Bachelor thesis

SmarTee:
A neural network based navigational bot

December 23, 2011

Jascha Neutelings

Supervisor: Franc Grootjen
Radboud University Nijmegen
Department of Artificial Intelligence

JaschaNeutelings@student.ru.nl

Contents

Abstract

Most modern computer controlled players in computer games are controlled by static rule-based systems. Rule-based computer controlled players often do not act in a natural human way. To overcome this problem I propose using an artificial neural network, trained on human player input data, instead. This bachelor thesis assesses the performance of an artificial neural network controlled player and its generalization capabilities by performing real-time simulations and comparing the results to human data.

1 Introduction

Most current computer games use static rule-based systems for computer controlled players. It often happens that the programmer has overlooked a certain game situation while writing the AI code, causing the computer player to behave poorly under these circumstances. Human players, being able to relatively accurately derive the rules used by the computer player, can then use these flaws to their advantage, making the game less challenging. To make computer controlled players more challenging, they are often programmed to excel at abilities which are trivial for a computer to perform, but cannot be done as effective by humans because of physical restraints. These abilities include tasks requiring a high level of precision or near instantaneous reaction times. The problem with this approach is that the computer controlled players look very artificial and are not much fun to play against. They excel at tasks that are computationally trivial but physically impossible for humans to achieve and they fail at tasks that are computationally complex but trivial for humans to perform. A possible solution to this problem is to have computer players learn from humans. Because human behavior cannot easily be defined using a rule-based system, it might be more suitable to use an artificial neural network (ANN) which is more flexible and – arguably – more closely mimics the human way of thinking.

1.1 Artificial neural networks

An artificial neural network is a graph of interconnected nodes called neurons. A special kind of ANN is the feedforward network. Feedforward networks consist of multiple sequential layers of neurons whereby each neuron in a layer is connected to every neuron in the next layer. The links between these neurons are weighted, i.e. they have a numerical value attached to them, representing the strength of the link. Typically these ANNs have a special input layer, an output layer and any number of layers in between called hidden layers. The neurons within the input layer or input neurons represent the inputs or parameters of a function. The output neurons represent the function's result or output. In order to use a neural network to compute the result of a function, the parameters of the function are encoded as the activation levels of the input neurons. The activation is then propagated through the different layers of the network. This propagation is achieved by computing the weighted sum of all the neural activation levels in the previous layer using the weights from the links between the two neurons. The resulting weighted sum is then used as the input for the activation function which calculates the new activation level of the neuron. This is done for all neurons in the network. Often, a special kind of bias neuron is added to the network. This bias neuron is connected to all the non-input neurons. It does not receive activation from other neurons, but is usually fixed at an activation level of 1.0. This way it functions as a simple means to translate the input of the activation function as the bias weight is added to weighted sum. After the activation has been propagated to the output layer, the output activations are decoded to form the result of the function. However the true strength of a neural network is not so much in the ability to behave like a function, but the fact that it can easily be used to approximate

Figure 1: A feedforward network.

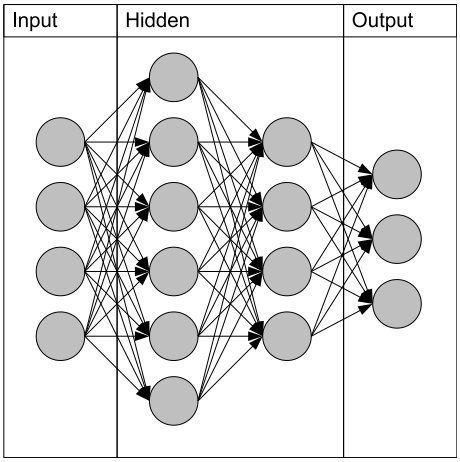
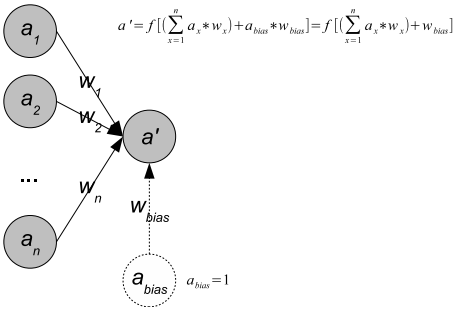


Figure 2: Neural propagation.



an unknown function, when the inputs and outputs are known. To make this possible, we have to use a supervised learning algorithm. The most commonly used algorithm for training the network is backpropagation learning[3, 13, 12]. It uses *gradient descent* to converge to a local minimum in the error landscape. It relies on the fact that the difference (delta) between the desired output and the obtained output of the network can be used to adjust the weights. This process of adjusting the weights starts at the links between the output nodes and the second to last layer of nodes and then propagates backwards towards the input layer. The links are updated according to the *delta rule*. The speed at which the weights converge depends on the *learning rate* parameter. It is often difficult to find the right learning rate for the problem at hand. To this end the resilient propagation (RPROP) algorithm was created by Martin Riedmiller and Heinrich Braun in 1992[10, 11]. It is similar to Manhattan update rule[11] in that the amount by which a weight is updated is not determined by the magnitute of the gradient and the learning rate. Both methods use

$$\Delta w_{ij} = -sign(\frac{\partial E}{\partial w_{ij}})\Delta_{ij}$$

to calculate the weight change. The Manhatten update rule uses a constant update value Δ_{ij} whereas the RPROP update value depends on whether a

local minimum was passed during the last iteration.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} = 0 \end{cases}$$

$$\text{where } \begin{cases} 0 < \eta^- < 1 < \eta^+ \\ t \text{ is the current iteration} \\ t-1 \text{ is the previous iteration} \end{cases}$$

If a minimum was passed ($sign(\frac{\partial E}{\partial w_{ij}})$ has changed between the last iteration and the current iteration) the new update value for the neuron is the update value of the previous iteration multiplied by η^-, where $0 < \eta^- < 1$. The update value is thus decreased. When the sign of the gradient remains the same, the update value is increased by multiplying the previous update value with η^+, where $\eta^+ > 1$. The RPROP algorithm is used for training the neural networks in this thesis.

1.2 Previous work

Some work has been done on using neural networks to control computer players in first person shooter games, most notably Quake II and Quake III. Experiments were done where neural network operated bots were taught to move through virtual worlds (maps) and aim for enemies [2]. Other research involved using neural networks in situational weapon and item selection [14]. The results from both articles show very positive results. Another paper[5] compared a number of classification/regression algorithms (ID3, Naive Bayes and ANN) on various tasks (selecting move direction, face direction, whether to jump and whether to accelerate). The results from this research show that the ANN outperforms all other classification algoritms as long as the training set is sufficiently large. It also shows that using the ensemble learning algorithm boosting with ANNs achieves even better results.

1.3 Research question

This bachelor thesis addresses the following questions regarding the viability of neural networks as a means of controlling a computer controlled player:

- Is it possible to train an artificial neural network to navigate a computer controlled player (bot) in a virtual game world?
- If so, how well does the artificial neural network generalize to situations outside of the training data?

In the next section I will discuss the setup of my research, including the software used and the experimental setup. The results section contains the

results of the experiments and related observations. These results and further research are discussed in the conclusion.

2 Methods

In this section I will describe the methods used. I will first describe the game I have used as the basis for my research. This is followed by a description of the concrete task that the bot has been trained on. The following subsections describe the tools I have created and used in order to create the bot and have it interact with the game. The remainder of this section discusses the configurations used for the networks driving the bot and the setup of the experiments.

2.1 Platform

The platform I have used for the experiments is Teeworlds[1], a game created by Magnus Auvinen. Teeworlds is a two-dimensional *run and gun platform* multiplayer computer game in which the player competes with other players in various game modes. The player controls a small ball-shaped character that can walk, jump and wield various weapons. These game modes include *capture the flag*, a *(team) death match* mode and an (unofficial) *racing* mode. Tees are able to walk, jump and fire their weapon, using the mouse cursor to aim. They can also use their grappling hook in order to reach areas that are unreachable by just jumping. There are a variety of weapons that can be found throughout the various maps (virtual worlds). These weapons are different in respect to rate of fire, projectile movement and damage dealt to other tees. Maps are built out of equally sized and fully aligned tiles (a grid). Each tile can either be solid, empty/air or a special tile which kills a Tee on contact. Tees can stand and walk on solid tiles and attach their grappling hooks to them, except for a special type of solid tile that doesn't allow hooking.

