View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by GSTF Digital Library (GSTF-DL): Open Journal Systems (Global Science and Technology...

Universal Algorithm for Creating A Small Scale Reusable Simulation Data in Real-time Strategy Games

Damijan Novak, Aleš Čep, Kristjan Košič, Domen Verber
University of Maribor
Maribor, Slovenia

Abstract— **Real-time strategy games are of such high complexity that consideration of trying to brute force all actions and states is not only impractical, but impossible. Approximations, information abstractions, and models are, therefore, the necessity when creating game bots that play this genre of games. To create such bots, the detailed data is needed to base them on. This article introduces a universal algorithm that creates reusable simulation data of one attacking unit on a building and tests the feasibility of doing such a task. This paper concludes that capturing all relevant data in a sub-segment of real-time strategy games is feasible. Gathered data holds valuable information and can be reused in new research without the need of repeating the simulations.**

Keywords - real-time strategy games; simulation; game data; game engine; Spring; game bot; AI

I. Introduction

Tic-tac-toe and chess are two examples of games widely known to the general population as classic board games` representatives, with a playing history of more than a thousand years. Strategy games, which are the main point of interest in this article, are based on classic board games. Game mechanics (and gameplay) are very alike. Players perform in turns. Players use different strategies to try to gain an advantage over one or more opponents by making moves over a series of turns. During the game, the outcome of each move is evaluated, with the possibility of adapting the strategy to the new circumstances.

Real-Time Strategy (RTS) games are a sub-genre of computer strategy games. They are usually the simulations of war games, with many aspects which must be taken into account by players. Aspects can be of different points of view: Hierarchy abstraction (strategy, tactics, reactive control), gameplay behavior (resource gathering, diplomacy, intelligence gathering, terrain analysis, etc.), game world mechanics (uncertainty, partial information, luck, non-deterministic behavior, etc.)... The point of view is dependent on the covered abstraction to be taken into account. Therefore, computer operated players (game bots or bots for short) are of high complexity and must process huge amounts of data and information in a very short time frame.

For a bot to be successful in playing an RTS game it can resort to extensive processing of game world information (before actually making a move), which includes, but is not limited to: Executing simulations of common battle scenarios, building models and representations of the game worlds, creating knowledge databases, searching for patterns, etc. Execution of simulations is an especially interesting choice for offline (and to some extent also for online) processing and was the method of choice for many research articles (more about it in the Chapter on Related Work). Time and computer resource demands can be so high (which are typically just a precise part of the gameplay, e.g. scenario of two tanks battling two rocket marines) that the extent of performed simulations is very limited.

The problem of limits hinders the size of performed simulations. Simulations are limited currently by attributes such as: The number of units that can battle, proportions of scenario game world size, time needed for executing low-level game commands, etc. In this article, the feasibility of brute forcing the specific segment of battle in advance, which will create reusable simulation data, is studied. To reach our goal the intelligent automatic algorithm was designed. Algorithm is universal across different game worlds and game rules. The core of the algorithm executes basic simulations using low-level game commands (that can, because of their low-complexity, be performed in reasonable time) and creates a large volume of reusable simulation data. Data extraction and purification can serve for tackling the problems of various segments of RTS game worlds (abstraction of the game world information space will lower search space).

The structure of this paper is as follows. Related work is presented in the second Chapter. In the third Chapter, the universal algorithm background is introduced and search space size is discussed in more detail. The requirements for algorithm are set and the universal design is created. In the fourth Chapter, the experiment and algorithm assessment feasibility is made, based on a large data acquisition. In the fifth Chapter, possible improvements for faster data gathering and an extrapolation are discussed, related to the current state-of-the-art game engine StarCraft™. This paper concludes with a positive feasibility evaluation of our universal algorithm.

DOI: 10.5176/2251-3043_5.2.364

© The Author(s) 2017. This article is published with open access by the GSTF

II. RELATED WORK

What sets RTS games apart from other games is their general complexity. Game complexity is best presented with the branching factor, b, and the depth of the game, d. This gives us a total game complexity of b^d [1]. Comparing game complexity between the game of chess (the complexity is set to be b ≈ 35, d ≈ 80) and a very famous commercial RTS game StarCraft™ (an estimate of $b \in [10^{50}, 10^{200}]$, d ≈ 36000). Games are a magnitude of complexity classes apart [2]. Chess is not an easy game problem, yet, a computer program beat the World Chess Champion so long ago (the year 1997) that it is now considered history and it may not be the best comparison anymore. The game of Go, which is the latest of the games that attracted lots of media attention after AlphaGo beat a world-class player, has a game complexity of b ≈ 30 – 300, d ≈ 150 – 200 [3]. This is a considerable step up in complexity, but still far away from the complexity of RTS games, which is the consequence of the enormous search space (more about search space in the Chapter 3. B).

