Coordinating bots for automated game testing using deep learning

Ciprian Paduraru
Dept. of Computer Science
University of Bucharest
ciprian.paduraru@unibuc.ro

Miruna Paduraru
Dept. of Computer Science
University of Bucharest
miruna.paduraru@drd.unibuc.ro

Alin Stefanescu
Dept. of Computer Science
University of Bucharest
alin.stefanescu@unibuc.ro

***Abstract*—Video game development is a growing industry with important revenues nowadays. However, there is a negative trend when it comes to the quality of the released games, many of them including bugs or performance issues that affect the user experience. This paper presents the first prototype of a new open-source framework that automates game testing by coordinating bots. For that, we leverage previous work in the field of smart agents that play video games as well as deep learning methods that interpret the feedback of their actions from visuals and sounds.**

***Index Terms*—bots, game testing, automated testing, deep learning, reinforcement learning, software architecture**

I. Introduction

Video games are an important part of the entertainment industry [1], showing solid growth and large market size (e.g., in 2020, the revenues from video games surpassed those of the global movie and US sport markets combined). Due to constant changes in the source code to implement new features and pressure from tight release deadlines, many video games or updates are released with several bugs that decrease the overall experience of users [2]. While there are important resources allocated for testing of the games, most of the efforts are directed towards manual testing. This is certainly valuable for checking the game graphics or animation components, or gameplay interaction, but there are many opportunities for productivity and efficiency gains through test automation.

To this end, we had several unstructured interviews with development and QA managers from one of the top video game companies[1] and used our own previous experience in the game industry, to have a firsthand understanding of the current gaps in automating game testing. Our initial ideas and prototypes to address them using coordinated bots are presented in this paper.

The contributions and structure of the paper are the following: a short survey on bots that can be used for testing games, a presentation of state-of-the-practice gaps and issues for game testing, our proposed methods for filling these gaps, and implementation details of our prototype framework.

II. Related work

a) Bots used to simulate human-like behavior in games: Several papers described how to use bots that can play a game as close as possible to real humans. In general, users interact with games by executing a sequence of actions on observed scenes. Thus, it is natural to express their behavior using a Markov Decision Process (MDP). This is one of the reasons for studying methods for playing games with reinforcement learning (RL) techniques, as existing recent literature demonstrates. The authors of these previous works typically connect RL techniques to Monte Carlo Trees Search (MCTS). E.g., this is the case for [3], which demonstrates how $Sarsa(\lambda)$ RL-method is used for the classic game of Pac-Man. Similar works are [4] for Unreal Tournament, [5] for Super Mario using neuroevolution, [6] for a 3-match game, or GVG-AI [7] for competition agents.

Other works use the idea of penalizing agents that deviate too much from human behaviors through reward functions as presented in [8], [9], [4]. Instead of providing manually reward functions, to make sure that the learned policy mimics a human behavior, Inverse Reinforcement Learning [10] is used to extract the reward functions from real users sequences of actions. The papers presented above incorporate domain knowledge to speed-up the bots training and their quality, but there are also results showing that game bots can be trained only from images [11], [12], [13].

b) Bots used for testing: The work in [14] uses Inverse Reinforcement Learning to extract reward knowledge from human user trajectories, then it creates a generative test oracle that can produce similar kind of trajectories.

Testing UI interfaces for Windows 10 was studied in [15] using a combination of RL methods (Q-learning) and Graph Neural Networks (GNN) to represent the state of the application. Adventure-like game testing using 2D graphics using similar RL methods combined this time with memory was reported in ICARUS framework [16]. As noted in [17], the previous work in the literature was focused more on making better agents for game playing, rather than testing. However, their work uses RL and evolutionary algorithms to evaluate as many states of the game as possible for putting the game in various contexts. Continuing on the idea of generating tests using RL agents, the work in [18] extends the previous idea in 3D games and tries to create a coverage heatmap of situations analyzed at any time during the testing process.

Finally, another way to define agents [19] that can test games is described in Aplib [20], where the authors create

[1]The name of the company is not disclosed until a formal approval in this respect is obtained.