2.2 Task

I chose to limit my research to the capture the flag (*CTF*) game type. In CTF there are two teams, with both teams having their own base and flag. Initially the flags are at their respective bases. The goal is to go to the opponent base, take their flag and return it to the player's own base. Points are then given to the scoring team and the flags are returned to base. Scoring this way is only possible if the player's own flag is at their base. Just like most other game types in Teeworlds, players can use weapons to hinder their opponents. However, unlike (Team) Death Match, shooting opponents is not the main goal and very few points are awarded for "kills" compared to capturing a flag (1:100 ratio). CTF is an interesting game type because it allows for a lot a strategies. Players can go straight for the opponent's flag, ignoring any enemy players they encounter on the way. Or they could stay at their base and protect their own flag. When a player has the enemy flag, but the opponent has their flag as well, the player could try to hunt down the enemy flag carrier or he could go back to his own base and wait for a team member to get the flag back. The player could even wait in the enemy base and take the enemy flag carrier by

surprise. These are just a few strategies that players can use. When the teams get bigger, there are even more possibilities. However, for this research, the task of the bot is limited to taking the flag at the enemy base and bringing the flag to the allied base, without having to deal with opponents.

2.3 JTee

For the implementation of the bot the Java programming language was chosen. This choice was made because of Java being a very flexible and operating system independent platform. Teeworlds, however, is written in C and C++ and used to have a rather messy codebase[1]. To this end JTee was created. JTee is a framework I made to build a bridge between the Java environment and the native Teeworlds code. In order to connect the Java interface with the game code, the Java Native Interface (*JNI*) was used. JNI provides an Application Programming Interface (*API*) for native programming languages to manipulate Java objects, classes and primitives. The JTee API contains classes that correspond to a subset of the internal data structures and functionality. The structure of these classes is rearranged to better suit the object-oriented programming paradigm and to provide a more conceptual hierarchical structure than the actual internal data structures.

Plugin system

In order to access these APIs, JTee provides a plugin system. A plugin or *module* is a set of archived classes with one special main class as its entry point. This main class differs from the traditional main class in Java as it does not provide a static main method. Instead, the main class or *module class* has to be an instantiable class, either through a no-argument public constructor or a static factory method. It also has to implement the *Module* interface. This interface provides callback methods that are invoked when the module is loaded and certain events in the game occur. In order to create a loadable module, the module class and all auxiliary classes are put into a *JTee Module*(.jtm) file. A jtm file is a special kind of jar file in that its manifest file contains a few special entries that tell the module loader how to load the module and some additional information about the module. The module can be loaded using the game's command line interface or a Teeworlds command file.

The Module interface

The *Module* interface is used as the module's entry point. It also contains the most important callbacks.

```
public interface Module {
  void init();
  void destroy();
  void render(Graphics graphics, RenderPoint renderPoint);
  void update(World world);
  void stateChanged(State oldState, State newState);
```

[1]The version I used is 0.5.2; the current version at the time of writing is 0.6.1 and had its complete codebase refactored.

```
  void processInput(PlayerInput playerInput);
}
```

- When the module is loaded, the *init* method is called. This allows the module to register additional callbacks and do proper initialization.
- *destroy* is called when the module is unloaded. Can be used to perform clean-up code. All callbacks used by the module must be unregistered during this method call.
- The *stateChanged* method is called whenever the global state of the game changes. This global state is primarily used to determine whether a game is active.
- The *render* method is called multiple times when a frame is being drawn. Each invocation corresponds to a different point in the drawing phase. This allows drawing in between the game layers.
- The *update* method is called whenever the game world changes. This callback should be used to extract information about the world and to act upon the world.
- *processInput* is called whenever new input from the player is received.

The API

The *Game* class serves as the central access point for all API components.

```
public abstract class Game {
  public static Console getConsole();
  public static World getWorld();
  public static Controls getControls();
  public static State getState();
  public static View getView();

  public static boolean isRunning();
}
```

- The *Console* API provides access to all console/command line related functionality, including:
  - Registering console commands and callbacks.
  - Outputting text to console.
  - Capturing raw console input.
  - Executing console commands.
- The *World* API provides read-only access to the world state. The entire world object can be serialized. The world object is not available when there is no active game running. The world state consist of the following parts:
  - Player and character info for every player, e.g. the player's name, id, team and coordinates.

  - The map.
  - Power-up locations.
  - Flag locations.

- The *Controls* API is used to act upon the game. It contains methods for moving the Tee around.
- The *getState* method retrieves the current global state.
- Finally, the *View* API provides methods to retrieve the camera position and viewport width and height.

In addition to the main API, there are also a few utility classes that can be used. These classes include among others a basic implementation of a 2D vector and an API providing scripting access.

2.4 SmarTee

SmarTee is a collection of modules for use with the JTee framework I made. There are three modules:
smartee-dc is used for data collection.
smartee-cf is used for machine learning and simulation.
smartee-tt is used for running (simulated) time trials.

Data collection module

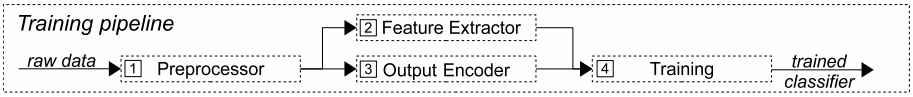
The data collection module can record raw game data obtained through the JTee framework to a file. The data is serialized using the Java Serialization API and its default protocol. The data contains the entire world state excluding the map (which in only included once and then back-referenced) for every frame recorded. It also includes a continuity flag for every frame to indicate whether the data is continuous at that point. Incontinuous data may result from switching servers, maps or from pausing the game.

Machine learning and simulation module

The machine learning module is used for training and testing a classifier and also for running simulations with a classifier. A *classifier* tries to predict the next action the bot should take, based on the a set of features extracted from a previous world state. This world state is from about 200ms before the next action. This is done to simulate the trainer's reaction time. The classifier takes an input vector of real numbers and also outputs a real vector. This vector is then decoded by an *output codec*. The output codec is also used to encode target vectors for use with training. The training pipeline is a modular pipeline in which every component can be replaced. The module only contains the interfaces used to build a pipeline; it contains no implementation of these interfaces. Following are the components that make up the training pipeline.

1. The *preprocessor*. Used to fix/filter out incorrect data, balance the types of inputs, etc. Its input and output are raw game data.

Figure 3: The training pipeline.



2. The *feature extractor*. Used to retrieve interesting data to be used as inputs for the classifier. It extracts a vector of real numbers from raw samples.

3. The *output codec/encoder*. Encodes raw player input into a real target vector.

4. The *training component*. Takes a classifier, a set of normalized features and encoded target outputs and trains the classifier on this data.

Time trial module

The time trial module is used to run a series of capture the flag simulations and record the time it takes for a bot or human to capture an enemy flag.

2.5 Classifier and pipeline implementation

The implementation of the classifier and the pipeline makes use of the *Encog* framework for Java.

Encog

Encog[6, 7] is an artificial intelligence framework that is primarily aimed at providing fairly complete neural networks functionality[2]. Encog provides various neural network models including feedforward networks, recurrent networks such as Elman[4] and Jordan[9] networks and self organizing maps. In order to train these models Encog has a number of training methods. The supervised learning methods of Encog include propagation strategies such as back propagation, resilient propagation and manhattan update rule propagation, but also simulated annealing and genetic algorithm learning. Encog allows the user to use multiple training methods at once. For example, you can use back propagation as the main training method and have Encog automatically switch to simulated annealing when the error improvement becomes too small. This can be used to overcome local minima. It also supports multi-threaded learning for some of the learning strategies. This can greatly increase the speed at which networks are trained if used on a multi-processor architecture, like a modern multi-core CPU. Recent versions even have experimental GPU (Graphics Processing Unit) support. This means that the computer's graphics card can be used to do part or all of the training. The GPU allows many more threads to run at once, however, the interaction between different threads has to be kept to a minimum. Therefore, in order to use the GPU for training a network a different kind of algorithm is required. Because of the difficulty of creating

[2]The new version of Encog (v3.0; not used in this research) supports a more generalized machine learning framework.

a parallel training algorithm with minimal thread interaction, I found that the current implementation did not actually improve training speeds on my machine when using the GPU.

