RiverGame - a game testing tool using artificial intelligence

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

Miruna Paduraru
Electronic Arts &
Dept. of Computer Science
University of Bucharest, Romania
miruna.paduraru@drd.unibuc.ro

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

2022 IEEE Conference on Software Testing, Verification and Validation (ICST) | 978-1-6654-6679-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICST53961.2022.00048

***Abstract*—As is the case with any very complex and interactive software, many video games are released with various minor or major issues that can potentially affect the user experience, cause security issues for players, or exploit the companies that deliver the products. To test their games, companies invest important resources in quality assurance personnel who usually perform the testing mostly manually. The main goal of our work is to automate various parts of the testing process that involve human users (testers) and thus to reduce costs and run more tests in less time. The secondary goal is to provide mechanisms to make test specification writing easier and more efficient. We focus on solving initial real-world problems that have emerged from several discussions with industry partners. In this paper, we present RiverGame, a tool that allows game developers to automatically test their products from different points of view: the rendered output, the sound played by the game, the animation and movement of the entities, the performance and various statistical analyses. We also address the problem of input priorities, scheduling, and directing the testing effort towards custom and dynamic directions. At the core of our methods, we use state-of-the-art artificial intelligence methods for analysis and a behavior-driven development (BDD) methodology for test specifications. Our technical solution is open-source, independent of game engine, platform, and programming language.**

***Index Terms*—game testing, automated testing, BDD, deep learning, reinforcement learning, computer vision**

I. Introduction

The video game segment is one of the leaders in the entertainment industry by looking at the revenues obtained in recent years [1]. The diversity of devices on which games need to be deployed, the continuously increasing complexity of game engines and internal tools, put several challenges [2] for developers to deliver quality products within tight release deadlines.

In this context, testing is a major activity in the game development process, with big companies in the game industry employing hundreds of testers for their flagship products. However, most of the resources are directed towards manual testing and this is understandable given the high complexity and interactivity of the games. Nonetheless, talking to several industrial partners in the domain and from our own experience in industry, we identified plentiful unexplored opportunities for test automation using the latest methods and advancements in artificial intelligence.

In this paper we report on several novel contributions that we implemented in our RiverGame testing tool:

- Identification of several aspects that are usually not automated during game testing and several ideas on how to address them.
- A reusable 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 programming language.
- A methodology that lets non-technical stakeholders write tests and expected behaviors for the features of the game under test. For this, the framework uses the Behavior-Driven Development (BDD) methodology and implements the separation of concerns concepts. This allows to use different programming languages and to be agnostic to the deployment platform.
- A scheduling system for tests that is automatically managing in the background their execution order depending on the availability and the desired rate of execution between them. Thus, by selecting different priorities, the user is guiding the process of tests execution through different areas of the game under test.
- Sound processing techniques that automatically evaluate the sounds being played during the game without human involvement.
- Automatic evaluation of statistical metrics registered by the end-user. The purpose of this is to test statistically metrics such as frames per second, memory footprints, quality of different gameplay systems over the entire game sessions or a time-limited period.
- Improvement of the performance of the evaluation processes by using different methods for the pose recognition and objects detection, replacing existing models based on Yolo [3] with EfficientNet [4] and OpenPose [5] with MoveNet [6].

  The strategies used at the implementation level are indeed novel, because at the technical level many commercial public game engines (e.g., Unity, Unreal, etc.) have their own solutions for performing unit or functional tests, but

978-1-6654-6679-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ICST53961.2022.00048

they differ and generally do not try to solve the 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.

We evaluated the capabilities of our tool on the game engine Unreal Engine 4[1] by deploying our tool as a plugin via the official demos *Shooter Game* and *Car Configurator* from Epic Marketplace [7], an open-source 3D tank game based on Unity https://github.com/AGAPIA/BTreeGeneticFramework [8], and two Electronic Arts well-known games, *Start Wars Rogue Squadron* and *FIFA 22*. As an important observation, our framework has a plugin architecture and can be reused with any other engine or game.

