Automated game testing using computer vision methods

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

Miruna Paduraru
Ubisoft Romania &
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 nowadays with high revenues. However, even if there are many resources invested in the software development process, many games still contain bugs or performance issues that affect the user experience. This paper presents ideas on how computer vision methods can be used to automate the process of game testing. The goal is to replace the parts of the testing process that require human users (testers) with machines as much as possible, in order to reduce costs and perform more tests in less time by scaling with hardware resources. The focus is on solving existing real-world problems that have emerged from several discussions with industry partners. We base our methods on previous work in this area using intelligent agents playing video games and deep learning methods that interpret feedback from their actions based on visual output. The paper proposes several methods and a set of open-source tools, independent of the operating system or deployment platform, to evaluate the efficiency of the presented methods.**

***Index Terms*—AI agents, 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 game testing, 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.

Overall, the main contribution of the paper is to provide ideas of improving the current state of automated game testing in video games using computer vision techniques. To this end, the final products could become more stable, more fun to play, and the cost of human-performed testing could be reduced. We break down our novel contributions in the following:

- A list of current problems identified with game industry practitioners regarding the difficulty of applying automated testing to their products.
- An analysis of computer vision techniques that can be used to mimic the existence of a human user behind the testing process.
- A reusable prototype framework architecture and implementation (to the authors' knowledge, the first in the field at the moment of writing) that is independent of the engine and deployment platform (i.e., it works on different devices such as PCs, game consoles, or different operating systems) and that allows users to directly re-use existing open source code of computer vision libraries. The strategies used at the implementation level are indeed novelties, because at the technical implementation level, many commercial public game engines (e.g., Unity, Unreal, etc.) have their own solutions for performing unit or functional tests, but they differ and generally do not try to solve the low-level problem of replacing the human user performing the tests with an AI agent analysing the visual state of the running game, which we discuss in this paper. Many developers have their own game engine or want to switch between public engines without having to change the test code at all.

Our prototype framework is available as open-source at https://github.com/unibuc-cs/game-testing. We are evaluating its capabilities on the game engine Unreal Engine 4 [2] by deploying our tool as a plugin via the official demos *Shooter Game* and *Car Configurator* from Epic Marketplace [3], and an open source 3D tank game based on Unity https://github.com/AGAPIA/BTreeGeneticFramework [4]. We note that our framework has a plugin architecture and can be reused with any other engine or game.

The paper is organized as follows. The next section describes related work in the field of game development and testing processes. Some missing gaps and difficulties in au-

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

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

tomating the testing process after discussions with our industry partners are outlined in Section III. The proposed computer vision methods are defined in Section IV. Architectural and technical implementation details are presented in Section V. Finally, plans for future work are presented in the last section.

II. Related work

a) AI agents 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 [5], which demonstrates how $Sarsa(\lambda)$ RL-method is used for the classic game of Pac-Man. Similar works are [6] for Unreal Tournament, [7] for Super Mario using neuroevolution, [8] for a 3-match game, or GVG-AI [9] for competition agents.

Other works use the idea of penalizing agents that deviate too much from human behaviors through reward functions as presented in [10], [11], [6]. Instead of providing manually reward functions, to make sure that the learned policy mimics a human behavior, Inverse Reinforcement Learning [12] 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 [13], [14], [15].

b) AI agents used for testing: The work in [16] 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 [17] 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 [18]. As noted in [19], 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 [20] 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 [21] that can test games is described in Aplib [22], where the authors create 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

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 [23].
- **Animation testing**: The agent stays in place and watches an AI character 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 [24].
- **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?

All of these types of tests are currently performed by human users (testers) who analyze the visual feedback of the game. This process is expensive and often not enough testing can be done before products are released. Our motivation is to automate parts of this process as much as possible, reduce the costs, and scale it with hardware resources that are generally easier to get than human resources.