The classifier

For the classifier I created a wrapper around the feedforward network and the Elman and Jordan recurrent networks from Encog.

The preprocessor

The preprocessor I used is pretty simple. The raw data sets often contain some bad data. The jump input state is not actually the moment at which the Tee jumps, but whether or not the player is pressing down the jump key. After a Tee has used his jump and double jump and has not yet landed on the floor, an additional jump key press does not actually do anything in the game. Also, when a player presses the jump key at a legitimate jumping moment, he will still probably hold the key down longer than just one frame. Therefore the preprocessor changes the jump input state to only be active when the character did actually jump and only for the first frame of the jump. The same problem occurs when the Tee is right next to a wall and the player presses the movement key into the direction of the wall. This input has no actual effect on the game and therefore the sample is simply removed. All samples in which there is no movement input at all are also removed. This is because if there is no change in the world at all, not even a change in the location of the Tee, some combinations of features and stateless networks may cause the Tee to just stand still forever. Finally, it can balance sample diversity by inserting copies of samples that are rare.

Feature extractors

Multiple feature extractors were tested, but about a handful were picked in the final implementation. Although the framework supports only one feature extractor, building a composite feature extractor that combines outputs from multiple feature extractors was a trivial task.

The first feature extractor uses a small grid around the Tee and determines for every cell in the grid whether the terrain in the cell is completely solid or partially/fully empty. Terrain is considered solid when it is impassible. It does not take into account other players. Solid cells are encoded as 1.0 and empty cells as 0.0.

The second feature extractor is a radial sensor that determines the distance from the Tee to the surrounding walls. It sends rays from the centre of the Tee into multiple directions. Rays are stopped by terrain and a certain maximum distance. The outputs are the distances for the various angles up to the maximum distance, scaled down to $[0.0, 1.0]$.

The third feature extractor extracts a vector representing the distance between the player and a flag.

The final feature extactor uses A* search to find an approximate path from the player to a flag. It uses the game's map grid as its search space. The states are the 32x32 pixel cells. The search algorithm does not prohibit physically

Figure 4: Visualisation of the grid.

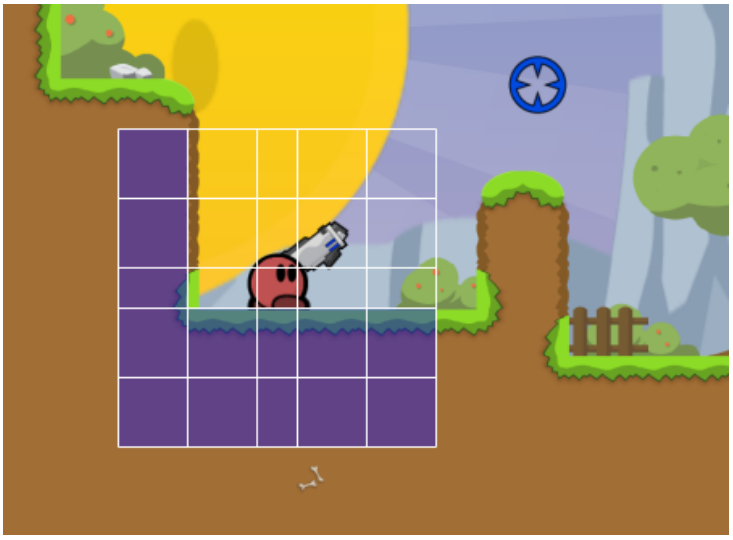
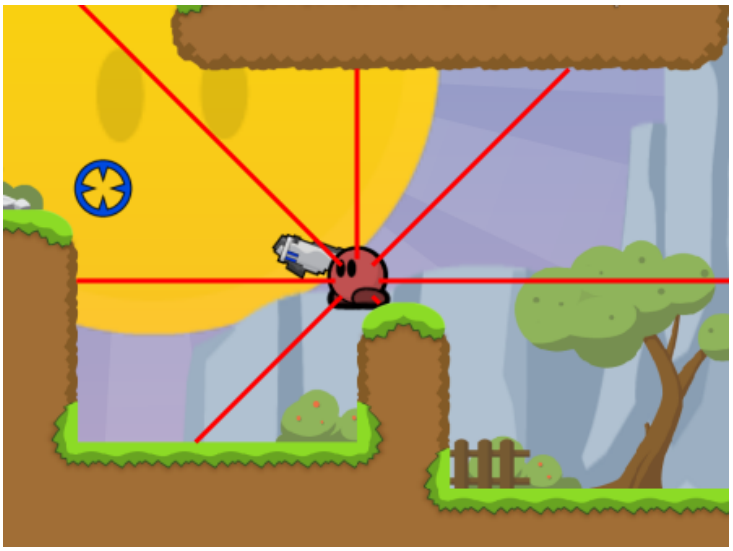


Figure 5: Visualisation of the radial sensor.



unreachable states, such as ledges unreachable by jumping; it simply interprets the map as two-dimensional grid, where you can move from one cell to a vertically or horizontally adjacent non-solid cell. A special cost function is used to determine a path that consists mostly of long straight lines.

Output codec

The output codec is responsible for encoding and decoding outputs from the network. The following information can be encoded and decoded by the codec:

- The direction in which the player wants the Tee to move. Encoded as

Figure 6: Path segment visualisation.

1.0 (left) and −1.0 (right). The case in which the player is not trying to move the Tee at all is undefined. However this case is filtered out by the preprocessor.

- The angle in which the player is looking. The angle is quantized and encoded as set of binary values (−1.0 and 1.0) of which only one value is 1.0.
- Whether the player pressed the jump key. It is encoded as a binary value: −1.0 for false and 1.0 for true.
- Whether the player is trying to shoot the gun. It is encoded as a binary value: −1.0 for false and 1.0 for true.
- Whether the player is trying to shoot the grappling hook. It is encoded as a binary value: −1.0 for false and 1.0 for true.

The output codec is also used to calculate the similarity between two player inputs. Each component of the five components described above are matched. For every component that matched exactly, one point is awarded. Consequently, the maximum number of points is five, the number of components.

Training method

Various training methods were tested, but the resilient propagation algorithm was ultimately chosen. The parameters are kept at their default values, an initial delta of 0.1 and a maximum delta of 50.0.

2.6 Experimental setup

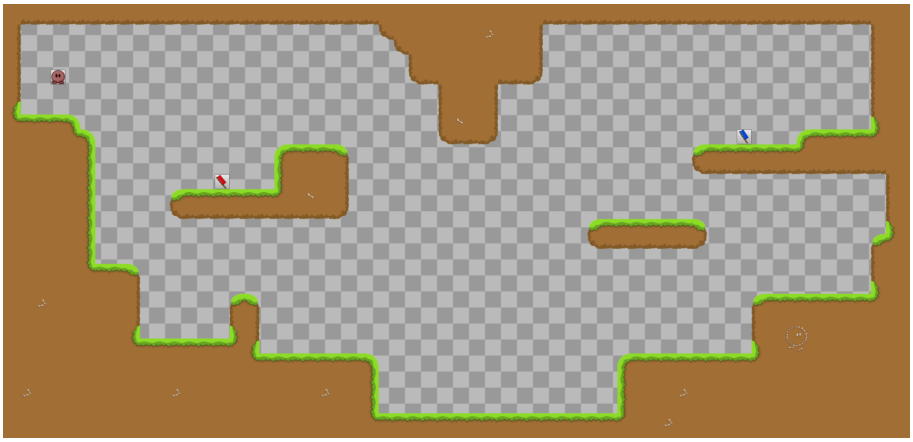
Three different network configurations were tested and compared. The only parameter I varied was the type of the network, i.e. Jordan, Elman or feedforward. The feature set was kept the same during the different experiments. It consisted of the following components:

- A grid extractor using a 5x5 detector grid with the outer cells being 48x48 and the middle cells being adjusted to the dimensions of the Tee (28 pixels in both dimensions).
- The radial sensor using 8 beams with maximum length of 1024 pixels and a precision of one pixel.
- Two flag distance extractors; one for each flag.
- Two flag path extractors; one for each flag, using only the first two straight lines of the path as an input vector for the network. The first component is always a horizontal line. If the path happens to start with a vertical line, only the vertical component is encoded and horizontal component is set to 0.0.