Note that we previously published a position paper about the first initial version of the tool in [9]. There are major differences between our tool paper and the previous workshop position paper, which only presented a preliminary architecture and the ideas that we planned to pursue. Most of the material presented in this paper is new.

Our paper is organized as follows. The next section describes related work. Gaps and difficulties in automating the testing process are outlined in Section III. The proposed methods are defined in Section IV. Architectural and technical implementation details are presented in Section V. Evaluation is described in Section VI and lessons learned in the last section.

Our prototype framework is available as open-source at: https://github.com/unibuc-cs/game-testing.

The demo video of our tool can be viewed online here: https://youtu.be/qFfWvaLtOU0.

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, one abstract way to define their behavior is to use 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 [10], which demonstrates how $Sarsa(\lambda)$ RL-method is used for the classic game of Pac-Man. Similar works are [11] for Unreal Tournament, [12] for Super Mario using neuroevolution, [13] for a 3-match game, or GVG-AI [14] for competition agents.

Other works use the idea of penalizing agents that deviate too much from human behaviors through reward functions as presented in [15], [16], [11]. Instead of providing manually reward functions, to make sure that the learned policy mimics human behavior, Inverse Reinforcement Learning [17] 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 [18], [19], [20].

b) AI agents used for testing: The work in [21] uses Inverse Reinforcement Learning to extract reward knowledge from human user trajectories, then it creates a generative test oracle that can produce similar kinds of trajectories.

Testing UI interfaces for Windows 10 was studied in [22] 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 [23]. As noted in [24], 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 [25] 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 [26] that can test games is described in Aplib [27], 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 Head 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 [28].
- **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

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

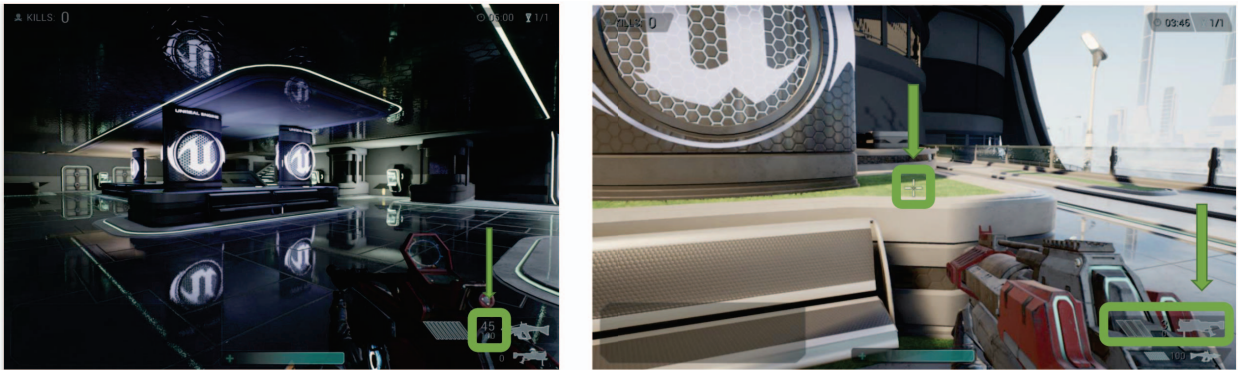

Fig. 1: Visual elements checked automatically after triggering different tests in the official $ShootingGame$ demo from $Unreal\,4$ game engine. For example, in the left figure, the game testing agent used the weapon to fire a fixed number of rounds. The test checks if the visual text representing the new ammo amount is correct. In the right 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 and the camera's zoom level. The new weapon selection should correspond to the icon in the bottom right of the screen.

screenshot from our demo in Fig. 6, showing the skeleton of an enemy agent tracked using MoveNet [6].