Computationally playing combat games with excelence is hard [4] and in practice approximations are being used. The goal of this line of research (approximations) is to handle larger combat scenarios with more than 20 units on each side in real-time (current real-time varies in the range of 8 vs. 8 units). The authors also stated that new spatial and unit group abstractions are needed to reduce the size of the state space [5].

In [6] they found appropriate solutions to their anti-group selection problem by offline learning and employment of a simple lookup procedure. They didn't need to execute simulations because the learning process was performing well with correctly chosen and simple substitute objective function (caution, it must be stated that only the relative strength of units was considered, not including parameters such as the speed of units).

Compact representation of group behaviors in a combat as a combination of influence maps and potential fields' parameter had a restricted search space to 2^{48} [7]. Authors state that genetic algorithms always find high-quality solutions, while faster methods (like Hill Climber) find quality solutions in 50% to 70% of the time.

If we combine all the mentioned information in the above articles an association thinking pattern emerges, which reads as follows: Obviously, search space of RTS games is too big even to start considering trying every possible game action. Therefore, in practice, approximations are necessary. Executing time and resources expensive simulations for every cycle, a result of a decision in advance is not always necessary. Usage of restricted search spaces can still produce satisfying and quality results. Information can be captured in different forms of knowledge and data representations.

In the following Chapter practical use of thinking pattern is applied.

III. UNIVERSAL ALGORITHM

A. Background

During the game-play, the player (or bot) chooses some units and commences some commands (e.g. move, shoot, harvest, etc.). The player can combine moving and some other commands into one directive. For example, she/he may initiate the attack to an adjacent opponent's unit which is out of range for the given weapon. The game engine will initiate the movement toward the target (using some path finding algorithm) and commence firing when the target is in range.

The units in a game are categorized into several unit types. Each unit type has a set of characteristics (attributes) that determine the essential behavior and the outcome of interaction of the units with the environment and with the other units in the game. Initial values of those features are predetermined. When a unit of some type is instantiated, the features are set. However, the values may change during the game play according to the game mechanics. For combat scenarios, the most relevant characteristics are the number of "damage points" that the unit may produce with its weapon, the rate of fire, and the number of "hit points" the unit can sustain before it is destroyed. When a unit hits the other, the value of the damage points converts into the hit points according to some function. Most RTS games try to simulate lifelike behavior of the combats. In this, the game mechanic ensures that the hits are not always 100% accurate. The calculation behind this can be very sophisticated. Frequently, the game engine tries to mimic the physical laws of nature and the behavior of actual weapons. The damage can be further reduced by the unit "shield" amount, the natural obstacles on the terrain, etc.

The interaction between the units must be evaluated to derive the wining tactical strategy of the bot. Because of complex interaction and the sophisticated game mechanics, this can be time-consuming. An advance forecasted evaluation would be very beneficial.

B. Search space

The search space for even the most general RTS game is much larger than for any board game. A game like chess has a discrete number of possible states. In chess, there are 64 possible places where a chess figure can be placed and only up to 32 pieces can be placed on the board at the same time. Furthermore, each piece has only a limited set of legal moves and interactions between the figures are limited. On the other hand, the RTS game is being played in continuous space. They are usually played in the large game world with real-type coordinates systems. Each unit can be placed almost anywhere on the game map. Furthermore, it is not uncommon that there are several hundred units interacting with each other at the same time. Also, the numbers of legal actions that can be taken with each unit are immense, and the interaction of one unit with the others may be very sophisticated.