IV. Proposed methods

As summarized in Section II, the literature focuses on implementing game agents with the goal of either achieving better results than humans or testing games by improving the coverage of states in game environments. Our work instead

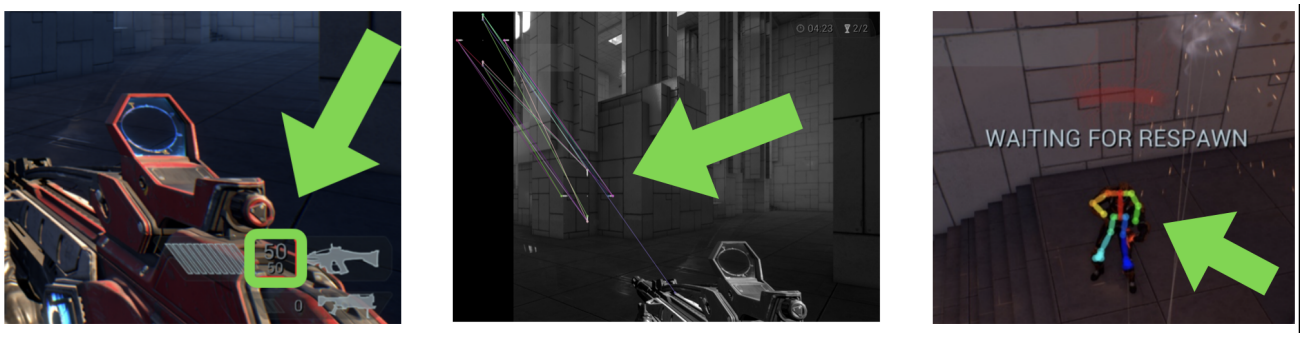

Fig. 1: Various visual elements checked automatically after triggering different tests in the official $ShootingGame$ demo from $Unreal\,4$ game engine. For example, in the left figure the game testing agent used the weapon to fire a fixed number of rounds. The test checks if the visual text representing the new ammo amount is correct. In the middle picture, the game testing agent triggered a weapon change test and checks visually if the weapon-cross icon on the screen is the correct one for the new weapon selected. In the right picture, we present a failed test after the game testing agent was killed. While waiting for respawning in the game, as a time limited penalization for being killed, the skeleton of the AI agent's character should have been grounded but it is up on its legs.

focuses on a different topic: leveraging existing work on creating and coordinating test agents to further improve the automation of game testing. To this end, we propose a set of methods and a framework that uses modern computer vision techniques to associate expected in-game behaviors with the visual feedback that the game application actually produces. Our idea is to use automated scripts or AI agents that play a game (which we refer to hereafter as *game testing agents*), and then use both the output visual images and (optionally) the internal game state to associate the specified expected behavior with the action performed by the game testing agent in specific states of the game.

A. Tests specification and execution

Tests are specified as a set of elements that map the game state and action to be executed in that particular state, to the expected behavior that a human user would likely see either immediately or after a series of frames. Formally, we refer to this test set as T. The elements of this set are of type: $[(State, Action) \rightarrow ExpectedBehavior]$. The $State$ represents a union of the internal game state representation for one or more frames of the game. The $Action$ is represented as a one-hot encoding representing the action type code name (e.g., jumping, moving, firing etc.) plus a dynamic array of float numbers describing the action details. The specification of the test is independent of the game mechanics, engine, or other technical implementation details.

Considering a game test agent exploring the application (regardless of the method, e.g., scripted, classical AI, RL agents etc.), the runtime component of the game checks the internal game state against the states specified in the test set T and creates a compatible sub-set for each frame: T_{compat}. Thus, any of $test \in T_{compat}$ can be executed at that time. If $test$ is selected for execution, then the game testing agent will execute the specified action $test_{Action}$. For this test to be considered correct, the output of the game must match the conditions specified in $test_{ExpectedBehavior}$. By output we mean the union of the *visual rendered feedback* and the internal game state changes after the action has been executed in the environment. The user is free to decide, for each test, whether to use only the visual output, only the changed internal game state representation, or both. From our experience, considering both at the same time produces the best results, as the visual output can be matched with the new internal game state after the action is executed.