- **Sounds**: Was a specific sound played in the last N frames as requested? The problems reported usually by industry partners in this area are sound assets that are not playing at all or are interrupted by other sounds without a correct reason.
- **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 respond 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 a car, do 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 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 with the following features:

- Automatically test the rendered output content of the game using computer vision techniques to associate the expected behavior in the game with the visual feedback that the game application actually generates. 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 visual output images and (optionally) the internal game state to link the specified expected behavior to the action that the game testing agent takes in certain states of the game.
- Automatic sound testing to understand whether the game's sound feedback is correct, rather than relying on human efforts to evaluate it; the methods used imply both classical and natural language processing (NLP) methods for analyzing the sound content or text behind the played voices.
- Automatically monitor and report certain metrics within the game, such as frames per second (FPS), memory footprint in different scenarios, or custom registered metrics by the developers.

A. Tests description methodology

Behavior-driven development (BDD) [29] is used in software development projects to promote collaboration between different stakeholders such as developers, QA, and non-technical or business participants. It is typically used within an agile development methodology. BDD is also successfully used for testing purposes, as evidenced by the literature [30] and the variety of open-source or commercial solutions that use it. For this project, we used the $Behave$ library[2].

In software testing, BDD extends the test-driven development by allowing both developers and non-programmers (e.g., quality assurance members on a team)

[2]https://github.com/behave/behave

to use natural language combined with the language of domain-driven design to describe the purpose of features within a software project. Using this technique, both technical and non-technical stakeholders can discuss and understand the features and expected outcomes in specific scenarios. The natural language that describes the scenarios is called *Gherkin* [31]. An example of a test written in this language and methodology can be found in Fig. 2. The motivation for using BDD and natural language to write tests in game programming comes from two directions: (a) Many of the QA staff have no programming experience, so having a natural language with a guiding library of possible choices for writing tests helps. (b) Tests need to be reusable. The guiding library of test description language components is the link between the development team, which exposes the functionality needed for testing, and QA, which uses them.

Basic understanding of BDD usage: The example in Fig. 2 shows only some features of the tests specified in BDD and Gherkin. The tag keyword, such as *@Sound*, denotes a category for the test. It is important to group the tests in categories for the sampling methods that decide in which directions more or less computational resources should be spent for the tests. The general pattern of a test specification is to define three main steps:

1) Set the context of the application. In our example, the state of the application under test must have a launched game instance in order to run the test. It is specified with keywords such as *Given*, *And*, or *But*. More complex examples that use a similar context can be specified with keywords: *Background* or *Outline*.
2) Set when the test should start. This is the trigger to start the test. In this example, we start the test when certain conditions occur, as shown in the table in Fig. 2. A test can be valid multiple times during runtime.
3) Expected test outcome. Specifies the correct expected results of the test, using *Then* keyword.

The implementation library for each of the steps is created by developers in a different source code file to hide implementation details that other stakeholders are not interested in. In the given example in Fig. 3, we use a data table to reuse the same test for a parameterized set of contexts, triggers and expected results. These parameters become input arguments to the source code of the test implementation. More examples can be found in our repository. BDD can also be applied for animation or vision tasks in a similar way. A test definition for animation testing would involve calling external code to first project the 3D agents onto the 2D screen space and then analyze their motion from either a high-level perspective (e.g., testing the direction of motion) or from a lower-level skeletal motion reconstruction. A similar method can be applied to computer vision analysis, by invoking inference on pre-trained deep neural models or classic OpenCV techniques.

Feature: Sound testing

@Sound
Scenario: Check dialogs
  Given we start a game instance
    And we loaded sub-level {MissionId}
  When {EntityId} started {BehaviorId}
  Then we should hear {SoundId} with similarity above {SimT}

Examples:

MissionId	EntityId	BehaviorId	SoundId	SimT
mission1	aircraftW	startEngine	snd_startAirEngine	0.75
mission2	Wedge	startComment1	comment_1	0.8
mission2	gameMusic	missionStart	snd_backgroundMiss2	0.75
mission1	aicraftW	hitByEnemy	comment_2	0.8
mission2	aicraftW	crashed	snd_crashAirplane	0.75

Fig. 2: Sound test template and data table defining the set of instances of the test. The specification is given in natural language. The context parameter is used to setup the context between tests. For example, in the *Given* step implementation, the code could cache information about the components of the game instance.

```
@then("we should hear {SoundId} with similarity above {SimT}")
def step_impl(context):
    def checkSoundSimilarity(context, SoundId, SimT):
        assert context.sndCheck.CheckSim(SoundId, context.recordedSound) >= SimT
```

Fig. 3: Implementation source code of the expectation step. External components invoked to do separation of concerns.

