

# Chuck Norris rocks!

Daniel Cuadrado, Yago Saez *Member, IEEE*

**Abstract**—In this introductory work we present our first approach to a computer controller player for first-person shooter videogames, where we have applied genetic algorithms in order to evolve the best dodge rules. This paper is a report of the results obtained by our bot during the first competition held in Trondheim, Norway, during the IEEE Congress on Evolutionary Computation (CEC 2009).

## I. INTRODUCTION

The world of videogames is unstoppable. Nobody can deny or even overlook the fact that the videogames is making more profits by far than any other entertainment industry. Games have reached an amazing level of visual realism and perfection. We believe that we are living in a moment where big changes are coming up. Improvements made in artificial intelligence (AI) will constitute the major difference between videogames. The *proof* of this is that, so far, nobody has successfully passed the Turing test. Change is not far away: a large number of developers and researchers are focusing their efforts on looking for more complex and real artificial intelligent behaviors and companies are spending money and time on this, much as they did with graphics in the past.

Within videogame AI we can find different approaches, from trying to create behaviors as similar as possible to human behavior to just dealing with programming issues, which are tough to code or conceptualize by humans. This paper will deal with the latter concern.

The remainder of this paper is organized as follows: a review of some AI techniques applied to videogames is described in Section 2. Section 3 introduces the reader to the competition, and the videogame and interface used are explained. In Section 4 we explain the design of the bot we developed. Section 5 describes specifically the AI developed for this contest. Section 6 shows both results and scores obtained during training and competition, and finally, Section 7 summarizes the main conclusions of this work.

## II. RELATED WORK

Interest in using computer games as a test bed for advanced computational intelligence has significantly increased in the scientific community. A wide variety of contributions can be found whose common point is that they focus on First Person Shooter (FPS) games, such as Quake, Unreal Tournament (UT) or Half-Life.

Bots (computer-controlled players) are usually developed as rule-based expert systems, and many of them are customized with values which have to be assigned by the developers. Therefore, the effectiveness of the bot depends

on the skills or knowledge of the developers. In order to avoid this type of situation, some authors have presented a method for tuning these parameters using genetic algorithms (GA), [1].

Not only the values but the rules themselves can also be a matter of primary research. For example, [6] offers a nice approach using different machine learning techniques such as Naïve Bayes classifiers and neural networks (NN), among others. In other works, authors have applied evolutionary strategies to obtain the best behavior tree, where each leaf represents an atomic—or non-divisible—behavior; see [4].

These techniques are easy to implement; however, they have problems in coping with complex behavior. In [7], the authors propose learning directly from human examples by recording game plays. These games are analyzed and divided into a set of simple sub-behaviors. They are then used to train a machine learning system in which each state is a neural network. GAs are then used to evolve the NN connection weights. Another study is also focused on behavioral cloning is [10]. In this case, the authors place a combination of self-organized maps and a multilayer perceptron over the data obtained from human players. In [9], the same authors describe an interesting work where neural GAs are used to represent the structure of the environment in order to improve action-related behavior. The study conducted by Priesterjahn *et al.* [8] relies on evolutionary strategies to achieve the same goal.

Team behavior is a more challenging object of research. An interesting approach can be found in [2], where well-coordinated team behavior is achieved by using computational fields created by bots. The idea is that all the bots can perceive other bots' fields and move in accordance with them, e.g., following the fields' gradient. Other work along the same lines of collaborative research uses evolutionary algorithms to evolve team strategies for the “capture the flag” game [3].

### A. Dodge

Among all the different states a bot can take, the dodge state plays, from our point of view, a key role. Firstly, it is common to most states, so it does not matter what you are doing at any stage; you can always end up dodging an incoming projectile. Secondly, even if you had, for example, the best values for some rule-based expert system, if your dodge system is not good enough, your bot will be engaged frequently. Nevertheless, dodging is not always possible, depending on the speed of the bullet or projectile. Thus, if the weapon is, for example, a mini-gun, then there is no way to avoid the attack. But if the enemy is shooting a rocket launcher, the bot will have time to react by processing much

of the information about the situation, such as speed, direction, source, damage radius and so forth; then it has a chance to avoid the projectile.