I used the preprocessor described earlier, but without making use of the sample balancing, because there was no significant improvement when using it. The output codec above is used, but is slightly simplified. The hook and shoot states are not used and neither is the angle (as it is only used for shooting and using the grappling hook). The similarity calculation is changed accordingly.

The raw training data set consists of 6789 samples recorded in one session. It contains data of me controlling the Tee to run up to the enemy flag, take it and return it to the allied base/flag. During the recording I did not shoot or use the grappling hook. Shooting is simply not part of the task of capturing the flag and hooking would greatly increase the difficulty of the task for the bot. Therefore a custom-made simplified map was used, on which the flags can be reached by just walking and jumping. The fully processed training data set contains 4587 samples.

Figure 7: The map.



The test data set is made up out of 10777 raw samples recorded during a different session than the training data. However, the same map is used and the same movement restrictions apply. The processed test data set contains 7347 samples.

The networks were trained with the resilient propagation algorithm as described earlier. The parameters were kept at default values and the networks were trained for 250 iterations on the training data set. The final error as well as the performance (as calculated by the output codec) were recorded. These values were also recorded for the test data set. Finally 25 time trials were run and the times (in milliseconds) to complete the full task of taking the enemy flag and returning it to base were recorded. Whenever a trial took longer than 30 seconds it was assumed that the Tee got stuck and the trial was considered a failure. The 25 time trials were also run by a me to get a human reference time.

3 Results

It turned out that the simulation performance was largely determined by the inital weights of the networks. Many of the trained networks failed every single simulation by getting the Tee stuck at a single point even though the training data set and test data set scores were decently high: $\geq 96\%$ for the training data set and $\geq 92\%$ for the test data set. This would mean I would have no simulation data to compare. In order to overcome this problem, all of the networks were trained from scratch 10 times and only the networks with the best simulation time means were picked.

Figure 8: Test results.

	Feedforward	Elman	Jordan	Human
$\text{Error}_{\text{train}}$	0.0317	0.1303	0.1380	-
$\text{Performance}_{\text{train}}$	0.9900	0.9654	0.9569	-
$\text{Error}_{\text{test}}$	0.2016	0.2502	0.2342	-
$\text{Performance}_{\text{test}}$	0.9444	0.9313	0.9302	-
N	25	25	25	25
$\text{Mean}_{\text{simtime}}$	11571	13656	13180	11944
$\text{Std.dev.}_{\text{simtime}}$	88.891	1954.1	1217.4	1120.2

All of the simulation runs for the selected configurations succeeded. The simulation time difference between the three network configurations is significant ($p < 0.05$). The feedforward network has the best simulation performance (11.6 seconds) followed by the Jordan network (13.2 seconds) quickly followed by the Elman network (13.7 seconds). The difference in simulation time between the human player and the feedforward network is not significant ($p > 0.05$).

3.1 Subjective analysis

From watching the real-time simulations I can conclude that all of the networks perform reasonably well, with the feedforward network almost perfectly exe-

cuting the shortest route every single time. The Elman and Jordan networks sometimes fail to make the Tee jump onto a ledge and the Tee will just fall down again. After a couple of jumps the recurrent networks succeed in getting the Tee to reach its goal.

In order to see if the networks are able to generalize I tried to interfere with the Tee as would happen in any real scenario. I used a player controlled Tee to harass the AI controlled Tee by dragging it into different areas. As long as these areas were part of the training data set, the Tee would get back on track and finish what it was doing. However, as soon as the Tee was dragged into areas it was not trained on, it was very likely that the Tee would get stuck. This is especially true for the feedforward network. The other networks are slightly more flexible in this situation as long their memory had not faded. I also tested the networks on maps they were not trained for. The first problem that occurred was the A* algorithm used in one of the feature extractors not being able to deal with medium to large sized maps; therefore I were limited to fairly small maps. I tested on one trivial map, consisting of a flat area with a flag on either side and a map that only slightly differed from the original training. The Tee failed to even reach the enemy flag on the trivial map and performed reasonably on the altered map. The difference between the networks was neglible. From the fact that the networks perform badly in untrained areas of the training map and on a different map, I can safely say that the networks do not generalize well.

4 Conclusion and discussion

From the results I can conclude that all network types do a fairly good job at performing the navigation task in a CTF game. The feedforward network turned out to be the best from these tests. However, the simple recurrent networks are strictly more powerful models and should be able to learn to perform at least as well as the feedforward network given the right training method and sufficient training time. I tried to use various different parameters for the resilient propagation algorithm and the number of training iterations to see if I could improve the performance of recurrent networks but to no avail. I can only assume that I did not pick the right parameter values or that resilient propagation in its default form might not be suited for training recurrent networks at all. Due to time limitations I did not try any alternative training algorithms for the tests.

The generalization capabilities of the neural networks are quite bad. The networks fail to generalize accross game maps and even fail to generalize to untrained areas within the same map. This is definitely not a wanted feature, but there is a reason to why this happens. In order for the network to succesfully execute its task, it has to perform near perfect. Whenever the network makes a wrong decision (generates a wrong output) along the route, it might move back into a previous state and enter a loop. This is especially true for the feedforward network as it lacks an internal state. In order to avoid these loops I had to overfit the network on the training data, which leads to bad generalization. The better solution would have been to pick a better (and thus smaller) feature set. However, selecting features for such a non-trivial task, without shifting all the work from the network to “smart” feature extractors

(effectively creating a rule based system), is a very time-consuming task.

We can now answer the two research questions stated at the beginning of this thesis:

- Is it possible to train an artificial neural network to navigate a computer controlled player in a virtual game world?
  - *Yes, it is possible to train an artifical neural network to navigate a computer controlled player (*tee*) sufficiently well in a virtual game world (*Teeworlds*).*
- If so, how well does the artificial neural network generalize to situations outside of the training data?
  - *The generalization capabilities of the artifical neural network are poor.*

Further research could be done to find out how well a bot can be trained on more realistic tasks. Adding other players, the use of weapons and the grappling to the task might yield more interesting information as this thesis only focused on the navigational aspects of this game's genre.

Other research could be done on how to train the two simple recurrent networks properly and see how well they perform when properly trained. Because the memory of Elman and Jordan recurrent networks fades rather quickly, it might be interesting to see if network types with a longer memory span, such as a LSTM network[8], would perform better. Alternatively, finding a way to prevent the networks from getting the bot stuck by using an external mechanism (outside of the network) might also be useful.

References

[1] M. Auvinen. Teeworlds. Website: `http://www.teeworlds.com`.

[2] C. Bauckhage, C. Thurau, and G. Sagerer. *Learning Human-like Opponent Behavior for Interactive Computer Games*, volume 2781 of *Lecturenotes in Computer Science*, pages 148–155. Springer Berlin / Heidelberg, 2003.

[3] A.E. Bryson and Y.C. Ho. *Applied optimal control*. American Institute of Aeronautics and Astronautics, 1979.

[4] J.L. Elman. Finding structure in time* 1. *Cognitive science*, 14(2):179–211, 1990.

[5] B. Geisler. An empirical study of machine learning algorithms applied to modeling player behavior in a Şfirst person shooterŤ video game. Master's thesis, University of Wisconsin - Madison, 2002.