B. Tests scheduling and prioritization

We denote by T the set of all tests available and defined using the method described in Section IV-A. 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 *Given* and/or *When* clauses (triggers) specified in each $test \in T$ and creates a compatible subset for each frame: T_{compat}. Thus, any $test \in T_{compat}$ can be executed at that time. When $test$ is selected for execution, the playtest agent performs the specified action given in the *When* and/or *Then* clause group of the test. The expected correct result is determined by evaluating the specifications within the *Then* clauses.

Our strategy is to tag the tests and prioritize them using a custom tool that allows the user to select the priority of each test group or individual. This is important because the user can target the tests to specific areas of the game where problems are common or that might be affected by the newly added code. Priorities can be adjusted dynamically, even at runtime (see supplementary video material for a visual example). 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 encourage rapid writing of test specifications by building a visual tool interface.

Our prioritization tool also allows users to organize tests into categories and assign the execution rate of each test category or individual tests, $test \in T$, $test_{rate}$. Each $t \in T$ has a

scheduled time $test_t$ of when it should be executed next. The schedule is stored in a priority queue PQ_T, sorted by the scheduled time. Thus, at each system time $CurrentTime$, the framework executes Algorithm 1 to evaluate the compatible set of tests at that time, T_{compat}. Note that the executed states are rescheduled. This mechanism is also known as time-slicing scheduling in operating system environments.

C. Methods used for computer vision analysis

Our framework deploys various external technologies based on computer vision to test the expected behaviors. 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 [28] for text recognition. For testing purposes where the presence of objects or specific features in the image recognition output needs to be located or proven, either template matching [28], a model such as the one described in EfficientNet [4], or scene segmentation methods such as the one presented in [32], are used. Examples 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 and Fig 6.

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 [3] model with specific object classes (visual features) for each individual environment. An example is shown in Fig. 4. 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, then reproject it internally in the brain as being in the 3D space. Thus, to analyze if, for example, a particular object is moving towards a certain direction, considering as input only the visual output, it suffices to build the representation of the objects from the 2D image space, reconstruct a 3D space and analyze the motion of the objects in there. The reconstructed 3D space does not have to contain the ground truth coordinates, but needs to keep the correctness of the relative positions of the analyzed objects.

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 [33] 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 EfficientNet [4] method, 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. 5.

D. Methods for animation testing

After analysing the problems with animations in games, we concluded that the main problems in this context are that characters freeze (do not move) or move in a different direction from the user's requested input. To address these issues, our method for testing animations consists of two parts:

- First, we determine a set of fixed points in the scene using the methods described in Section IV-C. We consider these as reference points of the scene. An example of a result of this step from the *FIFA22*[3] game is given in Fig. 6.
- After a series of transformations (documented in the same figure and in our source code repository), the system obtains the skeleton of the character in each frame of the test using the MoveNet [6] method. The trajectory of the skeleton related to the fixed point found in the first step is compared to the trajectory desired by the user. Usually, the specification of tests involves a selection of a time period for the motion test, e.g. between $30 - 180$ frames. Note that the skeletal method used can be applied not only to humans, but to any entity that can be abstracted with a skeleton, including animals, vehicles, buildings, static objects in general, etc.

E. Methods used for sound testing

Our framework can automatically test whether the sound generated by the game is correct or not (with some degree of error). The methodology of exposed tests is to have the client create a batch of tests that start on certain triggers (e.g., when a person starts talking, an engine starts, etc.), record the audio capture from an internal device, and then check whether the recorded output has a certain degree of similarity to the ground truth in the database. A set of these tests is shown in Fig. 2.

From our experiments, we concluded that there are two different ways to achieve good sound comparability, depending on the use case.