There is research that tries to deal with this situation using neural networks evolved by the NEAT algorithm; see [4]. A small corridor is created for training, with a start point at one side, and a goal point at the other side. The bot that shoots at the target is situated near the goal point, without moving. The trained bot starts from a predefined point and has to reach the end without being damaged. Two experiments were performed: one with incomplete information about the incoming missile, and a second one providing complete knowledge of the threat. The results showed that NEAT can be successfully used to optimize dodge rules. Finally, another, similar work but using a different scenario, (Quake III instead of UT), is that conducted by Bonacina *et al.*, [5]. The conclusions showed that NEAT needs just a few generations to evolve a network that is able to survive for a long time.

### III. COMPETITION SETUP

#### A. Competition

The competition called “*Unreal Tournament 2004 Deathmatch*” was held for the first time in Trondheim, Norway, during the 2009 IEEE congress on evolutionary computation. This competition was inspired by the 2K BotPrize [11], a Turing test contest. The objective of this competition is to create a winning computer-controlled bot using any evolutionary computation (EC) technique. To do this, several deathmatch games are run where all the programs created by the participants and the bots developed by the organizers fight. The winners are the bots capable of killing as many of the opposing players as possible, while limiting the number of times they are killed.

#### B. Unreal Tournament 2004 (UT04)

This game is classified as a first-person shooter (FPS), where users interact with the 3D environment through the eyes of a virtual character (the “first person”) and fight against other human or artificial bots using the weapons given (“shooter”). Besides the different weapons supported, other elements, such as shields, health packs and the like, are spread throughout the maps to help players to achieve their goals.

Deathmatch is a game mode where players compete in a free-for-all match: in other words, there are no teams. At the end of the match, the one with the highest score wins. The score increases one point every time a player makes one frag, killing an opponent, as well as in case of suicide or self killing. Falling down into lava or dying from your own grenade will count as a negative point.

A modified version of the game is used for the competition. This modification, named GameBots [12], is a project started by the University of Southern California's Information Sciences Institute and provides a socket-based interface for connecting to UT04. This means that we have a protocol for controlling bots and for retrieving game state information with text commands.

To make things easier, an interface has been developed over the Gamebots layer: Pogamut [13].

#### C. Pogamut

An IDE developed as a Netbeans plug-in, Pogamut makes the development of the bots' logic easy for the users. Before Pogamut came along, strong videogame developing skills were needed before anyone could apply any AI techniques to them. Now, thanks to the combination between GameBots and Pogamut and due to the high level of bot control, a great chance has been given to all researchers. With this interface, researchers can concentrate all their efforts on what they really want without worrying about difficult issues. This interface makes everything accessible and in a very simple and intuitive way. With the Pogamut class library, it is easy to find out where weapons or other items are and easy to retrieve information from the environment or find out the bot's state. It provides a method for interacting either with the game map or bots. One can even use ray-casting to detect gaps, wall collisions and other structural features of the environment.

Nevertheless, Pogamut is not fully finished at version 2.3, which was used for this study. It suffers from several bugs and inefficiencies which force the user to dedicate time to figure out how to overcome these problems.

#### D. Setup

The battles that took place at the competition were held between our bot, called Chuck Norris; another participant, named Agent Smith; and a default bot, called Hunter. The organizer asked each participant to select one map, plus another one proposed by the chairman. Then, all the bots were run for five minutes within each map. They received their scores at the end of the match. The winner was the one who achieved the highest total scores. The final results can be shown at section VI.

### IV. DEVELOPED BOT

#### A. Objectives

The main goal was to develop the best possible bot in order to beat any other participant. To this end, we improved a basic bot using different EC techniques. Some examples could be

- Evolutionary strategies to find out which rules are suitable to change the states of the agent,
- Genetic algorithms for optimizing parameters behind each action,
- Swarm optimization for team movement behavior,
- Genetic programming to evolve some parts of the code.

