STUDIA Z AUTOMATYKI I INFORMATYKI
TOM 42 – 2017

Paweł Stawarz*

IMPLEMENTATION OF A DETERMINISTIC VIDEO GAME AGENT TESTING ENVIRONMENT

Keywords: Multiplayer video games, testing environment, artificial intelligence, replay system

1. Introduction

Video games play an important part in everyday life, improving the economy, health, art and even decreasing the impact of depression [3]. The Entertainment Software Association states that over 42% of Americans play video games regularly [4, 5]. A more in-depth look at the subject reveals that nearly all of the highest revenue games sold in the US either are limited to online gameplay or are multiplayer focused [27]. This statement seems to hold true even regarding worldwide data [29].

1.1. Artificial intelligence in computer games

An ever-important part of modern games is artificial intelligence (AI) [20]. One that will prove itself challenging, realistic and fair to the players. The fact that human behavior and strategies evolve and tend to become different as players explore the game [1] impedes the development of such AI.

Evolution of player strategies calls for the development of methods and algorithms that can self-adapt to the game state and is a challenge for scientists all around the world [20]. *Adaptation*, in the case of AI-controlled agents, should be understood as the ability to adjust behavior depending on the players' actions to fit a predefined, particular role while building an illusion that the player is facing a human being. In some cases, this can be achieved by developing a complex and sophisticated finite-state machine. However this approach is not sufficient when it comes to more advanced behavior, due to the complexity of human actions. A vast number of adaptive solutions had been tested and proven too slow or too static to find use in online games [18, 22, 24, 8]. In earlier work, the manuscript's author designed a pathfinding algorithm and postulated that it could be utilized in controlling autonomous robots [26].

1.1.1. Characterization of the problem

The goal was to develop an adaptive pathfinding algorithm for a simple top-down multiplayer, competitive video game agent. The game consisted of matches, each lasting about 15-20 minutes. A small, yet arbitrary number of players would be present, as well as a small, arbitrary number of agents. The agents interpreted the map as a digraph structure, in which

*Rzeszów University of Technology, Faculty of Electrical and Computer Engineering, Department of Computer and Control Engineering, 2, W. Pola Str., 35-959 Rzeszów, e-mail: p.stawarz@prz.edu.pl

the vertices represented turns, crossroads and strategic points on the map and edges served as the roads connecting those points. The simplified process of converting a game map into a graph has been presented in Figure 1. Please note that Figure 1 is simplified and does not show a digraph, but an undirected graph. The conversion process is analogical when converting the map into a digraph, but splits each path into two separate edges.

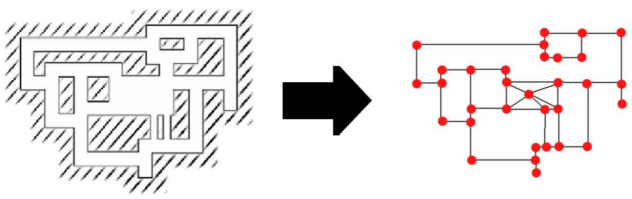
Fig. 1. An example video game map with the corresponding map graph

Let us assume that the map has only one planar level, contains open spaces and tight corridors. When a player joins the match, or his avatar does so, they need to be respawned, a random point from a predefined list of spawn points is picked. If any other participant is present within a set distance of that point, a different point is picked.

The match ends whenever any of the human or AI-controlled characters destroy a set number of other participants.

The algorithm must imitate live player movements, present consistent behavior, that varies on a per-player basis.