[3]https://www.ea.com/en-gb/games/fifa/fifa-22

Algorithm 1 Test set scheduling in the framework

Input: PQ_T, $AppState$, the application state
repeat(time $CurrentTime$ of application)
    // Create the test suite that are ready for execution and have the compatible context in $AppState$:
    $ReadySet$ = $\{test \in PQ_T \,|\, CurrentTime - test_t <= 0\}$
    $ReadyCompatSet = \{\text{test}|\ \text{test} \in ReadySet\ and\ \text{test} \in T\_compat\}$
    // Remove the set from the priority queue:
    $PQ_T = PQ_T - ReadyCompatSet$
    // Step 1: Execute the tests now:
    for $test$ in $ReadyCompatSet$ **do**
        res = Execute(test)
        (depending on the test type, it can either update to continuing progress, or just write some output value)
        $OnResultsReady(test, res)$
    end for
    // Step 2: Re-schedule the removed tests by their test rate parameter:
    for $test$ in $ReadyCompatSet$ **do**
        $test_t = CurrentTime + test_{rate}$
        $PQ_T = PQ_T \cup test$
    end for
until End of application life

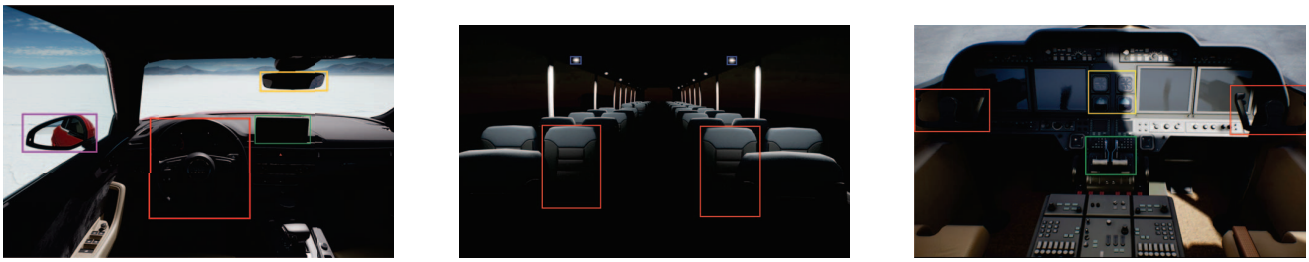

Fig. 4: The image shows the detected visual features (bounding boxes) after an environment change, when one expects to be in different vehicles after the game testing agent has triggered an action key. The left image is from the interior of a car, and the recognized feature classes are: mirrors, steering wheel, and navigation board. In the middle, it can be seen the interior of a bus. The detected features are two bus seats and two particular light bulbs. In the right image, it is displayed the interior of an airplane, where two rudders, an info display and a speed controller are detected. 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 and some additional assets in a custom scene. We used different camera angles to also test whether or not the newly trained recognition model overfits and affects its performance (Section IV-C).

Fig. 5: 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 [8]. As mentioned in Section IV-C, 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.

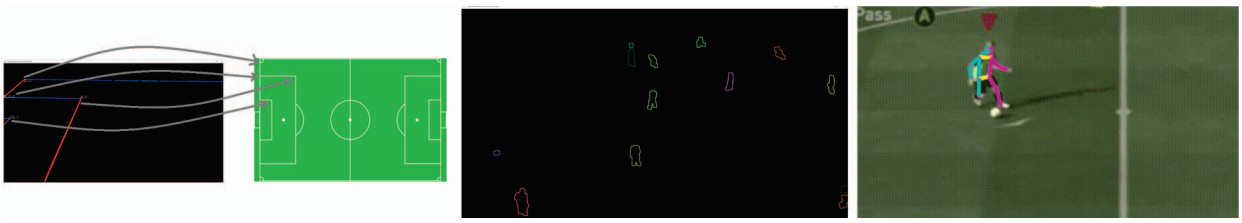

Fig. 6: The process of detecting the correctness of the played animation against the user input in the *FIFA22* game. On the left, our first step is to detect key points of the scene using raw computer vision methods and calculate the homographic transformation from 3D space to 2D. In the middle of the image, it is shown an intermediate step of the process (more details about all the steps involved can be found in our source code repository), which detects the contours of the character and correctly finds the player controlled by the user. On the right, the method zooms in on the controlled player and runs MoveNet to detect the player's skeleton. Then, for a requested sequence of N Frames, the system records the movement with respect to the detected scene's keypoints and compare them with the user's input trajectory to evaluate the correctness of the movement.