Our strategy is to prioritize these tests using a custom tool that allows the user to select the priority of each test. This is important because users can target the tests to specific areas of the game where issues are common or that may be affected by the newly added code. Priorities can be adjusted dynamically, even at runtime. Since there needs to be a mapping from the game mechanics to the abstract representation of a test specification that we use in the framework, developers could promote fast test specifications writing by building a visual tool interface.

B. Methods used

Various external technologies based on computer vision are used to test the expected behaviors within our framework. Depending on the purpose of the user tests, the technologies currently used can be replaced or new ones can be added, similar to a plugin architecture. Below we outline the methods used to cover the testing requirements for covering the tests proposed in Section III.

Currently, the framework uses Tesseract OCR from OpenCV [23] for text recognition. For testing purposes where the presence of objects or specific features needs to be located or proven in the output image recognition, either template matching [23], the Yolo model [25] or scene segmentation methods such as [26] are used. For animation testing, we use the OpenPose method [24], which applies computer vision techniques to identify the skeletons of characters in a given

image. Using this method, our system is able to track multiple characters at skeleton level from a sequence of images to perform animation tests automatically. An example of how these methods and tools are used to verify the correctness of the tests in a demo game can be seen in Fig. 1.

For detecting changes in the environment (e.g., the test wants to check if the environment has actually changed after pressing a button to get into a vehicle after a certain time), we re-trained the [25] model with specific object classes (visual features) for each individual environment. An example is shown in Fig. 2. During inference, when the test wants to evaluate whether the displayed image is specific to that particular environment, the model returns the bounding boxes of the detected objects. If the number of detections is higher than a fixed threshold, it means that the environment change was successful.

To cover tests that need to analyze the movement of objects based on visual output (e.g., to confirm that a physical object has been moved or that an entity is moving in a certain direction), a simplification arises from the fact that human users (testers) see and interpret the sequence of frames in 2D space, not in 3D as the game state is internally represented in the general case.

Technically, we perform the following steps to test the correctness of the movement of objects:

1) For each scene that needs this type of testing, we take four points of static objects that are rendered on the screen. For these points, we have both the ground truth location in 3D space from the game state representation (source localization S_{loc}) and the position on the 2D output image where these points are rendered on the 2D visual image output space (target localization D_{loc}). A homograph matrix transformation [27] H is constructed that maps the points from S_{loc} to D_{loc}. Using the inverse of this matrix H^{-1}, arbitrary points from the 2D space of the visually rendered image can be projected back into 3D space (with some noise).
2) The 2D bounding box coordinates of the objects of interest in the test scenario are obtained using the Yolo method [25], Obj_i^{2Dbbox}.
3) The coordinates of the bounding boxes are projected back into 3D space using the matrix constructed in Step 1: $Obj_i^{3Dbbox} = H^{-1}(Obj_i^{2Dbbox})$.
4) Motion analysis is then performed in this reconstructed space to verify that the visual output matches the internal game state and the desired behavior. One such example is shown in Fig. 3.

V. Architectural and implementation level

The architecture of our framework 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 tests, i.e., mappings of $(State,\ Actions)$ 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 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 agents that play given roles. These types of agents 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 $(State,\ Action)$ 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. 4, on each frame of the game, the $GameBot$ creates a corpus of tests to execute. Thus, on each frame, the framework builds pairs of:

- (State, 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, 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. 5 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

Fig. 2: The image shows the detected visual features (bounding boxes) after a change of environment, expecting to be in a vehicle after the game testing agent triggers an action key. The detected feature classes are: mirrors, steering wheel, and navigation. The number of detection thresholds used in this particular case is two. The demo is based on the assets from Unreal 4's $CarConfigurator$ demo. We used different camera angles to also test whether or not the newly trained recognition model overfits and its performance is affected or not (Section IV-B).