This paper systematically explores different aspects of inter-unit interaction. Firstly, a set of variables that can be influenced by the bot and a set of variables that are the result of interactions determined by the game-engine itself must be identified. In the end, the experiments require data mining techniques. Therefore, we decided to use similar nomenclature as in data warehousing. The "dimensions" represent the independent variables that can be affected directly by the bot. These are choosing type and the position of the unit and the performed action. The "facts" are some intrinsic characteristic of units, the observable results of the interactions and some

© The Author(s) 2017. This article is published with open access by the GSTF

derived values important for the bot implementation. The intrinsic characteristics of units are the value of attributes defined by the game engine for each unit type. For example, a combat unit would have defined attributes such as the maximum damage per hit, maximum weapon range, maximum rate of fire, etc. The observables are defined by the result of interactions between the units. The most obvious one is how much damage is taken by each hit. This depends mostly on unit types, on relative position, velocity and alignment between the units. The amount of damage may be stochastic and determined with some continuous probabilistic function. In general, the observable facts are the only values available to the programmer. The inner working of the game engine is not usually revealed. The derived variables are determined from the other facts. For example, whereas the amount of damage taken is important, more important is the time during which one unit can completely destroy the other. In a one-to-one combat scenario, the unit with a shorter destruction time will almost certainly win.

This research focuses on two observables: The amount of damage produced by the unit with each shot and the overall time required to completely destroy the opponent. The damage is measured in generic units imposed by the game-engine (called damage points). The time is measured in game-frames. Different spatial configurations of units are observed and observable facts are recorded. To cope with the stochastic nature of observables, experiments have to be repeated multiple times with the same configuration. A detailed probabilistic function of each variable is usually not needed. To reduce the complexity, the values can be converted into the discrete space by using binning or similar devices.

The number of configurations included in the experiment can be reduced by knowing some intrinsic values of units, by observing symmetry, considering the rate of change of facts, etc. It is also possible that the relative angle between units has no significant impact on the results. Therefore, the results would depend only on the relative distance between the units. Also, the small changes in distance between the units would produce similar results. Therefore, the dimension of distance can be transformed from a continuous variable to a discrete one, using the intervals. The boundaries of those intervals are set by the amount of changes of the results. For example, the quantiles of measured probabilistic distribution function of the damage can be used. In an extreme case the distance has no effect on the results at all. Outside the maximum range of the weapon no hits are possible. Inside the weapon range the hits are equally probable regardless of the position.

C. Intelligent automatic algorithm for creating reusable simulation data

In this Chapter the algorithm written in universal form is presented. It can be used in any classical real-time strategy game (the algorithm is presented in pseudocode).

For the algorithm to work we:

Presume - every game unit has a read-only attribute set available (usually presented in the form of unit definition). The game world can introduce variable factors (randomness) which contribute to small deviations when constant values are used during gameplay. Static units that fire back (e.g. a fortress with guns on top of it) are not supported. Dynamic units must fire. The engine must have support for the execution of independent simulations.

Require - deviation from the relative damage value of the unit (threshold). A maximal number of simulations executed per point that meets the criteria on one pair of unit and building. A time limit for execution of one simulation (one pair of unit and building). A set of all unit definitions and a set of all building definitions.

Define – point: The value of a point is set by the game engine and it represents the minimal amount of distance that a unit can travel on the map.

```
select threshold, maxNumberOfSimulations, timeLimit
set attackUnitsSet = getDefinitionsOfAllAttackUnits()
set buildingsSet = getDefinitionsOfAllStaticBuildings()
for each attackUnit in attackUnitsSet do
    for each building in buildingsSet do
        set maxDamageRange =
            getMaxDamageRange(attackUnit)
        set pointsSet =
            getValidPoints (building, attackUnit,
                maxDamageRange)
        set smallSetOfPoints = getFewPoints(pointsSet)
        preTestingSimulation(attackUnit, building,
            smallSetOfPoints, threshold, size(pointsSet),
            maxNumberOfSimulations,
            out numberOfSimulationsNeeded,
            out timeEstimate)
        if numberOfSimulationsNeeded <=
            maxNumberOfSimulations  and
            timeEstimate <= timeLimit
        then
            set data =
                executeSimulation(attackUnit,
                    building, pointsSet,
                    numberOfSimulationsNeeded)
            addToRepositoryOfData(uniqueKey(attackUnit,
                building), map)
        else
            addToFailedSimulations(uniqueKey(attackUnit,
                building))
        end if
    end for
end for
```

Word descriptions are used for methods that are called inside the algorithm instead of the actual implementation. The reason for this lies in the fact that the implementation of the algorithm and its methods is dependent on the API's, which are different for every game engine (in our case Spring engine). A specific implementation of the methods would only act as a distraction. Methods are described in the order that they first appear in the algorithm.

getDefinitionsOfAllAttackUnits: The method obtains all unit definitions (set of read-only attributes) from the game

© The Author(s) 2017. This article is published with open access by the GSTF

engine, sifts through them and returns only those units which have the capability of attack.