- Testing background music and effects (e.g., an engine noise on startup, an explosion, the firing of a gun). Behind the scenes, we use the Librosa library [34] to read sound data and then convert it from the time domain to the frequency domain using the FFT transform. We use a default sampling rate of 44.1 kHz, and each frequency bin used has a size of 1 Hz, with each bin containing the average of the original frequencies in the corresponding range at the end. To compare which sample should have been played in ground truth and which was actually played, we compare the difference between the two spectrograms. A similarity check for the background music in the tank game is shown in Fig. 7. The hypothesis of whether or not it is the same class is tested using a final statistical T-test.
- Sound testing for in-game character dialogs. In this case, we are relying on NLP solutions to convert the played voice into text. We then compare the text similarity between the output of the conversion model and the text in the database entry corresponding to the voice. For the model itself, we use Facebook's *wav2vec 2.0* model [35], which uses Deep Learning and Transformers methods to convert the voice to text. For the second part of checking the similarity between the output of the model and the entry in the database, we simply consider the ratio between the original text and the longest common substring (LCS) between the two tests.

F. Evaluating registered metrics

Games typically need to store a set of statistical metrics over extended play sessions. Common functionalities in any game are things such as frames per second (FPS), memory usage, the number of entities in a given area of a map, etc. Other functionalities relate to statistical regression of gameplay, difficulty, or AI. For example, in a soccer game, it would be desirable to evaluate the dribbling or shooting abilities of the AI agents during a series of consecutive games for each difficulty level. Then, the test specification could claim that the results are not within a valid range specified in the $Then$ clause (a similar test specification with a data table of conditions and parameters is shown in Fig. 2. The same automatic evaluation and conditions can be performed for all ranges with similar variables or parameters. To implement this, we store persistent data within the plugin framework component that is deployed on the game side. The user can specify the time rate for collecting these variables (e.g., the FPS metric typically has a cadence of 1 frame, while statistics such as those collected by AI systems can be collected at 10 frames). For more details, see our supplementary video material and Section V.

V. Architectural and Implementation Level

It is important to do separation of concerns as much as possible at the architectural level, for two main reasons we have identified:

- It is expected that games will be tested on different platforms such as embedded devices (e.g., mobile devices, consoles), PCs, web browsers, etc. In addition, testing should often be performed on different machines other than the application under test, since the deep network architectures used to evaluate correctness for some of the tests require different computing power, e.g., a specific class of GPUs.
- The decoupling of concerns is an important adoption factor. For example, different games may have different requirements, so managing components through interfaces and reusability can be a valuable benefit.

The overview architecture is shown in Fig. 8. The implementation of the game application usually provides an API for interacting with the testing side and possibly mocking some of its internal components. In the following, we call this API and its interaction with the game's internals the $Model$ (right side of Fig. 8).

The framework architecture consists of three main components, described below:

- *Framework - Tests Specifications*. The test scenarios are written in different files and are generally visible to all stakeholders. The source code implementation can be in