[6] J. Heaton. Encog. Website: `http://www.heatonresearch.com/encog`.

[7] J. Heaton. *Programming neural networks with encog 2 in Java*. Heaton Research, Inc., 2010.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[9] M.I. Jordan. Serial order: A parallel distributed processing approach. *Advances in Psychology*, 121:471–495, 1997.

[10] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[11] M. Riedmiller and H. Braun. Rprop-description and implementation details. Technical report, Citeseer, 1994.

[12] D.E. Rumelhart, G.E. Hintont, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[13] P. Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Hardvard University, 1974.

[14] J. Westra. Evolutionary neural networks applied in first person shooters. Master's thesis, University Utrecht, 2007.

A JTee Javadoc

This section contains the javadoc documentation of the Java code for the JTee framework. The documentation was converted to LaTeX using a custom doclet I created specifically for this thesis. If you want the binaries or source for JTee, SmarTee or HiTeX doclet, you can send a request to `JaschaNeutelings@student.ru.nl`.

A.1 Package nl.ru.ai.jtee

Class AbstractModule

```
public abstract class AbstractModule
implements Module
```

Abstract base class providing an empty implementation of `Module`.

Plug-in modules can be derived from this class.

```
public AbstractModule()
```

```
public void destroy()
```

Description copied from interface: `Module`
Perform finalization code.

This method is called when the module is unloaded.

Specified by:

`destroy` in interface `Module`.

```
public void init()
```

Description copied from interface: `Module`
Perform initialization code of the module.

This method is called when the module is loaded.

Specified by:

`init` in interface `Module`.

```
public void processInput(PlayerInput playerInput)
```

Description copied from interface: `Module`
Process player input.

This method is called whenever new input is received from the player.

Specified by:

`processInput` in interface `Module`.

Parameters:

`playerInput` the input received from the player.

`public void render(Graphics graphics, RenderStage renderStage)`

Description copied from interface: `Module`
Perform drawing operations.

This method is called multiple times during the drawing phase of a frame at different rendering stages.

Specified by:

`render`	in interface `Module`.

Parameters:

`graphics`	the `Graphics` object used for drawing.
`renderStage`	the rendering stage for the current frame.

`public void stateChanged(State oldState, State newState)`

Description copied from interface: `Module`
Receive notification when the global state changes.

This method is called every time the global state changes.

Specified by:

`stateChanged`	in interface `Module`.

Parameters:

`oldState`	the previous global state.
`newState`	the current global state.

`public void update(World world)`

Description copied from interface: `Module`
Receive notification when the world updates.

This method may be used to update the state of the Module and to act upon the World.

Specified by:

`update`	in interface `Module`.

Parameters:

`world`	the `World` object containing all the information about the current state of the game world.

Interface Character

`public interface Character`

Class representing a Tee.

`public double getAngle()`

Return the angle the Tee is facing in radians.

Returns:

the angle in radians.

`public int getHeight()`

Return the height of the Tee in pixels.

Returns:

the height of the Tee.

`public Vector2D getPosition()`

Return the position of the Tee in the world.

Returns:

the position of the Tee.

`public PlayerState getState()`

Return the state of the player.

Returns:

the state of the player.

`public Vector2D getVelocity()`

Return the velocity at which the Tee is moving.

Returns:

the velocity vector.

`public Weapon getWeapon()`

Return the type of the weapon the Tee is currently holding.

Returns:

the weapon type.

`public int getWidth()`

Return the width of the Tee in pixels.

Returns:

the width of the Tee.

Interface CommandCallback

`public interface CommandCallback`

Interface used for processing console commands.

`public void execute(String command, String[] params, KeyStroke stroke)`

Receive notification when the command associated with this `CommandCallback` object is invoked.

Parameters:

`command`	the command entered.
`params`	the parameters following the command.
`stroke`	the keystroke (`null` for non-stroked commands).

Class CommandInUseException

`public class CommandInUseException`
`extends RuntimeException`

Thrown to indicate that the command to be registered is already in use.

`public CommandInUseException(String msg)`

Interface Console

`public interface Console`

Interface representing the in-game console.

An object implementing this interface can be accessed through `Game.getConsole()` when the game is running.

`public void executeCommand(String commandLine)`

Execute a console command given the command line.

Equivalent to calling `executeCommand(commandLine, null)`.

Parameters:

`commandLine`	the command line to execute.

`public void executeCommand(String commandLine, KeyStroke stroke)`

Execute a console command given the command line and the key stroke.

Parameters:

`commandLine`	the command line to execute.
`stroke`	the keystroke; can be `null`.

`public void executeCommand(String command, Object... params)`

Execute a console command given the name of the command and a number of parameters.

The parameters are converted to strings using `Object.toString()`.

Equivalent to calling `executeCommand(String)` with a concatenation of the command name and the string representations separated by spaces.

Parameters:

`command`	the name of the command.
`params`	zero or more parameters.

`public PrintWriter getDebugWriter()`

Return a `PrintWriter` that sends the output to the debug loggers.

Returns:

a debug `PrintWriter`

`public PrintWriter getWriter()`

Return a `PrintWriter` that sends the output to the console.

Returns:

a console `PrintWriter`

`public void logDebugMessage(Object msg)`

Log a debug message.

Equivalent to calling `logDebugMessage(null, msg)`.

Parameters:

`msg` the message.

`public void logDebugMessage(String subsystem, Object msg)`

Log a debug message.

Message will be send to internally registered debug loggers.

The `msg` argument is converted to a string using `Object.toString()`.

Parameters:

`subsystem` the subsystem where the message originated.

`msg` the message.

`public void printMessage(Object msg)`

Print a message to the console.

The `msg` argument is converted to a string using `Object.toString()`.

Parameters:

`msg` the message.

`public void printStackTrace(Throwable e)`

Sends the stack trace to the debug loggers.

Equivalent to calling `e.printStackTrace(this. getDebugWriter())`.

Parameters:

`e` the exception object.

`public void registerCommand(String command, String params, CommandCallback callback, String help) throws CommandInUseException`

Register a console command with the console command dispatcher.

The `params` parameter takes a signature describing how many and what types of parameters this command accepts. The signature is a string describing what type of parameter is allowed at each position. The types are:

- 's' - a string
- 'i' - an integer
- 'f' - a float
- 'r' - the rest of the line as a string

Each of these characters can be suffixed with a question mark ('?') to indicate that the parameter is optional. For example, the string "sf?r" represents a command taking a string, an optional float and the rest of the line as parameters.

Parameters:

`command`	the name of the command
`params`	the parameter signature
`callback`	the object whose `execute` method will be invoked.
`help`	a string describing how to use the command or `null`.

Throws:

`CommandInUseException`	if a command by that name is already in use.

```
public void registerInputInterceptor(InputInterceptor
interceptor, String statusText) throws
InputInterceptorActiveException
```

Register a callback object to intercept any input sent to the console.

Only one can be active at a time.

Parameters:

`interceptor`	the callback object implementing `InputInterceptor`.
`statusText`	the text to be displayed in the autocomplete label.

Throws:

`InputInterceptorActiveException`	if an input interceptor is already active.

```
public void unregisterInputInterceptor()
```

Unregister the current console input interceptor.

Interface Controls

```
public interface Controls
```

Interface providing access to the in-game controls.

An object implementing this interface can be accessed through `Game.getControls()` when the game is running.

`public void fire()`

Make the Tee fire its weapon.

`public void jump()`

Make the Tee jump.

`public void moveLeft()`

Make the Tee walk to the left.

`public void moveRight()`

Make the Tee walk to the right.

`public void setTargetPosition(Vector2D position)`

Change the position of the crosshair.

Parameters:

`position`	the new position of the crosshair in game coordinates.

`public void stopMoving()`

Stop the movement of the Tee.

Enum Direction

`public final enum Direction`
`extends Enum<Direction>`

Enumeration of the possible walking directions of a Tee.

LEFT To the left.

NONE No direction/standing still.
RIGHT To the right.

`public Direction opposite()`

Return the direction opposite to the direction of this object.

The opposite direction of `NONE` is `NONE`.

