: *Self-Organizing Maps.* Springer : Berlin, 2001

[Kra86] Krause, Eugene: *Taxicab Geometry.* Dover Publications : New York, 1986

[KS01] Kalyanpur, Aditya ; Simon, Mohan: *Pacman Using Genetic Algorithms and Neural Networks.* University of Maryland, 2001

[LSM91] Lin, Xia ; Soergel, Dagobert ; Marchionini, Gary: Information Retrieval. In: *Proceedings, 14th Ann. Int. ACM/SIGIR Conf. on R&D*, 1991, S. 262

[Mat02] *Kapitel* 3.1 Basic A* Pathfinding Made Simple. In: Matthews, J.: *AI Game Programming Wisdom.* Bd. 1. Charles River Media, 2002, S. 105–113

[MQLL07] Miles, Chris ; Quiroz, Juan ; Leigh, Ryan ; Louis, Sushil J.: Co-evolving influence map tree based strategy game players. In: *CIG*, IEEE, 2007

[Rey87] Reynolds, Craig W.: Flocks, herds and schools: A distributed behavioral model. In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques.* ACM : New York, NY, USA, 1987. – ISBN 0–89791–227–6, S. 25–34

[RMS92] Ritter, Helge ; Martinetz, Thomas ; Schulten, Klaus: *Neural Computation and Self-Organizing Maps: An Introduction.* Addison Wesley : München, 1992

[RN03] Russell, Stuart J. ; Norvig, Peter: *Artificial Intelligence : A Modern Approach.* Prentice Hall, 2003

[SZ06] Shen, Zhuoqian ; Zhou, Suiping: Behavior Representation and Simulation for Military Operations on Urbanized Terrain. In: *Simulation* 82 (2006), Nr. 9, S. 593–607. – ISSN 0037–5497

[Tan01] Tanenbaum, Andrew S.: *Modern Operating Systems.* Prentice Hall PTR : Upper Saddle River, NJ, USA, 2001. – ISBN 0130313580

[Toz02] Tozour, Paul ; Rabin, Steve (Hrsg.): *AI Game Programming Wisdom.* Bd. 1. Charles River Media, 2002. – 39–45 S.

[Val04] Valve: *Half-Life 2 - Raising the bar.* Prima Games, Division of Random House, Inc., 2004

[YH04] Yannakakis, Georgios N. ; Hallam, John: Evolving Opponents for Interesting Interactive Computer Games. In: *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04)* (2004), July

[YH05] YANNAKAKIS, Georgios N. ; HALLAM, John: A Generic Approach for Generating Interesting Interactive Pac-Man Opponents. In: *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games*, 2005, S. 94–101

D Weblinks

“Basic research is what I am doing when I don’t know what I am doing.”

Wernher von Braun, US (German-born) rocket engineer (1912 - 1977)

[1] Boost C++ Libraries, 2007. URL `http://www.boost.org`. July 9, 2007.

[2] Dungeons & dragons, 2008. URL `http://www.wizards.com/dnd/`. June 17, 2008.

[3] M. Babuskov. NJam, 2003. URL `http://njam.sourceforge.net/`. October 10, 2007.

[4] Martiño Figueroa, José González, Tucho Fernández, Félix Menéndez, and Marcos Caruncho. Glest, a free 3d real time strategy game, 2007. URL `http://www.glest.org`. December 16, 2007.

[5] GNU Software Foundation. Gnu general public license, version 2, June 2001. URL `http://www.gnu.org/licenses/gpl-2.0.html`. November 18, 2007.

[6] PG511. Ci in games. URL `http://www.ciingames.de`. January 30, 2008.

[7] The Apache Software Foundation. Xerces c++ parser. URL `http://xerces.apache.org/xerces-c/`. January 22,2008.

[8] Wikipedia. Ludo — wikipedia, the free encyclopedia, 2008. URL `http://en.wikipedia.org/wiki/Ludo_%28board_game%29`. June 17, 2008.

[9] Wikipedia. Pac-man — wikipedia, the free encyclopedia, 2007. URL `http://en.wikipedia.org/w/index.php?title=Pac-Man\&oldid=172015326`. November 18, 2007.

[10] Windward Studios. Enemy nations. URL `http://www.enemynations.com/`. February 11, 2008.