Although we considered using these techniques, only genetic algorithms were used for evolving dodging rules. The scarce time available for this competition joined with the problems suffered with Pogamut were the main reasons to only implement one of them. However, the aim is to carry on this research line adding and comparing different EC developments.

## B. Hand-coded skills

Following the model of finite state machines, we developed a bot based on four basic states: search, run away, pursuit and attack. We coded all of them ourselves, and so we have defined when and under which conditions each state has to be performed.

- 1) *Search*: every time a bot detects low levels of any important feature, such as life, shield levels, number of weapons or ammunition, we command the bot to look for what it lacks. The way UT04 deals with path finding is by using nodes or “navigation points”. These points are spread throughout the level, creating a 3D connected graph. An A\* search algorithm is used by the game engine to give the best path to the point specified. As bots have complete knowledge of the map, they are able to know which item is closer, and they can ask the game engine for the optimal path to follow. Every time an item is caught, it will appear again after a few seconds (UT04 uses the term “respawn”). Each item, depending on its relevance, has a different period. Our bot is able to keep a list of what has it taken and when the items are going to be available again. In the case of important items, such as that which gives you extra power, the bot can return to take the regenerated object.
- 2) *Run away*: one of the keys to reach victory is to always keep a superior state in comparison with the enemy; this means more health, more ammo and so on. It is impossible for us to know any enemy state but we can see the weapon it is holding. We use this information to decide whether it is worthwhile to fight or not. Some weapons are better than others, but as long as our bot has obtained one good enough, it will fight; otherwise we activate the “run away” state. This state looks for the weapon which is closer to the bot and further from our enemy.
- 3) *Attack*: this is a very important state to perform. Killing your enemies efficiently will help you to achieve an easy victory. We mainly divide this state into two types of attacks, again depending on the enemy’s weapon. Each weapon has an effective distance within which the damage it inflicts is greater, as well as a maximum range. If the opponent is further away than the maximum range, the shot will be pointless. Thus, there are basically two sorts of weapons: *melee*, which are deadly at short distances; and *range*, which are useful for longer distances. If our enemy is holding a *melee* weapon then our bot will step back until it reaches a safe distance while shooting with a *range* weapon. However, if the enemy is holding a *range* weapon, then the bot will approach him as closely as possible, shooting with a *range* gun and then with a *melee* weapon when close enough. A very common situation is running out of ammo and/or getting seriously damaged in the middle of the battle. This is the time for determining what will be more effective. In these instances, the bot will change to a

state where it is looking for items while facing the enemy and shooting him. The main objective in these moments is to get what is needed as quickly as possible without giving our back to the enemy, as this would be a fatal error.

Once a weapon is selected, it is very important to shoot accurately. Pogamut offers an easy way to do this: only one line command is enough to shoot at your enemy. Nevertheless, precision depends on the level assigned to your bot, which means that if the bot is set to the maximum level, it will be an expert shooter; on the other hand, if its level is low, it will miss frequently. In order to avoid this dependency, as this level could be modified at anytime during the competition, we developed our own shooting system. As with many other FPS games, the engine supplies the enemy coordinates, so it is easy to think that the only action that you have to do is to shoot at that position. However, it is not that simple: if the target is moving, you will miss, as the bullet takes time to reach its intended target. Our shooting system is able to predict where a mobile target is going to move. This is done according to its speed and direction at the moment of engagement.

- 4) *Pursuit*: a simple but necessary state which is activated each time the bot loses visual contact with the target. The action for this state is going to the last place where the enemy was seen.