1.1.2. First tests and their effects

To fulfill all the requirements presented in Subsection 1.1.1, an algorithm has been designed that is based on re-weighting the edge values in a map graph depending on the events that happen during the current and previous matches. Every time an event occurs (each time one player kills another), the weights of the edges corresponding to the player's current location are modified, as below

$$e'_x = e_x + (1-r)\delta_k - r\delta_d\,, \tag{1}$$

where:

$$\delta_k = (1-m)\left(\sum_{i=0\,,i\neq j}^{N} k(i)\right) + m\left(\sum_{i=0\,,i\neq j}^{N} k_p(i)\right)\,, \tag{2}$$

$$\delta_k = (1-m)\left(\sum_{i=0\,,i\neq j}^{N} d(i)\right) + m\left(\sum_{i=0\,,i\neq j}^{N} d_p(i)\right)\,. \tag{3}$$

In (1)–(3), *r* and *m* denote the risk and memory coefficients introduced in [26]. The edge values of the kills and deaths graph of participant *i* are denoted as *k(i)* and *d(i)*, k_p *(i)* and

d_p *(i)* contain data gathered during previous matches, and *N* is the total number of game participants and *j* is the identifier of the agent. For detailed information about the idea and the algorithm, see [26].

The algorithm has been tested in an offline environment in which one player competed against three AI-controlled opponents. The Unity Engine served as a base for creating the environment.

Seven unique players have been asked to play 30 matches each. For the first 15 matches, a standard pathfinding algorithm has been used, and data has been gathered about the players' movements and places in which major events happened (players died or destroyed other players' characters), and for the next 15 matches enhanced pathfinding algorithm has been used.

The players themselves have all volunteered for the tests. The age ranged from 18 to 65 years. Three of the participants were game designers, three have played computer games before, and one never played a similar video game. After the tests, all the participants were questioned and asked how they feel about the AI. All participants described the last 15 matches (the ones with the enhanced pathfinding algorithm) as more compelling and lengthy. Information about the game time has not been stored. Thus it was impossible to verify the second part of the statement after the tests.

However, the gathered data seems to agree with the first part of the statement and shows that the algorithm works in simple top-down games and exhibits different behavior when facing different players [26]. As the tests have been performed offline, the agents have not been tested in an environment where multiple live players exist simultaneously. This itself makes further tests a requirement, which in turn forces the creation of a better testing procedure.

However, player actions tend to be non-deterministic, rendering standard testing procedures insufficient, since each change in the algorithms' argument values requires a new set of tests. Those tests need to be either repeatable to exclude the possibility of a random event impacting their result or repeated multiple times. The first solution enforces a high degree of determinism and thus cannot be performed with live players [1]. On the other hand, putting the agent only against other agents does not test it's abilities on a live battlefield [6].

1.2. Existing artificial intelligence testbeds

Artificial intelligence that is meant to interact with humans has to be thoroughly tested. Pressure exists on virtual agents to remain believable in changing environments, but also predictable enough to fine-tune during the development phase of the product [28]. Testing is crucial and forces the development of various environments that allow researchers and developers to observe the agent before showing the effects to the end-user.

Tileworld is one of the oldest and well-known testing environments for artificial intelligence [17]. The solution developed by Pollack and Ringuette in 1990 can be used even in present times [12], [11]. Tileworld is a single player dynamic environment which allows testing of decision-making systems. Implementing the algorithm proposed in [26] is possible; however, testing requires only the simultaneous presence of multiple players, which is something Tileworld does not support.

In 2002 researchers from the Information Sciences Institute of the University of Southern California's developed an environment called GameBots [10]. GameBots is a 3D online multiagent AI testbed based on Epic Games' Unreal Engine. All the actors are allowed to move freely around a set of maps and compete against other players in multiple game modes, including Deathmatch, Capture the Flag and King of the Hill. The environment is free, flexible and robust, but does not gather information about the players. A solution would be to implement a data-gathering server, but even then, the information would be highly incomplete and unsuitable for our needs, which is why the solution has been rejected.

At roughly the same time, two students from Technion, Israel; Jonathan Nave and Shmuel Markovich; developed a testing environment to explore the performance of artificial intelligence path finding algorithms [9]. The project, called by the authors the Artificial Intelligence Benchmark (or AIB for short) allows for testing grid-based pathfinding solutions. The algorithm from [26] requires a graph-based map representation with a high degree of freedom, rendering the AIB insufficient.