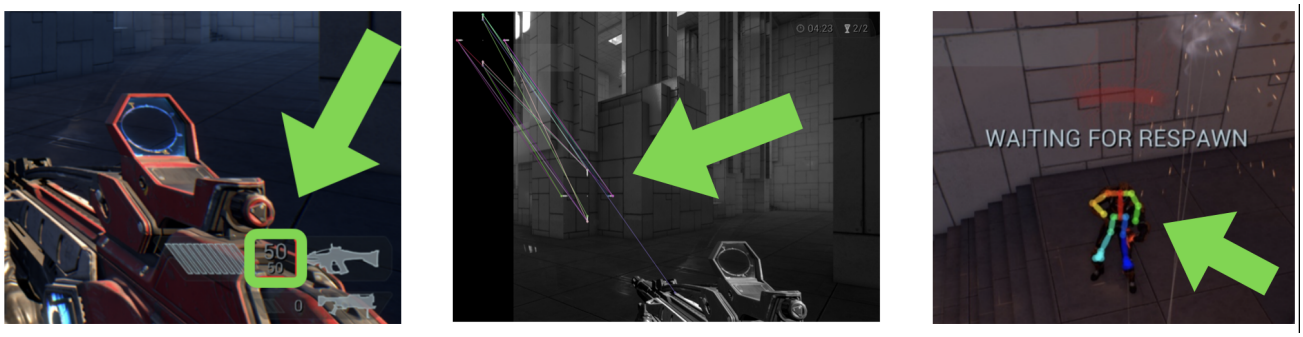

Fig. 1. Various visual elements that we target during test automation

a Domain Specific Language (DSL) composed of actions, goals, and conditions that define the agents' objectives and behaviours in the game and their expected actions. It can be seen as functional testing for games, written for games implemented in Java.

III. Motivation for our work and gaps in the automated game testing field

As summarized in the previous section, the literature concentrates on bots implementation with the purposes of either making better agents or game testing by improving the coverage of states in the game environments. Our work instead is focusing on a different topic: leveraging the existent work on bots creation and coordinating them in order to increase even further the game test automation. With this in mind, we propose a framework that uses modern deep learning techniques from computer vision or speech recognition to pair expected behaviours in the game with the visual/sound feedback produced by the game application.

Based on discussions with game industry partners, we compile below a few requirements, gaps, and aspects that take significant human effort during the manual tests performed by the QA department. We explain them through examples:

- **UI Testing**: If the user shoots someone, did the score increase on the Heads-Up Display (HUD)? After the game ended, did a certain menu appeared on the screen? Is the ammo displayed on the screen in sync with the value in the game memory? E.g., see a screenshot from our demo on the left of Fig. 1, where we check if the ammo displayed on the HUD at a given 2D bounding box (in that case, 50) is the same as the one expected and stored in the backend. If the user changed the weapon, is the cross icon on the screen positioned correctly? E.g., see a screenshot from our demo in the middle of Fig. 1, where we perform basic cross detection in our demo using simple visual feature matching methods with the classic OpenCV framework [21].
- **Animation testing**: The agent stays in place and watches an AI characteer with walk animation. Is it moving in the right direction over a sequence of N frames? E.g., see a screenshot from our demo in the right of Fig. 1, showing the skeleton of an enemy agent tracked using OpenPose framework [22].
- **Rendering testing**: Assume the user is being shot or in a low health condition. It is expected to see some post-processed effects on the screen. Are they visible? When using the binoculars item, is the camera centered correctly on the screen?
- **Physics**: A game agent could push an object over a sequence of frames. Does the collision system responds correctly?
- **Gameplay**: If a game agent is re-spawned on a map, does it respect a given set of spawning conditions? E.g., does it have a clear view and not in front of a wall or starting right away in front of an enemy? After an agent pressed the mapped button to enter in a car, does the visuals on the screen look like they are inside the car?
- **Sounds**: Was a specific sound played in the last N frames as requested?

All the above types of tests will be implemented by specialized testing bots as described in the following.

IV. Proposed methods and implementation

To validate that these game test bots can be coordinated to meet the automation requirements from the game development industry, we implemented a prototype framework that is open-sourced at https://github.com/unibuc-cs/game-testing. We prove its capabilities on a game demo running on Unreal Engine 4 [2], using our tool as a plugin over the official *Shooter Game* from Epic Marketplace [23]. We note that our framework has a plugin design architecture and can be reused with any other engine or game.