## V. DODGE CONTROLLER DESIGN

As we said before, dodging is a delicate moment in the game; it can define the difference between winning or losing a battle. Each time an enemy shoots a missile our bot gets an asynchronous event as a warning alarm. The first thing we do is to check whether the missile is really a threat to us by seeing if its trajectory intersects with our position. We also measure its shock wave, which can also take some life. This is done by measuring the distance between the bot position and floor/wall collision location; if it is bigger than the damage radius, then we are outside the shock wave. If one of these two conditions is true, then we act in consequence. Basically, what we have to decide is where and how long we want to jump (this constitutes the output), keeping in mind the actual environment around the bot (received as input).

### A. Input Data

In order to obtain environmental information the bot uses ray-casting, which acts like laser sensors, detecting collisions with any other element in the game. As can be seen in Figure 1, we have used three rays, one 90° to the left, another one 90° to the right, and the last one pointing at the bot’s back. Hence, we can check whether there are obstacles which could affect our dodge action. Three sensors mean 2<sup>3</sup> different conditions to take into account (see Table I).

TABLE I  
SENSORS STATES

Condition	Left Sensor	Right Sensor	Back Sensor
1	Off	Off	Off
2	Off	Off	On
3	Off	On	Off
4	Off	On	On
5	On	Off	Off
6	On	Off	On
7	On	On	Off
8	On	On	On

All the different conditions we can receive as input and their corresponding combination of sensors.

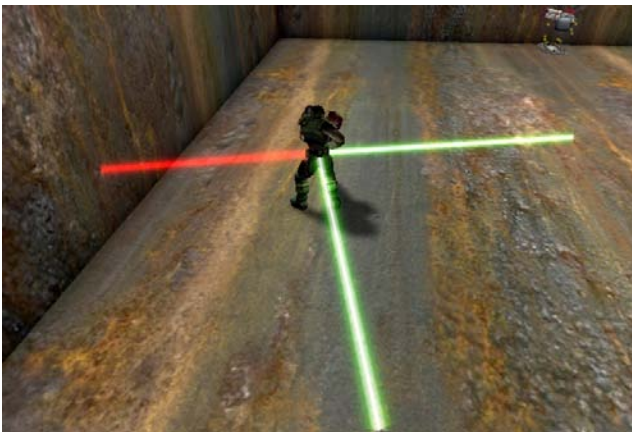


Fig. 1. An example of in-game ray-casting. The *Left Sensor* is on, while the *Right* and *Back Sensor* are off.

### B. Output

To implement the dodge command, we use a specific command given by Pogamut; we call it by passing three parameters:  $A$ ,  $B$ , and  $C$ .

1) The first two values,  $A$  and  $B$ , are the coordinates of the jump direction vector in reference to the missile angle.  $A$  belongs to the  $X$ -axis and  $B$  to the  $Y$ -axis. We have discretized the values; see Figure 2.

TABLE II MEANING OF $A$ AND $B$ VARIABLES			
$A$	$B$	Degree	Code
1	0	0°	000
1	1	45°	001
0	1	90°	010
-1	1	135°	011
-1	0	180°	100
-1	-1	225°	101
0	-1	270°	110
1	-1	315°	111

Fig. 2. Different directions, in degrees, that a bot can take to dodge a missile and the values that the parameters  $A$  and  $B$  have to take to make the dodge. Also shown is their bit encoding for the individual.

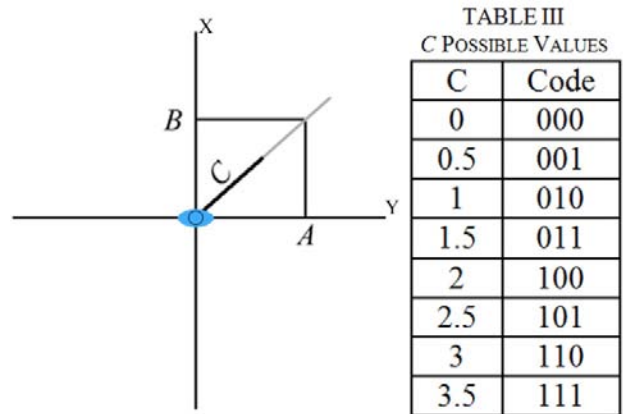


Fig. 3. We work with parameter  $C$  as a distance.