Returns:
the opposite direction

`public static Direction valueOf(String name)`

`public static Direction[] values()`

Interface Flag

`public interface Flag`

Class representing a flag in CTF games.

`public Player getCarrier()`

Return the player that is carrying this flag.

Returns:
the flag's carrier or `null` if none.

`public Vector2D getPosition()`

Return the flag's position.

Returns:
the position.

`public Team getTeam()`

Return the team to which the flag belongs.

Returns:
the flag's team.

```
public boolean isAtBase()
```

Return whether the flag is at its base.

Returns:

`true` if flag is at base; `false` otherwise.

Class Game

```
public abstract class Game
```

The main API entry-point.

Provides access to all subsystems using static getters.

```
public Game()
```

```
public static Console getConsole()
```

Return the console subsystem.

Returns:

the console object.

```
public static Controls getControls()
```

Return the controls subsystem.

Only available if `getState()` == `State.ONLINE`.

Returns:

the controls object or `null`.

```
public static Module getModule(String name)
```

Return a loaded `Module` object by name.

Parameters:

`name`	the name of the module.

Returns:

the module or `null` if no module by that name is loaded.

`public static String getResourcePath(String fileName)`

Return the absolute path for the given resource file.

Return `null` if the file could not be found.

Parameters:

`fileName`	the relative file name.

Returns:

the absolute path to `fileName` or `null`.

`public static State getState()`

Return the current global state.

Returns:

the global state.

`public static String getStoragePath(String fileName)`

Return a absolute path for storing the given file.

Parameters:

`fileName`	the relative file name.

Returns:

the absolute path to `fileName`.

`public static View getView()`

Return the view subsystem.

Only available if `getState()` == `State.ONLINE`.

Returns:

the view object or `null`.

`public static World getWorld()`

Return the world subsystem.

Only available if `getState()` == `State.ONLINE`.

Returns:

the world object or `null`.

`public static boolean isRunning()`

Return whether the game is running.

Returns:

`true` if the game is running; `false` otherwise.

Interface Graphics

`public interface Graphics`

Interface providing access to the in-game graphic through OpenGL (JOGL).

`public GL drawBegin()`

Prepare the graphics subsystem for drawing and return a non-debug GL object.

Equivalent to calling `drawBegin(false)`.

Returns:

the GL object.

`public GL drawBegin(boolean debug)`

Prepare the graphics subsystem for drawing and return a GL object.

If the `debug` argument is `true`, a debug GL object will be returned that will throw exceptions when something goes wrong. The non-debug GL object does not throw exceptions and the error state has to be checked manually.

Parameters:

`debug` whether to use a debug or non-debug version of the GL object.

Returns:

the GL object.

`public void drawEnd()`

Close access to the graphics subsystem.

This method has to be called after `drawBegin` before `drawBegin` can be called again.

`public void setProjection(Projection projection)`

Change the orthographic projection.

Equivalent to calling `setProjection(projection, -1.0d, 1.0d)`.

Parameters:

`projection`	the projection constant.

`public void setProjection(Projection projection, double near, double far)`

Change the orthographic projection.

Parameters:

`projection`	the projection constant.
`near`	the distance to the nearer depth clipping plane
`far`	the distance to the farther depth clipping plane

Interface InputInterceptor

`public interface InputInterceptor`

Interface used for intercepting console input.

`public void inputReceived(String input)`

Receive notification when new console input is received.

Parameters:

`input`	a line of input.

Class InputInterceptorActiveException

`public class InputInterceptorActiveException extends Exception`

Thrown to indicate that another `InputInterceptor` is already active.

`public InputInterceptorActiveException(String arg0)`

Enum KeyStroke

```
public final enum KeyStroke
extends Enum<KeyStroke>
```

Enumeration representing the keystroke states.

PRESS The key is down.
RELEASE The key is up.

```
public static KeyStroke valueOf(String name)
```

```
public static KeyStroke[] values()
```

Interface LocalCharacter

```
public interface LocalCharacter
extends Character
```

Interface representing the local Tee.

Provides more information about the Tee than its superinterface `Character`.

```
public int getAmmo()
```

Return the current ammo for the Tee's active weapon.

Returns:
the current ammo.

```
public int getArmor()
```

Return the Tee's current armor.

Returns:
the Tee's armor.

```
public int getHealth()
```

Return the Tee's current health.

Returns:
the Tee's health.

Interface LocalPlayer

```
public interface LocalPlayer
extends Player
```

Interface representing the local player.

```
public LocalCharacter getCharacter()
```

Description copied from interface: `Player`
Return the Tee associated with this player.

If the player is spectating, dead or off-screen, this method will return `null`.

Overrides:

`getCharacter` in interface `Player`.

Returns:

the player's Tee or `null` if not available.

Interface Map

```
public interface Map
```

Interface representing the game-layer of a map.

```
public int getHeight()
```

Return the height of the entire map.

Returns:

the map height.

```
public int getHorizontalIndex(double x)
```

Return the horizontal index given an x-coordinate.

Parameters:

`x` the x-coordinate

Returns:

the horizontal index.

```
public String getName()
```

Return the name of this map.

Returns:

the name of the map.

`public Tile getTileAtIndex(int x, int y)`

Return the tile at the specified index.

Parameters:

`x` the x-index.
`y` the y-index.

Returns:

a tile.

`public Tile getTileAtPosition(double x, double y)`

Return the tile at the specified position.

Parameters:

`x` the x-coordinate.
`y` the y-coordinate.

Returns:

a tile.

`public Tile getTileAtPosition(Vector2D p)`

Return the tile at the specified position.

Equivalent to calling `getTilePosition(p.getX(), p.getY())`.

Parameters:

`p` the position vector.

Returns:

a tile.

`public int getTileHeight()`

Return the height of the tiles on this map.

Returns:

the tile height.

`public Tile[][] getTiles()`

Return a two-dimensional array of the tiles in this map.

Returns:

a two-dimensional array of tiles

`public int getTileWidth()`

Return the width of the tiles on this map.

Returns:

the tile width.

`public int getVerticalIndex(double y)`

Return the horizontal index given a y-coordinate.

Parameters:

`y` the y-coordinate.

Returns:

the vertical index.

`public int getWidth()`

Return the width of the entire map.

Returns:

the map width.

Interface Module

`public interface Module`

The entry point and main callbacks for a JTee Module.

Plug-in creators should implement this interface and put it in a jtm file together with any auxiliary classes. A jtm file is a jar file with the .jtm extension and an extended manifest file. The manifest is required to include the `Module-Name` attribute. This attribute the denotes the name of the module as used by the module manager. Other required attributes are either the `Module-Class` attribute, which denotes the name of the class implementing the Module interface, or the `Module-Factory`, which indicates the name of a factory class followed by the name of the factory method, separated by whitespace.

`public void destroy()`

Perform finalization code.

This method is called when the module is unloaded.

`public void init()`

Perform initialization code of the module.

This method is called when the module is loaded.

`public void processInput(PlayerInput playerInput)`

Process player input.

This method is called whenever new input is received from the player.

Parameters:

`playerInput`	the input received from the player.

`public void render(Graphics graphics, RenderStage renderStage)`

Perform drawing operations.

This method is called multiple times during the drawing phase of a frame at different rendering stages.

Parameters:

`graphics`	the `Graphics` object used for drawing.
`renderStage`	the rendering stage for the current frame.

`public void stateChanged(State oldState, State newState)`

Receive notification when the global state changes.

This method is called every time the global state changes.

Parameters:

`oldState`	the previous global state.
`newState`	the current global state.

`public void update(World world)`

Receive notification when the world updates.

This method may be used to update the state of the Module and to act upon the World.

Parameters:

`world`	the `World` object containing all the information about the current state of the game world.

Interface Player

`public interface Player`

Class representing a player.

`public Character getCharacter()`

Return the Tee associated with this player.

If the player is spectating, dead or off-screen, this method will return `null`.

Returns:

the player's Tee or `null` if not available.

`public int getId()`

Return the player identifier.

Returns:

player id.