Fig. 3: The image shows the detected bounding boxes of the objects of interest and the trajectory of the projectile in a scene from the Unity demo [4]. As mentioned in Section IV-B, the purpose is to visually verify the correctness of the projectile movement from source to target. Although the perspective looks isometric, it should be noted that the representation of the game state representation and the rendering are fully 3D.

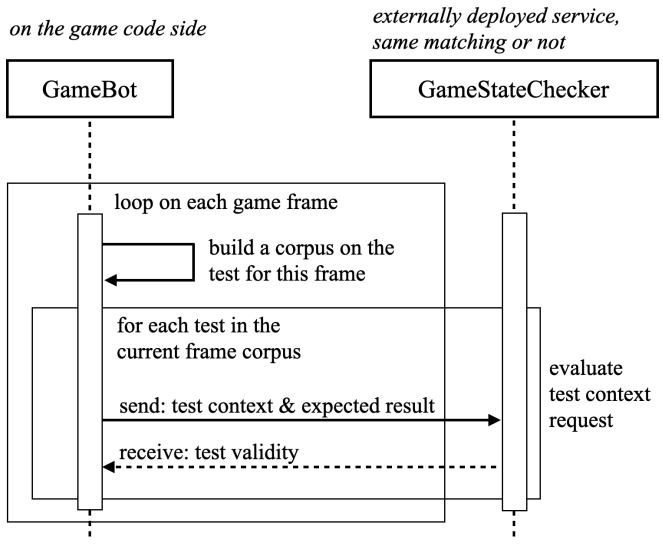

Fig. 4: Communication between the two main entities at runtime

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).

The $Foundation$ layer of the $GameStateChecker$ component, as shown in Fig. 5, hides the (user extensible) set of external tools used as core machine learning mechanisms for the testing purposes, as defined in Section IV-B.

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#,

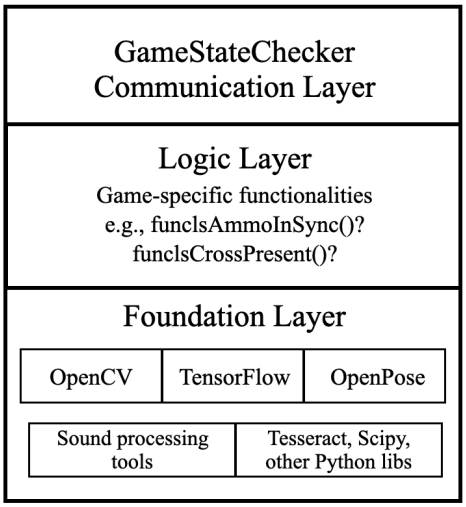

Fig. 5: The layered architecture of our testing framework

and there is also the potential to have these games deployed 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. 6, 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, from our experience, there is always a problem with building those on consoles. Therefore sockets seem to be the most common way to do OS/hardware abstraction on this component.

VI. Evaluation

The goal of the evaluation is not to show the result of code or test coverage, as this is independent of the methods described in this document. Indeed, the metrics for code or test coverage can be addressed by the behavior of the AI or scripted agents playing and testing the game, and by the variety of tests and actions specified. Instead, our main goal is to evaluate the quality of visual interpretation, i.e., what happens when human testers are replaced by our framework in different situations. As a secondary purpose, we also examined how expensive this kind of automatic visual verification of results is.

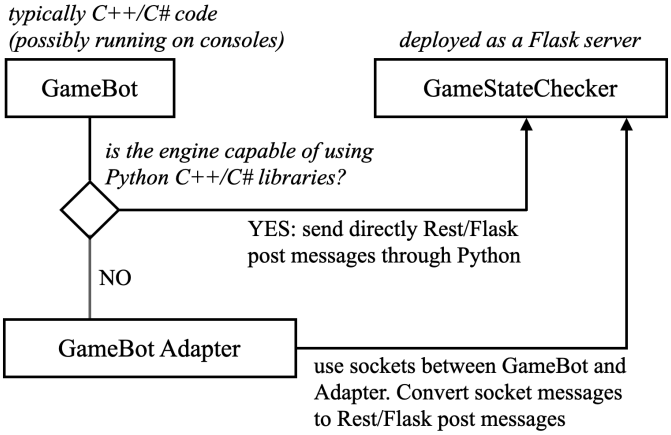

Fig. 6: The communication options between the $GameAgent$ component and the $GameStateChecker$