getDefinitionsOfAllStaticBuildings: The method obtains and returns all unit definitions which have an in-game role set to a static building.

getMaxDamageRange: The method receives as an attribute the unit definition set of the attacking unit. Then, a search is made through the attribute set. If a maximal damage range (the maximal range value from where a target can be hit) is found, the value is returned from the method. If the value is not found, the method executes the simulation that obtains it. Simulation = a unit is being moved point by point in a linear way from the center. For every point, execution of a test occurs, which determines if a unit can still hit the center. From the last point, from where the unit can still make the hit, and the center point, the distance is calculated and returned from the method.

getValidPoints: The method receives three attributes: A definition set of a static building, a definition set of an attacking unit and the value of maximal damage range. Simulation is then executed which determines the valid points. All the valid points are returned from the method. Simulation = algorithm tries to position the attacking unit on a specific point on the battlefield. Every point in the circle is tested (with a radius of maximal damage range) around the building. If a unit cannot be set on a specific point the point is not valid (e.g. the game engine does not allow positioning of a unit on a specific point because there is an obstacle – a building is covering the point). If a unit can be positioned on a specific point the point is valid.

getFewPoints: The method receives the whole set of valid points and returns a sample representative of five simulation points (e.g. Fig. 1).

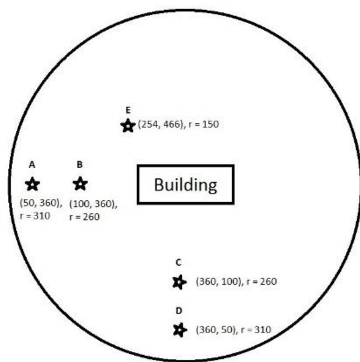

Figure 1. Five simulation points

Points that are returned must create linear (A-B and C-D) and circular (A-D, B-C) dependencies. Point E is chosen randomly (does not depend on points A, B, C and D).

preTestingSimulation: The purpose of the method is to get the minimal number of simulations needed to satisfy the threshold (maximal deviation from the relative damage value of the unit) parameter and to get the time estimate of the whole simulation. Input parameters into the method are: (i) a definition set of the attacking unit, (ii) a definition set of the building, (iii) small set of points, (iv) the maximal threshold deviation that must be satisfied and (v) the maximal number of simulations that can be executed on a single point. If the maximal number of simulations is reached, but the threshold is not yet satisfied, the output of the number of simulations needed is set to maxNumberOfSimulations + 1.

executeSimulation: The method receives the following parameters: (i) a definition set of the attacking unit, (ii) a definition set of the building, (iii) a set of all the points and (iv) the number of simulations to run on every point. The method executes the simulation on all the points and returns/saves the simulation data. Simulation = is a series of smaller simulations, where the attacking unit is set systematically on every point of the points set and is shooting at the structure until the structure is destroyed. Every simulation is recorded and saved.

uniqueKey: Returns a unique key consisting of a pair, attack unit and building.

addToRepositoryOfMaps: Created data is saved in a data storage under a unique key for further use.

addToFailedSimulations: If execution of the simulations has failed, the unique key is saved into a data structure of failed simulations.

IV. ASSESSMENT OF ALGORITHM FEASIBILITY

A. *Experiment: Game engine setup and data storing*

Experiment was conducted with the real-time strategy game engine Spring. Spring is a free and open-source 3D game engine, which was developed originally to bring the game Total Annihilation back to life, but has since then got support some other games. The experiment was carried out with the game Balanced Annihilation 7.63. Game options were left at default except that the option “No Unit Wrecks” was enabled. Game speed was set to 1.0 to avoid any random effects that could be caused by the game engine with increased game speed [8]. The map “blue fields” was used where the y coordinate does not change (flat map). We developed two game bots (dynamic and static unit) in the JAVA programing language. Dynamic units were tanks (game unit name “armstump”), static units were non-attacking buildings (game unit name “armlab”). Simulations were run simultaneously on ten Intel Core i3-4130 CPU @3.40 GHz, 8GB RAM and GeForce GT-610 computers using the Windows 7 operating system. RAW data from each simulation was collected and stored in an opensource NoSql database – MongoDB. MongoDB database was setup on a central server. Raw data was saved asynchronously, in the simulation cleanup phase - so there were no time/lag effects on the game play.