In 2005 Matthew Molineaux and David W. Aha presented the alpha version of TIELT [14]. TIELT is a middleware solution that allows combining an arbitrary game engine with learning-embedded decision systems. Researchers are required to define five distinct knowledge bases before the system can understand how to communicate the game engine with the decision system. Even when taking into account how the system has grown through the years, it remains unsuitable for our needs. The motivation behind TIELT was to create a system that will support online, offline and recorded training sessions. We failed to find any data compatible with TIELT that could be used as a base to teach the algorithm described by (1).

Also in the same year, Matthew Molineaux, David W. Aha et al. described Stratagus [19] and its integration with TIELT. Stratagus is an open-source real-time strategy game engine, widely used as an artificial intelligence testbed. Stratagus is suitable for testing AI only in RTS games and thus is unsuitable as a testbed for the solution proposed in [26].

The GVG-AI competition introduced in 2014 [16] gained momentum after its 2016 edition [15]. The competition contains a framework that allows programmers to define the gaming environment with VGDL (video game definition language) and then implement agents that compete to achieve the best score. The agents probe the environment by querying the game state via a forward model and react by performing one of six predefined actions: moving in one of the four directions, using an item or doing nothing. The frameworks' simplicity and extendability are excellent when it comes to defining multi-purpose agents, but single-purpose agents designed solely for competitive, multiplayer games require a wider range of actions, rendering the framework insufficient for the analyzed problem.

Yaqing Hou, Yew-Soon Ong et al. employed the Unreal Tournament 2004 video game to test their eTL (evolutionary transfer reinforcement learning framework) [7] in 2016. An existing video game has been considered in the role of a testing ground, but a decision has been made to develop a dedicated testing environment instead. The examined algorithm is focused solely on improving pathfinding techniques, and modern video games convey much unneeded information. Complexity also increases implementation time.

From existing game engines, we have decided to pick the Unity Engine. The engine has proven useful for similar research [23], and we had prior experience with it.

2. Avoiding determinism

Since including human players into the equation virtually excludes determinism as an idea, our solution is to develop a system that does not require determinism but offers predictable results.

The most important part of the system is the replay component. During the Game Developers Conference in 2013, Mark Wesley, Lead Gameplay Programmer for 2K Marin pointed out that such a system is helpful during the development phase [13]. Data about the player's movement is recorded during online tests, but the agent itself is trained offline, utilizing the replay system. The results of the training can be compared because the presentation of the agent remains the same during the tests.

It is important to note that gathering player data is a common practice in modern and popular multiplayer games [25, 21].

2.1. Study of possible solutions

There exist two main approaches when it comes to designing a replay system: incremental/deterministic and save-state based. Both solutions store recorded information in contrasting ways, offering different opportunities [2]. Figure 2 offers a visual representation of the distinctions.

Incremental replay systems store changes that occur each game frame. These changes are mostly caused by player input events or are the effect of background processes, like the AI or physics systems. The recording is done by monitoring the user input and background system state changes (if any). As the inputs carry the knowledge regarding the future frames, incremental replay systems are also known as deterministic replay systems.

The primary gain of the incremental approach is the relatively small size of log files. This asset may seem to not matter much at present times (as storage space is rarely an issue), but it is still of importance when data is stored on a remote machine. A different, but not less significant advantage, is the presence of data regarding human reaction speed and common input mistakes.

As the next game state is deduced from the current one, the replay cannot be simply started from any point of time, which is the most significant liability of incremental replay systems.

The save-state based approach requires storing the game state directly, instead of recalculating it each frame. Logical consequences include large save file size and the ability to start the replay from virtually any point in time.

Incremental and save-state based systems can be combined to form a hybrid replay system, that stores state information once every set number of frames and computes all the missing information between the keyframes.

2.2. Implementing the recording system

Various tests had been made before picking the right solution. Our first approach was to implement an incremental replay system, as it conveys more information about the players, such as their reaction time, and input errors. Since the final product has been built using the Unity Engine, the data has been gathered asynchronously each time a player pressed an input button and then has been annotated by a timestamp. This approach collided with the testing environment, where actions have been executed at fixed intervals, thus making the player characters move differently.