`public String getName()`

Return the name of the player.

Returns:

the name.

`public int getScore()`

Return the player's current score.

Returns:

the score.

`public Team getTeam()`

Return the team this player belongs to.

Return `null` if the current game type does not support teams.

Returns:

the player's team or `null`.

`public boolean isLocal()`

Return whether this player is local.

Returns:

`true` if this player is the local player; `false` otherwise.

Interface PlayerInput

`public interface PlayerInput`

Interface representing raw control input received from the player.

`public Direction getDirection()`

Return the direction in which the Tee will walk.

Returns:

the walking direction.

`public Vector2D getRelativeTargetPosition()`

Return the position of the crosshair relative to center of the Tee in game coordinates.

Returns:

the relative crosshair position.

`public boolean isHooking()`

Return whether the Tee will attempt to shoot its grappling hook.

Returns:

the jump flag.

```
public boolean isJumping()
```

Return whether the Tee will attempt to jump.

Returns:
the jump flag.

```
public boolean isShooting()
```

Return whether the Tee will attempt to shoot.

Returns:
the shoot flag.

Enum PlayerState

```
public final enum PlayerState
extends Enum<PlayerState>
```

Enumeration listing the values of the player state.

CHATTING The player is chatting.
IN_MENU The player is in the menu.
PLAYING The player is playing.
UNKNOWN An unknown state.

```
public static PlayerState valueOf(String name)
```

```
public static PlayerState[] values()
```

Interface PowerUp

```
public interface PowerUp
```

Interface representing an in-game power-up.

```
public Vector2D getPosition()
```

Return the position of the power-up.

Returns:
the position.

```
public PowerUpType getType()
```

Return the type of the power-up.

Returns:
the power-up type.

Enum PowerUpType

```
public final enum PowerUpType
extends Enum<PowerUpType>
```

Enumeration representing the different power-up types.

ARMOR An armor pick-up (a shield).
HEALTH A health pick-up (a heart).
WEAPON A weapon pick-up.

```
public static PowerUpType valueOf(String name)

public static PowerUpType[] values()
```

Enum Projection

```
public final enum Projection
extends Enum<Projection>
```

Enumeration representing built-in orthographic projections.

GAME_LAYER Orthographic projection onto the game-layer.
SCREEN Standard screen projection.

```
public static Projection valueOf(String name)

public static Projection[] values()
```

Enum RenderStage

```
public final enum RenderStage
extends Enum<RenderStage>
```

Enumeration representing the various render stages during rendering.

AFTER_BACKGROUND After the background layer has been drawn.
AFTER_FOREGROUND After the foreground layer has been drawn.
AFTER_HUD After the HUD has been drawn.
AFTER_PLAYERS After the Tees have been drawn.
FIRST Before anything else has been drawn.
LAST After all other drawing operations.

```
public static RenderStage valueOf(String name)

public static RenderStage[] values()
```

Enum State

```
public final enum State
extends Enum<State>
```

Enumeration representing the global state.

CONNECTING The client is connecting to a server.
DEMOPLAYBACK The client is playing a demo.
LOADING The client is loading data.
OFFLINE The client is running, but not connected to a server.
ONLINE The client is connected to a server.
QUITING The client is shutting down.

```
public static State valueOf(String name)

public static State[] values()
```

Enum Team

```
public final enum Team
extends Enum<Team>
```

Enumeration representing teams.

BLUE_TEAM The blue team.
RED_TEAM The red team.
SPECTATOR Spectator mode.

```
public Team getOpposingTeam()
```

Return the team opposing this team.

Return `null` if the current team is `SPECTATOR`.

Returns:

the opposing team or `null`.

```
public static Team valueOf(String name)
```

```
public static Team[] values()
```

Enum Tile

```
public final enum Tile
extends Enum<Tile>
```

Enumeration representing a game-layer tile.

AIR A basic non-solid tile.
DEATH A non-solid tile that causes death on impact.
NOHOOK A solid tile that cannot be shot with the grappling hook.
SOLID A basic solid tile.

```
public boolean isSolid()
```

Return whether the current tile is solid.

Returns:

`true` if the tile is solid; `false` otherwise.

```
public static Tile valueOf(String name)
```

```
public static Tile[] values()
```

Interface Vector2D

```
public interface Vector2D
```

Interface representing a two-dimensional vector.

```
public Vector2D add(Vector2D vec)
```

Adds the given vector to this vector.

Parameters:

`vec` the vector to add.

Returns:

the result of the addition.

`public Vector2D divide(double scalar)`

Divides the vector by a scalar value.

Parameters:

`scalar` the value to divide by.

Returns:

the result of the division.

`public double getEuclideanDistance(Vector2D vec)`

Return the Euclidean distance between this vector and the given vector.

Parameters:

`vec` the other vector.

Returns:

the Euclidean distance between the vectors.

`public double getManhattenDistance(Vector2D vec)`

Return the Manhatten (city block) distance between this vector and the given vector.

Parameters:

`vec` the other vector.

Returns:

the Manhatten distance between the vectors.

`public double getX()`

Return the x-component of the vector.

Returns:

the x-component.

`public double getY()`

Return the y-component of the vector.

Returns:

the y-component.

`public Vector2D multiply(double scalar)`

Multiplies the vector with a scalar value.

Parameters:

`scalar` the value to multiply by.

Returns:

result of the multiplication.

`public Vector2D subtract(Vector2D vec)`

Subtracts the given vector from this vector.

Parameters:

`vec` the vector to subtract

Returns:

the result of the subtraction.

Interface View

`public interface View`

Interface providing access to the game's viewport. An object implementing this interface can be accessed through `Game.getView()` when the game is running.

`public Vector2D getCenter()`

Return the point the camera is looking at.

Returns:

the center of the camera in game coordinates.

`public int getHeight()`

Return the height of game's viewport.

Returns:

the height of the viewport.

`public int getWidth()`

Return the width of game's viewport.

Returns:
the width of the viewport.

Enum Weapon

`public final enum Weapon`
`extends Enum<Weapon>`

Enumeration representing the different weapon types.

GRENADE The grenade launcher.
GUN The basic pistol.
HAMMER The hammer.
NINJA The ninja outfit and katana.
RIFLE The laser rifle.
SHOTGUN The shotgun.

`public static Weapon valueOf(String name)`

`public static Weapon[] values()`

Interface WeaponPowerUp

`public interface WeaponPowerUp`
`extends PowerUp`

Interface representing a weapon power-up.

A `PowerUp` can be cast to a `WeaponPowerUp` if the power-up type is `PowerUpType.WEAPON`.

`public Weapon getWeaponType()`

Return the type of the weapon.

Returns:
the weapon type.

Interface World

`public interface World`

Class representing the game world.

Maximum number of players.

`public Flag getFlag(Team team)`

Return the flag for the specified team.

Return `null` if the game type does not support flags.

Parameters:

`team` the team.

Returns:

the team's flag or `null`.

`public Flag[] getFlags()`

Return the flags in a CTF game.

Return `null` if the game type does not support flags.

Returns:

an array of flags or `null`.

`public String getGameType()`

Return a string describing the game type.

Returns:

the game type.

`public LocalPlayer getLocalPlayer()`

Return the local player.

Returns:

the `LocalPlayer` object.

`public Map getMap()`

Return a representation of the map.

Returns:
the map.

`public Player getPlayer(int id)`

Return a player given its identifier.

Return `null` if a player with the given id is not found.

Parameters:
`id` the player's identifier.
Returns:
player matching the id or `null`.

`public Player[] getPlayers()`

Return an array of all players on the server.

Returns:
an array of `Player` objects.

`public PowerUp[] getPowerUps()`

Return all the power-ups in the current game world.

Returns:
an array of power-ups.

`public PowerUp[] getPowerUps(Set<PowerUpType> types)`

Return all power-ups that match any given power-up types.

Parameters:
`types` a set of power-up types.
Returns:
an array of matching power-ups.

`public WeaponPowerUp[] getWeaponPowerUps(Set<Weapon> types)`

Return all weapon power-ups that match any given weapon types.