A. Interpretation of visual results evaluation

For the UI text and the detection of simple visual features recognition, such as the weapon cross tests specified in Sectioned IV-B, we had the automatic test agent run 1000 tests for weapon changes (between 8 models) at different times in the game, and the same number of tests with shot triggers with a number of bullets fired at each event, modeled by a Gaussian distribution, $bullets \sim \mathcal{N}(20, 5)$. The output text indicating the number of bullets remaining bullets was intentionally incorrect 50% of the time to mimic failed tests. The same percentage was used for the weapon cross, i.e., in half of the cases the visual output contained an incorrect weapon cross (e.g., pistol instead of shotgun). The accuracy for text recognition was 95.6%, and for the weapon cross 88.9%. To achieve these results, we also applied some post-processing techniques to the rendered images, such as conversion from RGB to HSV space, computed edge detection, and smoothing, all using standard features provided by OpenCV. This post-processing mechanism could be further improved to increase the accuracy of the results. The errors were mainly observed when the scene contained too many visual effects in the tested frame, e.g., when a user was hit or in a certain state that was displayed with high intensity.

For complex visual recognition with object detection and deep learning (e.g., for environment identification), we used the model trained in the $CarConfigurator$ demo. The same number of tests, 1000, was used. In half of the cases, the test agent was intentionally unable to enter the car even if the button was pressed, in the other half we mimicked a good test result (Fig. 2). In the interior, we used different camera angles and illumination models or intensities to ensure that the model was not overfitted on certain features. In this case, the accuracy was 100% in both cases. An interesting further evaluation could be to test between entering different vehicles and set a different class for each (instead of a binary decision as we have now) and then test whether or not the test agent is detected as entering the predicted vehicle class.

In the case of motion checking accuracy tests, we used the

$Tanks$ demo to fire projectiles between enemies and then verify that the visual representation of the fired projectile in each test confirms the internal game state in terms of the direction the projectile is flying or positioned in the world during a sequence of $N = 180$ frames (3 seconds, or less if the projectile is destroyed beforehand). The evaluation of accuracy yielded a precision of 89.7% for 1000 projectile tests. The accuracy threshold angle used, i.e., the maximum allowable difference between the trajectory in the game state and that reported by the visual output, was $T = 10\,deg$ to address small numerical precision errors in the reconstruction when computing the transformations mentioned in Section IV-B. The methods could be improved in further work. According to our observations, the accuracy is mainly affected by occluding objects and visual artifacts that obscure the projectile in some frames (e.g., particles triggered by dust, weapons, and other events). A better evaluation of motion control detection could also be considered by performing targeting at different angles and checking whether the visuals detect the correct angle (binned in some value ranges).

The accuracies obtained prove that an automatic agent that checks the visual results can be used with success to identify most cases. We also note that human testers are also prone to errors due to various physical factors such as composure, stress, visual occlusion or attention problems, etc. In future, more tests could be conducted to do random sampling and compare the accuracies achieved by human testers and computerized testers.

B. Evaluating Performance Impact

In the computer game industry, developers are always concerned with the overhead created by newly added methods. First of all, it should be clarified that there is no performance impact when the proposed testing methods are not enabled. In the development or testing phase, when the test methods are enabled, there are two main issues related to the performance topic and our addressed methods: (a) How much additional memory is required in the game to support the test procedures? (b) How is the execution speed affected at runtime (affecting the framerate)?