Before simulation start, a set of unit definitions from the game engine was retrieved. From the dynamic unit definitions information about the maximum weapon range was obtained, which gave us a circle of all valid locations. This is known as the Gauss circle problem. An experiment was ran to confirm that a unit cannot be set to a location where a unit from another team is. The Spring game engine moves a second unit to the closest available point if the point in question is not available (making the point invalid). Based on that information, all locations covered by the building were removed (Fig. 2). Pre-

© The Author(s) 2017. This article is published with open access by the GSTF

testing simulation was conducted on previously selected points (Fig. 1). The threshold was set to 0.151 (the threshold number was set by observing the results of a few previous test runs, where the damage difference between maximal and minimal value stabilized at the range of 10^-3) and a maximal number of simulations was set to 1000. Pre-testing simulations gave us a time estimation of 30 seconds for each simulation and 182 runs needed at one location.

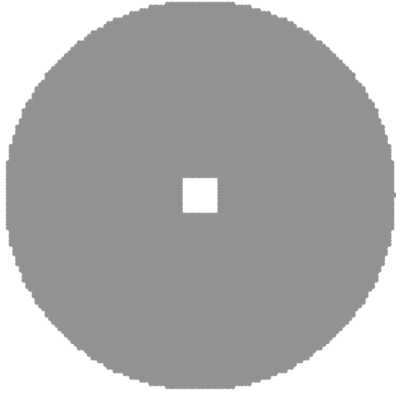

Figure 2. Image showing 2D view of all valid points

In simulations the static unit was always set at the same location in the center of the circle (relative position [0, 0]) and a dynamic unit on each location in the set of valid locations. When both units were set to map, the dynamic unit started to shoot at the static unit. After each shot at the static unit the Spring engine triggered an event where damage and the current frame were stored. When the static unit was destroyed the current location of the dynamic unit, current time, number of passed frames since positioning both buildings on the map and all the damages with corresponding times were saved to the database.

B. Assessment of large data acquisition

The whole data consisted finally of roughly 50 million JSON objects. It took 13 days to gather this data. From the practical point of view that is a long time and massive volume of data for just one data gathering, but from the theoretical viewpoint it not only shows such a thing is possible to do in the foreseeable future but also that it can be done with relatively cheap equipment.

This experiment was focused on the mean, minimum and maximum of the gathered values. To evaluate the spreading of values and the significance of measured results we also evaluated the standard deviation. 5% of collected RAW data was not valid, due to randomness of bot straying or halting. This was traced back to the game engine and data was excluded from final calculations. JSON simulation object included data about damage (min, max, avg), time related information (start frame, end frame, frame difference) and location details (posX, posY).

Gathered RAW data indicated that there is no significant difference in damage related to attack bot location (standard deviation of average damage was 0.1%). Data mining process identified other significant information related to time diference (frame diference with a standard deviation of 3.6%).

Basic RAW data can be observed in Table I (frames and damage points are presented), where unit position is ignored and no additional information processing was performed to preserve data integrity.

TABLE I. PRESENTATION OF RAW DATA

	Min.	*Max.*	*Avg.*	*St. dev.*	*Diff.*
Time	960	1050	1013	3.62	90
Damage	85,02	89,91	86,94	0,03	4,89

For instance, a combination of bot location, damage performed and kill speed (frame difference) could be used for data inquiry about attack bots next move. However, more detailed data mining has to be performed to confirm this hypothesis.

V. DISCUSSION AND CONCLUSION

The main point of this article was to introduce an intelligent automatic algorithm for creating reusable simulation data and to assess the feasibility of acquiring such data. To test the feasibility of acquiring large data an experiment that consisted of running the algorithm for one iteration was designed. The experiment was run at the native game speed of 1.0 which, on ten standard office computers, took roughly two weeks to complete. It is a long time for executing one iteration, but there are a lot of improvements that can speed up the experiment significantly. The most obvious one would be to run the experiment at a game speed of 10.0. This would lower the experimentation time to roughly one day, but the randomness impact would also have to be assessed so that results can be scaled back to the speed of 1.0. The pre-experimentation showed that our computer resources allow a game speed of 10.0, without introducing a lag between frames which could influence the experimentation and lead to unreliable data.

Another improvement would be to run the experimentation without actually running graphical interface (rendering of game objects uses a lot of computer resources), which would enable even higher game speeds with no lag between frames. It would also make possible to run parallel game engine instances on a multi-core processors. A game engine currently only uses one core with others being idle and it allows only one active game instance. If virtual machines are used the problem of multiple instances of game engine accessing graphic drivers occurs. No graphical interface rendering could solve this problem.