Attempts have been made to read the inputs in an asynchronous matter and react to them only during the fixed updates. This approach introduced a small, yet perceptible latency in the character reaction time, which in some cases bothered the players and thus was unacceptable.

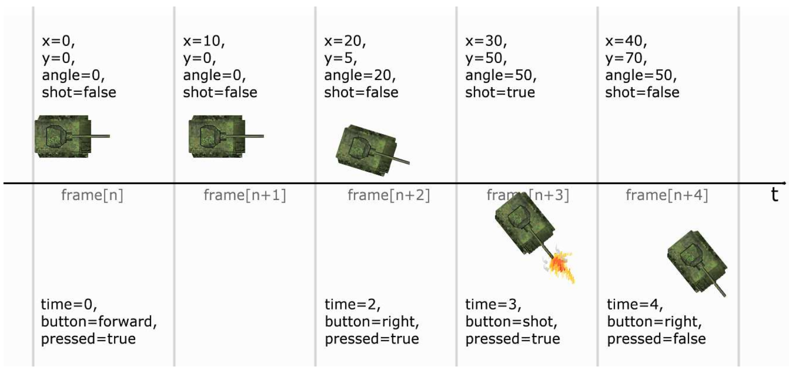

Fig. 2. Visual interpretation of data stored by recording systems. Upper values represent information stored by a save-state system, whereas the bottom values represent information stored by an incremental recording system

To fix the issue, implementing a save-state based replay system was required. During final product tests, a second problem has been found, as the player avatars showed desynchronization between each other. Further research revealed that the synchronization problems had roots in the latency between the clients. The final version of the replay system consists only of the server-side recording component and a standalone viewing application. The server creates a game log that contains a list of player logs with associated timestamps. After a new player joins (including the game host), a separate log is created, containing the particular players' movements.

The resulting replay system is save-state based but also gathers information normally gathered by incremental systems. Data about events that occur in the game are stored in a game log file, described in section 2.2.2. State of player objects is stored on a per-player basis in a file, called the *player log*. The format of the player log has been described in section 2.2.3 of the manuscript. Storing excessive data was possible due to the simplicity of the engine.

Both applications are implemented using the Unity Engine. All scripts are written in C# and object-oriented programming.

2.2.1. Event system

Communication between the game and the recording system occurs via an event system. Table 1 shows a list of possible events. All events are handled by a singleton object of class `eventManager`.

Tab. 1. List of events propagated by the event system

Event name	Entry	Additional data
Match start	Host	Host players' nickname, randomizer seed (application start timestamp)
Player connected	PlayerJoin	Players' nickname, players' log filename, relative timestamp
Player quit	PlayerLeave	Players' nickname
Player scored	Score	Scoring players' nickname, killed players' nickname
Match ended	End	-

Each of the events is implemented using a trio of a delegate function, static event connected to the delegate and a static method used to trigger the delegate (see Listing 1 for example).

```
public delegate void tankShotAction(GameObject tank);
public static event tankShotAction onTankShot;
public static void triggerTankShot(GameObject tank)
{
    if (onTankShot != null)
        onTankShot(tank);
}
```

Listing 1: Example implementation of the "player shot" event, inside the `eventManager` class

2.2.2. Game log

The game log is a text file created during the server start phase, which contains information about global events that happened during the match. Each entry is stored with a timestamp. The timestamp is relative to the log creation time. Each time a match-influencing event happens, a new line is added to the log, containing the timestamp, event name, and additional data.

A list of events considered important for replay purposes has been shown in Table 2.