On the game side, there is no memory overhead because, as mentioned in Section V, the computer vision models are used in the external test process. When the test process is enabled, the game side is affected in execution time by two mechanisms: (1) packing a sequence of frames and metadata and then sending it to the external test process, (2) waiting for feedback after a particular method has been sent for evaluation. In (1), packing and sending high resolution frames was the first bottleneck encountered. Therefore, we tried switching from the full HD resolution of 1920x1080 to a lower resolution, 640x480, and re-assessing the accuracy of our methods. The results were surprisingly good even at the lower resolution, with the accuracy metrics evaluated in Section VI-A degrading by only 1.2%. Of course, these results may vary depending on game type, render quality, etc. On a CPU running 8-core AMD Ryzen 5800x, the average execution time for UI text recognition was $\sim 7.42ms$ (milliseconds), for UI feature recognition using classical OpenCV recognition methods $\sim 12.1ms$, while for feature-object recognition using the Yolo model we obtained on average $\sim 19.37ms$ for a single analyzed image. Obviously, when analyzing a sequence of images simultaneously, this could become a bottleneck. Therefore, a pipelined mechanism that performs detection over many frames is recommended, if feasible on the testing process side, to avoid peaks in the client game waiting too long for results.

VII. Future work

There are several directions to follow from the current state of our framework. First, we want to conduct further evaluations, especially for specific game genres (e.g., strategy, sports, etc.) to see if there are any particularities between them with respect to our methods. Another important issue we want to address is improving the accuracy of object detection and motion checking detection when many rendering artifacts are presented on the screen simultaneously. A simple example of a difficult situation is a scene with dust particles from a slow-moving tank that are hard to distinguish and may decrease the accuracy of the computer vision detection.

Apart from the methods presented in this paper, in general we would like to extend the capabilities of automated testing process in the field of game development. 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 [28] inside modern engines with state-of-the-art methods of model-based testing, symbolic execution, and fuzzing [29]. Also, we plan to invest more efforts in making smart agents that play games using computer vision or reinforcement learning. In this respect, we started experimenting with a special type of agents from the fast-growing domain of Robotic Process Automation (RPA) [30] as testing agents at the UI level of the game [31]. We are also working on generalizability, e.g., how could our approach be used for actions whose expected outcome depends on various parameters. 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] Epic Games, "Shooter game example." [Online]. Available: https://docs.unrealengine.com/en-US/Resources/SampleGames/ShooterGame

[4] C. Paduraru and M. Paduraru, "Automatic difficulty management and testing in games using a framework based on behavior trees and genetic algorithms," in *Proc. of 24th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'19)*. IEEE, 2019, pp. 170–179.

[5] 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.

[6] 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.

[7] 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.

[8] N. Napolitano, "Testing match-3 video games with deep reinforcement learning," *ArXiv*, vol. abs/2007.01137, 2020.

[9] 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.

[10] S. F. Gudmundsson *et al.*, "Human-like playtesting with deep learning," in *Proc. of IEEE Conf. on Computational Intelligence and Games (CIG'18)*. IEEE, 2018, pp. 1–8.

[11] 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, pp. 85—90.

[12] 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.

[13] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[14] 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.

[15] 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.

[16] S. Ariyurek, A. Betin-Can, and E. Surer, "Automated video game testing using synthetic and humanlike agents," *IEEE Transactions on Games*, vol. 13, no. 1, pp. 50–67, 2021.

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

[18] 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.

[19] 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 *Proc. of IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'19)*, 2019, pp. 772–784.

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

[21] 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.

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

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

[24] 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.

[25] 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.

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

[27] D. Baráth and L. Hajder, "Novel ways to estimate homography from local affine transformations," in *Proc. of the 11th Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP'16)*. SciTePress, 2016, pp. 434–445.

[28] 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.

[29] C. Paduraru, M. Paduraru, and A. Stefanescu, "RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing," in *Proc. of IEEE Int. Conf. on Software Testing, Verification and Validation 2021 (ICST'21)*. IEEE, 2021, pp. 430–435.

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

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