Parameters:

`types` a set of weapon types.

Returns:

an array of matching weapon power-ups.

`public boolean` **`isGameSuspended`**`()`

Return whether the current game is suspended.

Suspension usually happens in between rounds.

Returns:

`true` if the game is suspended; `false` othwerwise.

A.2 Package nl.ru.ai.jtee.util

Class AbstractCharacter

```
public abstract class AbstractCharacter
implements Character
```

Abstract base class partially implementing `Character`.

```
public AbstractCharacter()
```

```
public int getHeight()
```

Description copied from interface: `Character`
Return the height of the Tee in pixels.

Specified by:

`getHeight`	in interface `Character`.

Returns:

the height of the Tee.

```
public int getWidth()
```

Description copied from interface: `Character`
Return the width of the Tee in pixels.

Specified by:

`getWidth`	in interface `Character`.

Returns:

the width of the Tee.

Class AbstractMap

```
public abstract class AbstractMap
implements Map
```

Abstract base class partially implementing `Map`.

`public AbstractMap()`

`public int getHeight()`

Description copied from interface: `Map`
Return the height of the entire map.

Specified by:

`getHeight`	in interface `Map`.

Returns:
the map height.

`public int getHorizontalIndex(double x)`

Description copied from interface: `Map`
Return the horizontal index given an x-coordinate.

Specified by:

`getHorizontalIndex`	in interface `Map`.

Parameters:

`x`	the x-coordinate

Returns:
the horizontal index.

`public Tile getTileAtIndex(int x, int y)`

Description copied from interface: `Map`
Return the tile at the specified index.

Specified by:

`getTileAtIndex`	in interface `Map`.

Parameters:

`x`	the x-index.
`y`	the y-index.

Returns:
a tile.

`public Tile getTileAtPosition(double x, double y)`

Description copied from interface: `Map`
Return the tile at the specified position.

Specified by:

`getTileAtPosition` in interface `Map`.

Parameters:

`x` the x-coordinate.

`y` the y-coordinate.

Returns:

a tile.

`public Tile getTileAtPosition(Vector2D p)`

Description copied from interface: `Map`
Return the tile at the specified position.

Equivalent to calling `getTilePosition(p.getX(), p.getY())`.

Specified by:

`getTileAtPosition` in interface `Map`.

Parameters:

`p` the position vector.

Returns:

a tile.

`public int getTileHeight()`

Description copied from interface: `Map`
Return the height of the tiles on this map.

Specified by:

`getTileHeight` in interface `Map`.

Returns:

the tile height.

`public int getTileWidth()`

Description copied from interface: `Map`
Return the width of the tiles on this map.

Specified by:

`getTileWidth` in interface `Map`.

Returns:

the tile width.

`public int getVerticalIndex(double y)`

Description copied from interface: `Map`
Return the horizontal index given a y-coordinate.

Specified by:

`getVerticalIndex`	in interface `Map`.

Parameters:

`y`	the y-coordinate.

Returns:

the vertical index.

`public int getWidth()`

Description copied from interface: `Map`
Return the width of the entire map.

Specified by:

`getWidth`	in interface `Map`.

Returns:

the map width.

Class AbstractWorld

`public abstract class AbstractWorld`
`implements World`

Abstract base class partially implementing `World`.

`public AbstractWorld()`

`public Flag getFlag(Team team)`

Description copied from interface: `World`
Return the flag for the specified team.

Return `null` if the game type does not support flags.

Specified by:

`getFlag`	in interface `World`.

Parameters:

`team`	the team.

Returns:

the team's flag or `null`.

`public LocalPlayer getLocalPlayer()`

Description copied from interface: `World`
Return the local player.

Specified by:

`getLocalPlayer`	in interface `World`.

Returns:

the `LocalPlayer` object.

`public Player getPlayer(int id)`

Description copied from interface: `World`
Return a player given its identifier.

Return `null` if a player with the given id is not found.

Specified by:

`getPlayer`	in interface `World`.

Parameters:

`id`	the player's identifier.

Returns:

player matching the id or `null`.

`public PowerUp[] getPowerUps(Set<PowerUpType> types)`

Description copied from interface: `World`
Return all power-up that match any given power-up types.

Specified by:

`getPowerUps`	in interface `World`.

Parameters:

`types`	a set of power-up types.

Returns:

an array of matching power-ups.

`public WeaponPowerUp[] getWeaponPowerUps(Set<Weapon> types)`

Description copied from interface: `World`
Return all weapon power-ups that match any given weapon types.

Specified by:

`getWeaponPowerUps`	in interface `World`.

Parameters:

`types`	a set of weapon types.

Returns:

an array of matching weapon power-ups.

Class BasicVector2D

```
public class BasicVector2D
implements Vector2D, Serializable
```

Basic immutable implementation of `Vector2D`

```
public BasicVector2D()
```

```
public BasicVector2D(double x, double y)
```

```
public Vector2D add(Vector2D vec)
```

Description copied from interface: `Vector2D`
Adds the given vector to this vector.

Specified by:

`add`	in interface `Vector2D`.

Parameters:

`vec`	the vector to add.

Returns:

the result of the addition.

```
public Vector2D divide(double scalar)
```

Description copied from interface: `Vector2D`
Divides the vector by a scalar value.

Specified by:

`divide`	in interface `Vector2D`.

Parameters:

`scalar`	the value to divide by.

Returns:

the result of the division.

`public boolean equals(Object obj)`

`public double getEuclideanDistance(Vector2D vec)`

Description copied from interface: `Vector2D`
Return the Euclidean distance between this vector and the given vector.

Specified by:

`getEuclideanDistance`	in interface `Vector2D`.

Parameters:

`vec`	the other vector.

Returns:

the Euclidean distance between the vectors.

`public double getManhattenDistance(Vector2D vec)`

Description copied from interface: `Vector2D`
Return the Manhatten (city block) distance between this vector and the given vector.

Specified by:

`getManhattenDistance`	in interface `Vector2D`.

Parameters:

`vec`	the other vector.

Returns:

the Manhatten distance between the vectors.

`public double getX()`

Description copied from interface: `Vector2D`
Return the x-component of the vector.

Specified by:

`getX`	in interface `Vector2D`.

Returns:

the x-component.

`public double getY()`

Description copied from interface: `Vector2D`
Return the y-component of the vector.

Specified by:

`getY`	in interface `Vector2D`.

Returns:

the y-component.

`public int hashCode()`

`public Vector2D multiply(double scalar)`

Description copied from interface: `Vector2D`
Multiplies the vector with a scalar value.

Specified by:

`multiply`	in interface `Vector2D`.

Parameters:

`scalar`	the value to multiply by.

Returns:

result of the multiplication.

`public Vector2D subtract(Vector2D vec)`

Description copied from interface: `Vector2D`
Subtracts the given vector from this vector.

Specified by:

`subtract`	in interface `Vector2D`.

Parameters:

`vec`	the vector to subtract

Returns:

the result of the subtraction.

`public String toString()`

Class Environment

`public final class Environment`

Utility class providing scripting access.

```
public void clear()

public static Environment createEnvironment(Module module,
String name, ClassLoader loader)

public static Environment createEnvironment(String name,
ClassLoader loader)

public static Environment createTemporaryEnvironment(ClassLoader
loader)

public T evaluate(File scriptFile) throws ScriptException,
FileNotFoundException

public T evaluate(String scriptText) throws ScriptException

public static Environment getEnvironment(Module module)

public static Environment getEnvironment(String name)

public static Environment getGlobalEnvironment()

public static Map<String, Object> getGlobalVariables()

public String getLanguage()

public String getName()

public Object getVariable(String name)

public Map<String, Object> getVariables()

public boolean isGlobal()

public void setLanguage(String language) throws
UnsupportedScriptingLanguageException

public void setVariable(String name, Object value)
```

Class UnsupportedScriptingLanguageException

```
public class UnsupportedScriptingLanguageException
extends Exception
```

Thrown to indicate a scripting language is not found on this system.

```
public UnsupportedScriptingLanguageException(String message)
```