Tab. 2. List of important in-game events stored in the game log

Event	Propagated online?	Arguments
Player joins	Yes	Character object, nickname, log name
Player leaves	Yes	Character object
Player character destroyed	Yes	Character object
Player destroys enemy	Yes	Character object
Player spawns	Yes	Character object
Player won	Yes	Nickname
Player changes target	No	New angle
Turret acquired target	No	Current angle
Turret lost target	No	Current angle
Turret shot	Yes	Character object
Player sent message	Yes	Nickname, message text

Handling the game log is done by a class called the `gameLogger`. As only one game log is generated per match, the `gameLogger` is defined as a singleton. At the start of the match, the class registers at the `eventManager` to receive three events: `onPlayerJoin`, `onPlayerLeave`, and `onPlayerScore`. After the setup phase finishes, the `game-Logger` creates the log file and instantly outputs the Host entry. Whenever an event triggers, `PlayerJoin`, `PlayerLeave` or `Score` entries are output respectively. When the gameLogger is destroyed, the End entry is the output to the log file, and the log file is closed.

2.2.3. Player log

A player log is a continuous binary stream stored on the server side. Each log describes a single player and is essentially a series of data frames shown in Figure 3. The first part of the frame contains data necessary to reproduce the character state in each frame: position and rotation of the main part, and the angle at which the turret is facing. The second part of the frame contains information regarding player input and has been included, despite not being crucial to the replay per se.

It is worth noting that saving the timestamp is unnecessary since the records are saved at constant intervals. Only the log creation time is mandatory for the system to work.

From previous experience, it has been assumed that a single match can last for not longer than 25 minutes. The map has been designed to allow at most 5 players (including the host) to compete simultaneously. A pack of 5 player logs after a 25-minute game, assuming all the players are present from the very beginning of the match, is roughly 8.1 MB in size. The scenario of a match being so long is very unlikely to happen, and as such, the value above can be interpreted as the upper bound of the log file size.

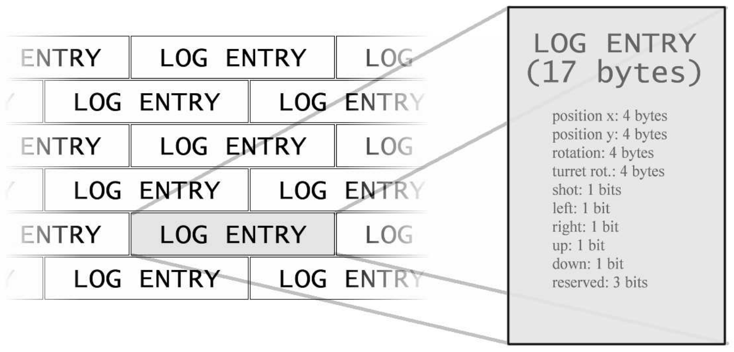

Fig. 3. Player log fragment. The size of one log entry is 17 bytes

2.2.4. Combining the information

If during the game any major events take place, the match host is prompted with a request to wait until all the files are finished uploading on a remote server, after the end of the match. Starting from the game log, files are sent to the server via the HTTP/POST method. The method is slow but has been chosen since it is already present in Unity and does not require any additional resources except a free hosting service to handle. The process can be represented by the sequence diagram shown in Figure 4.

The receiver is a simple PHP file that tests whether the data includes a certain control number and if so, moves all the received files to a set directory. All the files are treated as binary files, so no information is lost during the transfer phase. If the control number is not

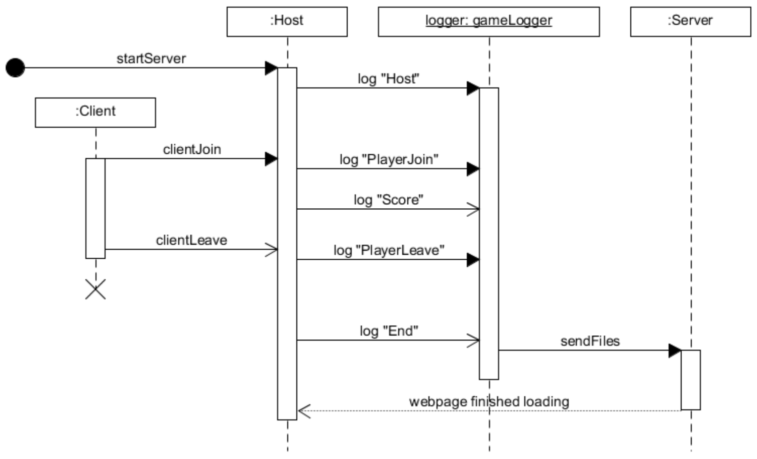

Fig. 4. Sequence diagram presenting the game log creation and uploading process

present withing the received header, the uploading is terminated to increase the security of the server. None of the files are executed during the copying phase, so there is no way of running malicious code on the remote server.