The architecture relies on two main entities:

- **GameBot** entity that plays the game in various environments/levels, with the purpose of testing various aspects. The $GameBot$ should be capable of registering mappings of $Actions$ to $ExpectedBehavior$ in a structured way (i.e., given a set of actions, what is their expected behaviors, either immediately or after a sequence of frames). We have implemented a first version (see details in the

[2]https://www.unrealengine.com

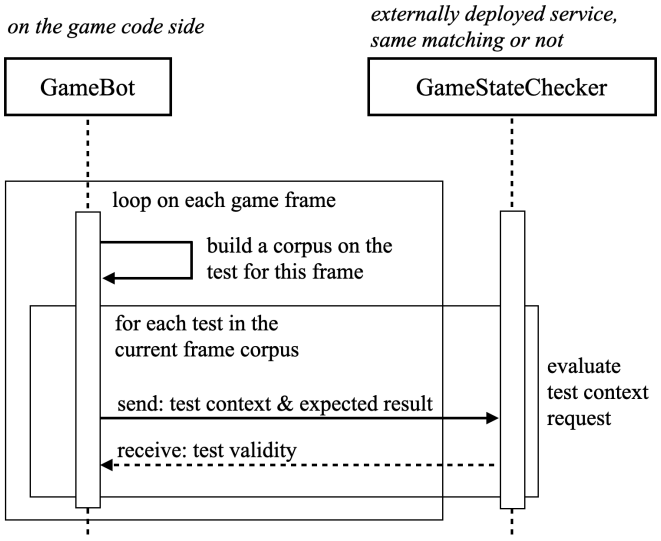

Fig. 2. Communication between the two main entities at runtime

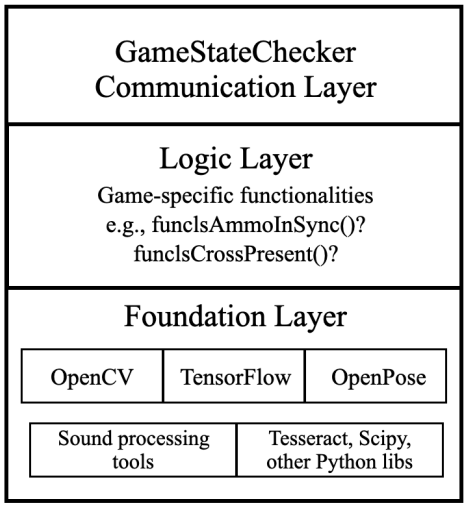

Fig. 3. The layered architecture of our testing framework

next subsections), but the format of this mapping is can be easily extended depending on the use case.

- **GameStateChecker** entity. Its purpose is to validate the expected behavior of the $GameAgent$ actions.

A. Detailed discussion about the two main entities

GameBot: This entity lies in the game development built application, as a plugin. For its implementation, we leverage existing work published in the literature, either using RL agents and their various combinations with other AI techniques, or common scripted bots that play given roles. These types of bots can use as inputs either images, internal game values, or developer's injected states. The reward signal is targeted for the purpose of testing. As an example, if the purpose is to test the $Physics$ component of the game, then the reward signal of the bot should be higher when triggering physical objects collisions, jumping, etc.

At runtime, the $GameBot$ can send to the $GameStateChecker$ entity the pairs of $Action$ and its $ExpectedBehavior$, with the meaning that it wants to test (either on the current frame of the game or in the sequence of the last N frames) if the requested actions produced the expected behavior. As shown on the diagram in Fig. 2, on each frame of the game, the $GameBot$ creates a corpus of tests to execute. Thus, on each frame, the framework builds pairs of:

- Action - represented by a dictionary of context variables that the game needs to pass to our tool to do the test properly, i.e., one or more screenshots representing a sequence, recorded game sounds, positions of various items on the screen, etc.
- Expected Behavior - represented by a dictionary describing the effect expected to happen, which must contain all the data necessary to quantify the results. For instance, it could contain a numeric value describing how much ammo should be visible as text on the screen.

GameStateChecker: This entity is implemented as an external service, decoupled from the game. It is currently implemented as a combination of Python and C++ code. The reason for having this component separated from the game is to leverage off-the-shelf state-of-the-art methods, open source-code as easily as possible using extensions at different layers, to speed-up development. When there is a need for prototyping or extending the framework with new techniques, the developer can use the layered architecture to their own advantage. Fig. 3 presents the architecture of the $GameStateChecker$, with the components summarized below:

- **Foundation layer** aggregates several models, some generic, other customized for the game itself. We would need a custom-per-game object because some games have customized effects. E.g., when a character is almost dying, trying to detect this generically would fail. The algorithm in the backend of our tool is the same, but the data on which the model is trained will be different.
- **Logic layer** handles services specific to a game, being based on the layer below it.
- **GameStateChecker Communication layer** is responsible for communication with the game side (i.e., $GameBot$ entity).