2)  $C$  works as the *jump distance*, with a value of zero for a big jump; as it increases, the jump becomes smaller. We work with eight possible discretized values, as shown in Figure 3.

### C. Rules

Each rules set has 48 bits, of which the first 6 bits belong to the first condition (Table I), the next 6 bits to the second, and so on. Each group being made of 6 bits, 3 are used for the direction (Table II) and the other 3 for the distance (Table III). Figure 4 describes an example of a rules set population.

	Condition 1		Condition 2		Condition 8														
	Direction	Length	Direction	Length	Direction	Length													
Rules set 1	0	1	1	1	0	0	1	0	1	0	1	0	...	0	1	1	1	1	1
Rules set 2	0	0	1	1	0	1	0	1	1	1	0	0	...	0	1	0	1	0	1
Rules set 100	0	1	0	0	1	1	0	0	1	0	0	0	...	1	1	0	1	1	1

Fig. 4. An example of a population with 100 individuals as rules sets.

### D. Evolutionary Rule System

First of all, we considered whether to develop a brute force algorithm or a heuristic search for this task. We decided to use GA's because with this model we had a search space of  $2.81E+14$  possibilities. Then, we created a special map to evolve the population of the rules set. The map is shaped like a box and is small enough to make sure most of the conditions from Table I are met at least once. We set up a stationary bot (the coach) at one side of the map, and the evolving bot (the student) at the opposite side. We programmed the coach to do nothing but shoot at the learning bot, and the learning bot to just dodge any incoming projectile. The dodge movement will follow the appropriate rules set. It will follow each rules set for five shots, watching how long it survives while following its instructions and evaluating them according to total time alive. We assume that the higher the fitness, the better the rules are for dodging.

After testing and evaluating all the rules sets, the system proceeds with the already known genetic steps: selection, crossover and mutation, and start with the next generation.

The main problem found at this stage is the noise that is deliberately introduced by the game engine, and which leads to different results given the same situation. This noise leads to situations where you might remain alive for five seconds, and on the next try, with the same rules set, you live for 10 seconds. See Figure 5.

## VI. RESULTS

### A. Training

It can take days to achieve just a few iterations of this stage, and as the rules get better the bot spends more time on the battle field, and so the iteration gets longer. Despite the noise mentioned above, we can observe a tendency to ascend. The best rules set allowed our bot to survive around 2 minutes. In order to get these results, we set the GA population to 100, the mutation percentage to 10%, elitism to 4 and the roulette wheel selection to a single point crossover.

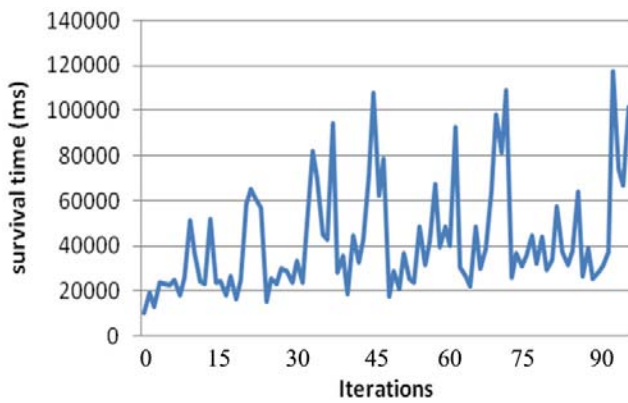


Fig. 5. Evolution of fitness

### B. Tests

The bot developed for the competition has been tested against the default bot used at the competition and against itself, without AI, so we can observe the improvement gained as the dodge rules evolve.