Improvements on the experimentation data collection side are also possible. For one iteration run of the algorithm all RAW data is saved. That allow us to process and analyze the data with different big data techniques, which could lead to learning new knowledge in the future. For practical use, there is no need to save all the data that simulations create. Data can be processed and information that is needed extracted while the experiments run (for example on a dedicated data server).

Current state-of-the-art research is focused on the game engine of StarCraft: BroodWar™ 1.16.1 (SC: BW for short) with an application programming interface called BWAPI, which enables replacing the player interface with C++ code. BWAPI for SC: BW allows running the game with different

© The Author(s) 2017. This article is published with open access by the GSTF

speeds and without using graphical interface. If we extrapolate the one day needed for the experiment in Spring to the SC: BW (with full knowledge that the differences between game engines could lead to different experiment execution times), it means that, to make a complete universal algorithm run it would take 144 days (race Terran - for all the combinations of non-upgraded eight ground units to eighteen buildings). The time needed for experiments is linearly dependent with the computer resources available, meaning that the actual running time would be much lower in practice.

To conclude, our experiment of a universal algorithm on the RTS game engine Spring showed that it is feasible to gather every data that an attacking unit does on a building on a flat terrain in foreseeable time. This also proves that, although the RTS games have an enormous overall game complexity, reducing the complexity by brute forcing of specific (sub)segments is actually feasible, and the data acquired can be used for further use. If the scenario of "one-on-one" scales up to "many-to-one", or even "many-to-many", combat scenarios, that would be a significant improvement in the field of state space abstraction.

REFERENCES

[1] G. Synnaeve, "Bayesian Programming and Learning for Multi-Player Video Games," Ph.D. dissertation, Universite de Grenoble, 2012.

[2] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," in IEEE Trans. on Comput. Intell. and AI in Games, vol. 5, no. 4, pp. 293–311, 2013.

[3] G. Synnaeve and P. Bessiere, "Multi-scale Bayesian modeling for RTS games: an application to StarCraft AI," in IEEE Trans. on Comput. Intell. and AI in Games, 2015, in press.

[4] T. M. Furtak and M. Buro, "On the complexity of two-player attrition games played on graphs," in Sixth Artificial Intelligence and Interactive Digital Entertainment Conference, 2010.

[5] D. Churchill, S. Abdallah, and B. Michael, "Fast heuristic search for rts game combat scenarios," in AIIDE, 2012.

[6] N. Beume, T. Hein, B. Naujoks, N. Piatkowski, M. Preuss, and S. Wessing, "Intelligent anti-grouping in real-time strategy games," in IEEE Symposium On Computational Intelligence and Games, pp. 63-70, 2008.

[7] S. Liu, S. J. Louis, and M. Nicolescu, "Comparing heuristic search methods for finding effective group behaviors in RTS game," in IEEE Congress on Evolutionary Computation, pp. 1371-1378, 2013.

[8] S. Liu, S. J. Louis, and C. Ballinger, "Evolving effective micro behaviors in RTS game," in IEEE Conference on Computational Intelligence and Games, pp. 1-8, 2014.

Authors' Profile

Damijan Novak graduated in 2011 in Computer Science at the Faculty of Electrical Engineering and Computer Science at the University of Maribor. He is currently a Teaching Assistant and post-graduate student at the Faculty where he received his academic degree. His current research subjects are Computational intelligence and Artificial intelligence in computer games. [damijan.novak@um.si]

Domen Verber is an Assistant Professor at the Faculty of Electrical Engineering and Computer Science at the University of Maribor. His main research subjects are Ubiquitous computing, High performance computing and Computer games. He has published numerous papers in international journals, books, and conference proceedings. [domen.verber@um.si]

Aleš Čep graduated in 2014 in Computer Science at the Faculty of Electrical Engineering and Computer Science at the University of Maribor, Slovenia. He is currently a Teaching Assistant and postgraduate student at the same Faculty. His current research subjects are ubiquitous computing and computer games. [ales.cep@um.si]

Kristjan Košič graduated in Computer Science at the Faculty of Electrical Engineering and Computer Science at the University of Maribor, Slovenia. He is currently a Teaching Assistant and postgraduate student at the same Faculty. His current research subjects are ubiquitous computing, human computer interaction and big data. [kristjan.kosic@um.si]

© The Author(s) 2017. This article is published with open access by the GSTF