In case any of the files fail to upload, the application retries ten additional times. If each of the retries fails, the process is stopped, and the application quits.

2.3. The testing procedure

Testers are required to play a set number of matches against each other in a deathmatch game mode (i.e., each player versus all other players). The first player to reach a set number of kills wins the game, similarly as in the basic tests described in Section 1.1.2. It is crucial to note that during the tests, players do not compete against AI-steered agents, only against other human opponents. Information gathered about each player is stored by the match host during the match, and sent to the remote server after the match ends.

After recording a set number of matches, the recorded data is enhanced with space context [26]. The time context is already present due to the chronological nature of the replay. A virtual agent, steered by the considered pathfinding algorithm, is then put onto the battlefield. Other agents do not need any artificial intelligence, as their movements are defined by the replay system.

The replay is then started, and the action proceeds as it did without the agent. All the player characters perform the same actions as they did in the normal match, at the same time relative to the start of the match.

Whenever the agent scores or gets destroyed (i.e., the match statistics start to differ from the ones saved in the replay itself), the replay stops. After that, the replay is reloaded into the original state it was at the current time. The agent is once more spawned in the current state of the game and the testing procedure repeats.

The agents' performance is defined by summing the outcomes of all the tests. Depending on the designers' intentions, the risk and memory coefficients can be adjusted accordingly to the change of the agent's behavior. Assuming the algorithm controlling the agent is deterministic, all the tests are fully repeatable and will always yield the same outcome. Determinism allows the researcher to compare the tested algorithm against other solutions, even when considering multiple different criteria.

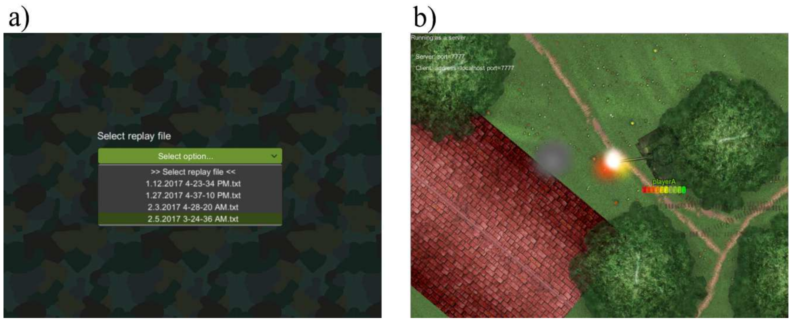

Fig. 5. Screenshots from the finished testing environment; a) replay selection menu, b) player character shooting during a replay

3. Summary

The article describes the design and implementation process of a novel deterministic artificial intelligence testbed. The proposed solutions can be used to fine-tune any adaptive pathfinding algorithm (including the one described in [26]) created to control agents in multiplayer, competitive video games. Data about the players is stored on a remote server and retrieved when requested. The final look at the environment is presented in Figure 5.

In the future, the authors plan to test the environment in action. At this moment the process of searching for volunteers to participate in the tests is ongoing, and the authors are implementing the algorithm into the testbed.

Acknowledgements

The paper has been supported by the Polish Ministry of Science and Higher Education grant-in-aid for conducting research by young scientists (Rzeszów University of Technology, Faculty of Electrical and Computer Engineering, Department of Computer and Control Engineering, contract number DS/M.EA.17.007).

References

[1] G. Christou. A comparison between experienced and inexperienced video game players' perceptions. *Human-centric Computing and Information Sciences*, 3(1), 2013.