B. Communication of GameBot and GameStateChecker

This can be handled with different options. Currently, the $GameStateChecker$ component is deployed on a Flask/FlaskRest server. This allows RestAPI to be used as a communication and persistence model.

The communication with the $GameBot$ side component is more difficult since the set of allowed languages in modern high-performance game engine such as Unity or Unreal Engine 4 is limited to programming languages such as C++ or C#, and there is also the potential to have these games deployed

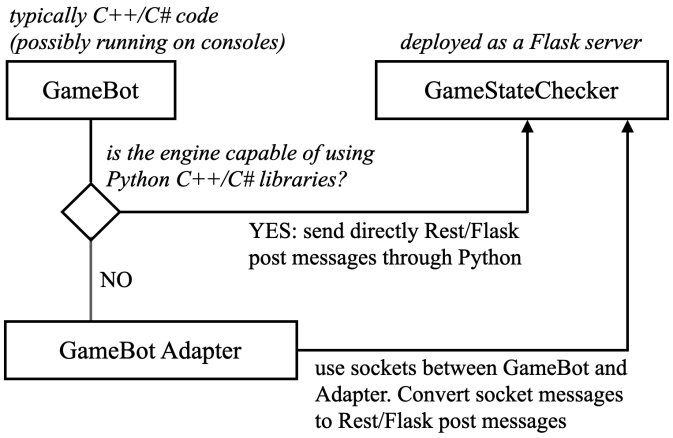

Fig. 4. The communication options between the $GameAgent$ component and the $GameStateChecker$

on embedded hardware such as game consoles or mobile devices, which limits even more the available languages and capabilities. Thus, the $GameBot$ can be implemented in two ways, see Fig. 4, each having tradeoffs:

- Method A: The C++/C# code of $GameBot$ is calling Python code directly to send messages to the $GameStateChecker$. This method could only work if the deployment of the game is not made using embedded deployment, but it provides easier customization and usage during the prototyping phase when using a PC development kit.
- Method B: The C++/C# code of $GameBot$ can send messages through sockets to an intermediate component $GameAgentAdapter$ that handles the communication between consoles and server PC which indeed uses method A. The role of the $Adapter$ instance then is to receive messages from a socket endpoint and convert them to Flask post messages. The $Adapter$ could be also implemented using higher-level libraries such as *libcurl* or *MicrosoftRestAPI*, but there is always a problem from our experience with building those on consoles. So sockets seem to be the most common way to do OS/hardware abstraction on this component.

C. Foundation layer technologies

As shown in Fig. 3, several external technologies are leveraged to do state checking inside the $GameStateChecker$ component. For text recognition, the framework uses Tesseract OCR from OpenCV [21]. For object recognition either template matching [21], or Yolo model [24], combined with scene segmentation methods such as [25], if needed. For the animation testing, we use OpenPose [22] method which applies computer vision techniques for identifying the skeletons of characters present on a given image, as shown on the right of Fig. 1. Using this method, our framework is able for example to track several characters at skeleton level during a sequence of frames to perform animation testing automatically. For sound testing (recognition and processing), our prototype framework uses Fairseq [26].

V. Future work

There are several directions to follow from the current state of our framework. We mention only a couple of ideas. For instance, we plan to extend the ability to find automatically interesting places for functional testing inside games by using additional information from the graphical blueprints [27] inside modern engines with state-of-the-art methods of model-based testing, symbolic execution, and fuzzing [28]. Also, we plan to invest efforts in making smart bots that play games either from computer vision or reinforcement learning. In this respect, we started experimenting with a special type of bots from the fast-growing domain of Robotic Process Automation (RPA) [29] as testing bots at the UI level of the game [30]. Last but not least, we will apply our tool to real games from the industry and, if possible, create a common dataset to evaluate the results of different methods for game testing.

Acknowledgments

This work was supported by a grant of Romanian Ministry of Research and Innovation UEFISCDI no. 401PED/2020.