The experiments were run 20 times in three different maps for 5 minutes each. The results show how our bot dominates the game without any problem, reducing the number of times it is killed and, hence, increasing the number of frags. This is due to the fact that when a bot is killed it has to start over from the beginning, which means starting with weak weapons, whereas if it survives a battle, it keeps its advantage. The final scores can be seen in Figure 6. They are assigned in the same way as in the competition, subtracting the number of deaths from the number of frags. The best scores were achieved in the three maps where dodge rules are used.

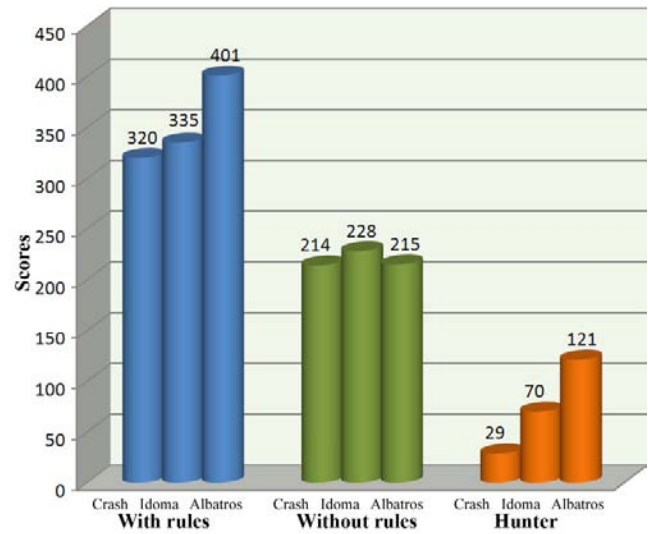


Fig. 6. Scores obtained by each bot (evolved rules, not evolved, and Hunter) in the three maps tested: Crash, Idoma and Albatros.

Using evolved dodge rules obtains a 56% higher improvement than when rules are not used. The improvement obtained with and without rules in each map can be observed in Figure 7 where, for example, playing in the Crash Map, a bot with rules survives 32.60% longer than does a bot without rules, and thanks to this, it makes 21.54% more kills. In general, three details can be observed in Figure 7: first, the improvement with evolved rules; second, that the larger the percentage of deaths avoided, the larger the percentage of frags achieved, which shows the relevance of dodging projectiles. Finally, the Crash Map is similar to the map where the bot was trained, while Albatros is a very different outdoor map; thus, the more similar the map is to the training map, the better are the results obtained. This third conclusion leads us to confirm that in following versions our bot should be trained in different scenarios.

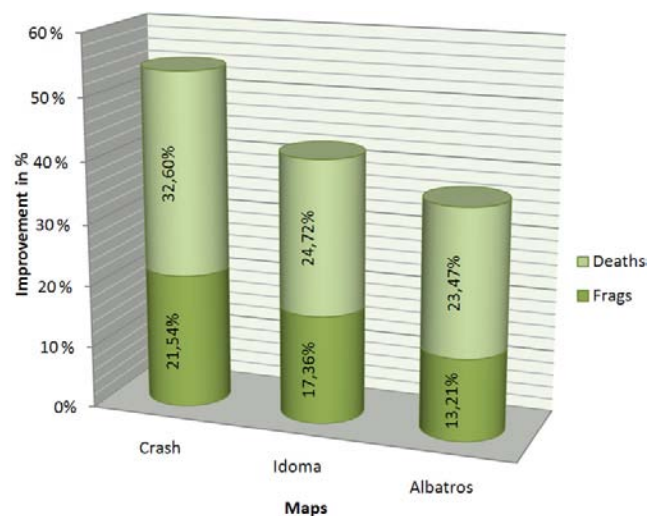


Fig. 7. Deaths and Frags obtained in different maps (evolved rules vs. not evolved)

### C. IEEE Competition

The competition results were good as long as the map did not have lava or any other surface which, makes the bot's life ebb away with contact. Our bot was not programmed to deal with these types of situations, so that was definitely our major weakness. Another problem was that our bot was trained to dodge against only one bot (one-on-one), and the contest was played with three different bots at the same time. Although victory went to another participant, we asked the organizer of the competition for a second test. Our aim was to confirm the problems described above. The last battles run after the competition by the chairman showed us that in maps without lava and in one-on-one combat, our bot was at least as competitive as the winner (Map: IDOMA, results: 4-3 for the opponent and 7-6 for Chuck Norris).