[2] Cyrille Wagner. Developing Your Own Replay System, Feb. 2004.

[3] Entertainment Software Association. Industry Facts.

[4] Entertainment Software Association. 2015 Sales, demographic and usage data: essential facts about the computer and video game industry. Sales, demographic and usage data, Entertainment software association, 2015.

[5] Entertainment Software Association. 2016 Sales, demographic and usage data: essential facts about the computer and video game industry. Sales, demographic and usage data, Entertainment software association, 2016.

[6] Y. Hou, L. Feng, and Y. S. Ong. Creating human-like non-player game characters using a Memetic Multi-Agent System. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 177–184, July 2016.

[7] Y. Hou, Y. S. Ong, L. Feng, and J. M. Zurada. An Evolutionary Transfer Reinforcement Learning Framework for Multi-Agent System. *IEEE Transactions on Evolutionary Computation*, PP(99):1–1, 2017.

[8] P. Huo, S. C. K. Shiu, H. Wang, and B. Niu. Application and Comparison of Particle Swarm Optimization and Genetic Algorithm in Strategy Defense Game. In *2009 Fifth International Conference on Natural Computation*, volume 5, pages 387–392, Aug. 2009.

[9] Jonathan Nave; Shmuel Markovich. Artificial Intelligence Benchmark, 2002.

[10] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. Gamebots: a flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45, 2002.

[11] B. Li, S. Mabu, and K. Hirasawa. Tile-world #x2014; A case study of Genetic Network Programming with automatic program generation. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 2708–2715, Oct. 2010.

[12] M. Lloyd-Kelly, P. C. R. Lane, and F. Gobet. The Effects of Bounding Rationality on the Performance and Learning of CHREST Agents in Tileworld. In M. Bramer and M. Petridis, editors, *Research and Development in Intelligent Systems XXXI: Incorporating Applications and Innovations in Intelligent Systems XXII*, pages 149–162. Springer International Publishing, Cham, 2014. DOI: 10.1007/978-3-319-12069-0_10.

[13] Mark Wesley. Implementing a Rewindable Instant Replay System for Temporal Debugging.

[14] M. Molineaux and D. W. Aha. Tielt: A testbed for gaming environments. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 4*, AAAI'05, pages 1690–1691. AAAI Press, 2005.

[15] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas, and T. Schaul. General video game ai: Competition, challenges and opportunities. In *Proceedings of AAAI*, 2016.

[16] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C. U. Lim, and T. Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, Sept 2016.

[17] M. E. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1*, AAAI'90, pages 183–189, Boston, Massachusetts, 1990. AAAI Press.

[18] M. Ponsen. *Improving adaptive game AI with evolutionary learning*. PhD thesis, Citeseer, 2004.

[19] M. J. Ponsen, S. Lee-Urban, H. Muñoz-Avila, D. W. Aha, and M. Molineaux. Stratagus: An open-source game engine for research in real-time strategy games. *Reasoning, Representation, and Learning in Computer Games*, page 78, 2005.

[20] S. Rabin, editor. *Game AI pro 2: collected wisdom of game AI professionals*. CRC Press, Taylor & Francis Group, Boca Raton, 2015.

[21] I. Riot Games. Riot Games API.

[22] C. Scheepers and A. Engelbrecht. Training multi-agent teams from zero knowledge with the competitive coevolutionary team-based particle swarm optimiser. *Soft Computing*, pages 1–14, 2014.

[23] J. Stamford, A. S. Khuman, J. Carter, and S. Ahmadi. Pathfinding in partially explored games environments: The application of the A*; Algorithm with occupancy grids in Unity3d. In *2014 14th UK Workshop on Computational Intelligence (UKCI)*, pages 1–6, Sept. 2014.

[24] K. O. Stanley, B. D. Bryant, I. Karpov, and R. Miikkulainen. Real-time evolution of neural networks in the nero video game. In *AAAI*, volume 6, pages 1671–1674, 2006.