References

[1] Grand View Research, "Video game market size and forecasts, 2020 - 2027," Market research report, no. GVR-4-68038-527-4, 2020.
[2] R. E. S. Santos, C. V. C. Magalhães, L. F. Capretz, J. S. Correia-Neto, F. Q. B. da Silva, and A. Saher, "Computer games are serious business and so is their quality: Particularities of software testing in game development from the perspective of practitioners," in *Proc. of ESEM'18*. ACM, 2018, pp. 1–10.
[3] N. Tziortziotis, K. Tziortziotis, and K. Blekas, "Play ms. Pac-Man using an advanced reinforcement learning agent," in *Artificial Intelligence: Methods and Applications - 8th Hellenic Conference on AI, SETN 2014*, ser. LNCS, vol. 8445. Springer, 2014, pp. 71–83.
[4] F. G. Glavin and M. G. Madden, "Adaptive shooting for bots in first person shooter games using reinforcement learning," *IEEE Trans. Comput. Intell. AI Games*, vol. 7, no. 2, pp. 180–192, 2015.
[5] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis, "Imitating human playing styles in Super Mario Bros," *Entertain. Comput.*, vol. 4, no. 2, pp. 93–104, 2013.
[6] N. Napolitano, "Testing match-3 video games with deep reinforcement learning," *ArXiv*, vol. abs/2007.01137, 2020.
[7] A. Khalifa, A. Isaksen, J. Togelius, and A. Nealen, "Modifying MCTS for human-like general video game playing," in *Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI'16)*. AAAI, 2016, p. 2514–2520.
[8] S. F. Gudmundsson *et al.*, "Human-like playtesting with deep learning," in *IEEE Conf. on Computational Intelligence and Games (CIG'18)*. IEEE, 2018, pp. 1–8.
[9] B. Tastan and G. Sukthankar, "Learning policies for first person shooter games using inverse reinforcement learning," in *Proc. of AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'11)*. AAAI, 2011, p. 85–90.
[10] A. Sosic, E. Rueckert, J. Peters, A. M. Zoubir, and H. Koeppl, "Inverse reinforcement learning via nonparametric spatio-temporal subgoal modeling," *J. Mach. Learn. Res.*, vol. 19, pp. 69:1–69:45, 2018.
[11] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.
[12] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.
[13] O. Vinyals *et al.*, "AlphaStar: Mastering the real-time strategy game StarCraft II," https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii, 2019.
[14] S. Ariyurek, A. Betin-Can, and E. Surer, "Automated video game testing using synthetic and human-like agents," *IEEE Transactions on Games*, pp. 1–1, 2019.

[15] L. Harries *et al.*, "DRIFT: deep reinforcement learning for functional software testing," 2020. [Online]. Available: https://arxiv.org/abs/2007.08220

[16] J. Pfau, J. D. Smeddinck, and R. Malaka, "Automated game testing with ICARUS: intelligent completion of adventure riddles via unsupervised solving," in *Extended Abstracts Publication of CHI PLAY 2017*. ACM, 2017, pp. 153–164.

[17] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'19)*, 2019, pp. 772–784.

[18] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," in *IEEE Conf. on Games (CoG'20)*, 2020, pp. 600–603.

[19] E. Enoiu and M. Frasheri, "Test agents: The next generation of test cases," in *NEXTA'19 Workshop affiliated with ICST'19*. IEEE, 2019, pp. 305–308.

[20] I. S. W. B. Prasetya and M. Dastani, "Aplib: An agent programming library for testing games," in *Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS'20)*, 2020, pp. 1972–1974.

[21] "The OpenCV Library." [Online]. Available: https://opencv.org

[22] Z. Cao, G. Hidalgo, T. Simon, S. Wei, and Y. Sheikh, "OpenPose: Realtime multi-person 2D pose estimation using part affinity fields," pp. 172–186, 2021.

[23] Epic Games, "Shooter game example." [Online]. Available: https://docs.unrealengine.com/en-US/Resources/SampleGames/ShooterGame

[24] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. IEEE, 2016, pp. 779–788.

[25] L. Porzi, S. R. Bulo, A. Colovic, and P. Kontschieder, "Seamless scene segmentation," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR'19)*, 2019, pp. 8277–8286.

[26] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proc. of NAACL-HLT 2019: Demonstrations*, 2019, pp. 48–53.

[27] M. Romero and B. Sewell, *Blueprints Visual Scripting for Unreal Engine: The faster way to build games using UE4 Blueprints, 2nd ed.* Packt Publ., 2019.

[28] C. Paduraru, M. Paduraru, and A. Stefanescu, "RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing," in *IEEE Int. Conf. on Software Testing, Verification and Validation 2021 (ICST'21)*. IEEE, 2021, p. to appear.

[29] W. van der Aalst, M. Bichler, and A. Heinzl, "Robotic process automation," *Business & Information Syst. Eng.*, vol. 60, no. 4, pp. 269–272, 2018.

[30] M. Cernat, A.-N. Staicu, and A. Stefanescu, "Improving UI test automation using robotic process automation," in *15th Int. Conf on Software Technologies (ICSOFT'20)*. SciTePress, 2020, pp. 260–267.