## VII. CONCLUSIONS

Despite the fact that we still have a long road ahead of us, we have succeeded in our first goal, which was to develop a competitive evolutionary computation bot for the IEEE CEC'09 competition. With this introductory work we have shown how EC techniques can be used successfully in videogames. In this work, we have used GAs to evolve adequate dodge rules. These evolved rules command the movement of our bot when a projectile is coming toward it. The difference between bots with evolved dodge rules and bots without is remarkable; see Figure 8. Finally, we have also survived Pogamut with an intense love-hate relationship and a report of problems that will certainly be solved in future versions.

As for future work, our next aim is to improve the dodge controller in order to make it able to consider gaps and holes (not only walls and steps). Thus, with the new controller the bot should be able to behave properly under any circumstances. Other research lines which remain open include using evolutionary strategies to find out which rules are the suitable for changing the states of the agent or to use genetic programming to evolve some parts of the code dynamically.

## ACKNOWLEDGMENTS

This work was supported in part by the Spanish MCyT project MSTAR, Ref: TIN2008-06491-C04-03.

## REFERENCES

- [1] N. Cole, S. J. Louis, and C. Miles, "Using a Genetic Algorithm to Tune First-Person Shooter Bot". University of Nevada, Congress on Evolutionary Computation, CEC2004, Vol. 1, pp 139-145, June 2004.
- [2] M. Mamei, and F. Zambonelli, "Motion Coordination in the Quake 3 Arena Environment: A Field-Based Approach," Lecture Notes in Computer Science, "Environments for Multi-Agents Systems", pp. 264-278, 2005.
- [3] S. Bakkes, P. Spronck and E. Postma, "TEAM: The Team-Oriented Evolutionary Adaptability Mechanism," Lecture Notes in Computer Science, "Entertainment Computing – ICEC 2004", pp. 273-282, August 2004.
- [4] R. Kadlec, "Evolution of intelligent agent behavior in computer games", Master Thesis, Dept. of Software Engineering, Charles University in Prague, 2008.
- [5] S. Bonacina, P. L. Lanzi, and D. Loiacono, "Evolving Dodge Behavior for OpenArena using Neuroevolution of Augmenting Topologies", Lecture Notes in Computer Science 5199, Workshop in Parallel Problem Solving from Nature - PPSN X, 10th International Conference Dortmund, Germany, September 13-17, 2008.
- [6] B. Geisler, "An empirical study of machine learning algorithms applied to modeling player behavior in a 'First-Person Shooter' video game," Thesis Master, Dept. Computer Science, University of Wisconsin.
- [7] S. Zanetti, and A. El Rhalibi, "Machine Learning Techniques for FPS in Q3," Proc. of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology, 2004, Vol. 74, pp. 239-244.
- [8] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels, "Evolution on Reactive Rules in Multiplayer Computer Games Based on Imitation," Lecture Notes in Computer Science, Advances in Natural Computation, pp. 744-755, July 2005.
- [9] C. Thureau, and C. Bauckhage, "Learning human-like Movement Behavior for Computer Games," Proc. 8th Int. Conf. on the Simulation of Adaptive Behavior, 2004.
- [10] C. Thureau, C. Bauckhage, and G. Sagerer, "Combining Self Organized Maps and Multilayer Perceptrons to Learn Bot-Behavior for Commercial Computer Games," Proc. GAME-ON, 2003, pp. 119-123.
- [11] <http://botprize.org/>
- [12] <http://www.cs.rit.edu/~jdb/gamebots/about.php>
- [13] <http://artemis.ms.mff.cuni.cz/pogamut/tiki-index.php>