[25] StatCounter. StatCounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share.

[26] P. Stawarz and Z. Świder. Data-Driven Video Game Agent Pathfinding. In R. Szewczyk, C. Zieliński, and M. Kaliczyńska, editors, *Automation 2017*, volume 550, pages 308–318. Springer International Publishing, Cham, 2017. DOI: 10.1007/978-3-319-54042-9_28.

[27] SuperData. SuperData Research | Games data and market research ≫ Worldwide digital games market: January 2017.

[28] F. Tencé, C. Buche, P. D. Loor, and O. Marc. The Challenge of Believability in Video Games: Definitions, Agents Models and Imitation Learning. *CoRR*, abs/1009.0451, 2010.

[29] M. Weinberger. The 11 top-grossing video games of all time.

ABSTRACT

Competitive multiplayer video games take many forms. From short, slow-paced, strategic card games, to fast and realistic first-person shooters, all have one thing in common – the presence of agents controlled by an artificial intelligence. The human factor often requires the application designer to perform various tweaks, as human behavioral patterns are complicated and evolve based on the individual game knowledge. Adaptive artificial agents are undoubtedly the best solution, but call for prolonged testing. The article describes ways to implement determinism in an AI testing environment, vastly decreasing the required number of individual agent training sessions. In the analyzed scenario, testers participate in a set number of matches, facing only human opponents in the environment which keeps track of their movement and actions. Then an artificially controlled agent is introduced into the recorded sessions, and once again the statistics of each participant are analyzed to determine whether the AI can adequately adapt. In the case of failure, AI parameters are changed, and the process is repeated until the result is acceptable for the game designer. The environment has been designed to test a novel algorithm that has been proposed in the previous work and allows to change agent pathfinding to maximize the entities score (i.e., the number of kills), depending on current and previous player actions.

IMPLEMENTACJA DETERMINISTYCZNEGO ŚRODOWISKA TESTOWEGO AGENTÓW W GRACH KOMPUTEROWYCH

STRESZCZENIE

Wieloosobowe gry komputerowe w których gracze walczą przeciwko sobie przybierają wiele postaci. Począwszy od krótkich, powolnych, strategicznych gier karcianych, a kończąc na realistycznych i emocjonujących grach z widokiem pierwszoosobowym, wszystkie cechują się obecnością agentów kontrolowanych przez sztuczną inteligencję. Czynnik ludzki wymusza na projektantach poświęcenie sporej ilości czasu na dopracowanie agentów, ponieważ ludzkie decyzje bywają skomplikowane i zmieniają się w zależności od posiadanej wiedzy. Adaptujące się do bieżącej sytuacji agenty niezaprzeczalnie są najlepszym rozwiązaniem, ale wymagają długich i mozolnych testów. Artykuł opisuje sposób wprowadzenia determinizmu do środowiska testowego, co w efekcie drastycznie zmniejszy wymaganą liczbę sesji uczących. W analizowanym przykładzie testerzy uczestniczą w ustalonej liczbie rozgrywek przeciwko innym ludzkim przeciwnikom w środowisku, które nagrywa ich ruchy i inne akcje. Po zakończeniu pracy z testerami, do nagrań wprowadzany jest agent. Wpływ agenta na rozgrywkę jest analizowany by określić jego zdolności adaptacji. W przypadku gdy wyniki są niezadowalające, parametry agenta są zmieniane i proces jest powtarzany do momentu, w którym projektant uzna zachowanie SI za zadowalające. Ĺśrodowisko zostało opracowane w celu przetestowania nowatorskiego rozwiązania, które opracowaliśmy podczas wcześniejszych prac. Stworzony przez nas algorytm ma na celu maksymalizację wyniku agenta (rozumianego jako ilość zabić), w oparciu o obecne i historyczne decyzje graczy i polega na modyfikacji wag krawędzi grafu mapy w reakcji na ważne zdarzenia, które mają miejsce podczas rozgrywki.

Received: 2017-10-13
Accepted: 2017-12-06