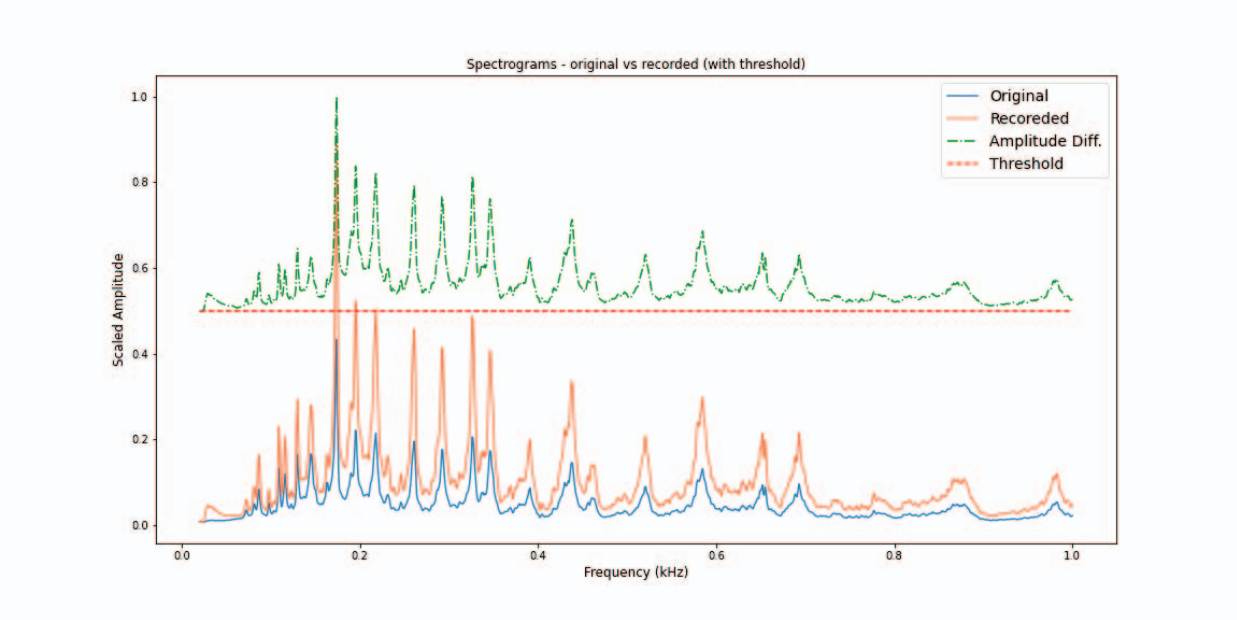

Fig. 7: Comparing the amplitude difference in the frequency domain of the recorded versus the expected sound sample.

different programming languages and it is the responsibility of the developers to implement the features needed for the test specifications. A library of patterns and examples could be provided by the game client/developer to provide simple, out-of-the-box specifications to non-technical people on the project. The implementation uses the model that defines the interaction with the game, as described below.

- *Framework - Tests Execution*. This is the place where various components manage the set of existing tests and their execution, result analysis, and the gameplay agent. Communication with the game side is handled by a communication stub component, defined below. Briefly, its subcomponents implement the following things: (a) $Agents$ - which is an extensible collection of agents that can play the game, as enumerated in Section II, (b) *Tests scheduling / core execution* - takes care of scheduling and concrete execution of tests, as defined in Section IV-B, (c) *Results Analysis* - is where the implementation of the components described in Section IV resides. (d) *Testing output support* - an extensible collection of tools to display the results. This can basically consume the output of the *Results Analysis* component and display it in visual tools and web browsers (see the supplementary video material for examples). Framework clients can extend this component to their own custom visualization tools. For the low-level communication aspects, the implementation can either use technologies such as REST API-enabled tools such as Flask[4], or the simple sockets libraries available in almost all modern operating systems. For the game world and performance aspects, choosing a socket library and building message classes and custom serialization on top of it is generally the better solution, in our practical experience.
- *Framework - Game Plugin*. This component is the middle layer between communicating with the*Framework - Tests Execution* features and the $Model$ exposed by the game application under test. It also stores persistent data such as images, sound recordings, or motion data captured over a series of frames for further analysis by the other components. A subcomponent that controls the scene or agents playing the game via the $Model$ is also required here to perform various tasks. The component itself is integrated into the game under test using a decoupled plugin architecture, so that if the game needs to be deployed to the market either physically or as an online service, there is no concern that the components on the test side will be an overhead or another source of problems or exploits.

[4]Flask project: https://flask.palletsprojects.com/en/2.0.x/

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 is this kind of automatic visual verification of results.

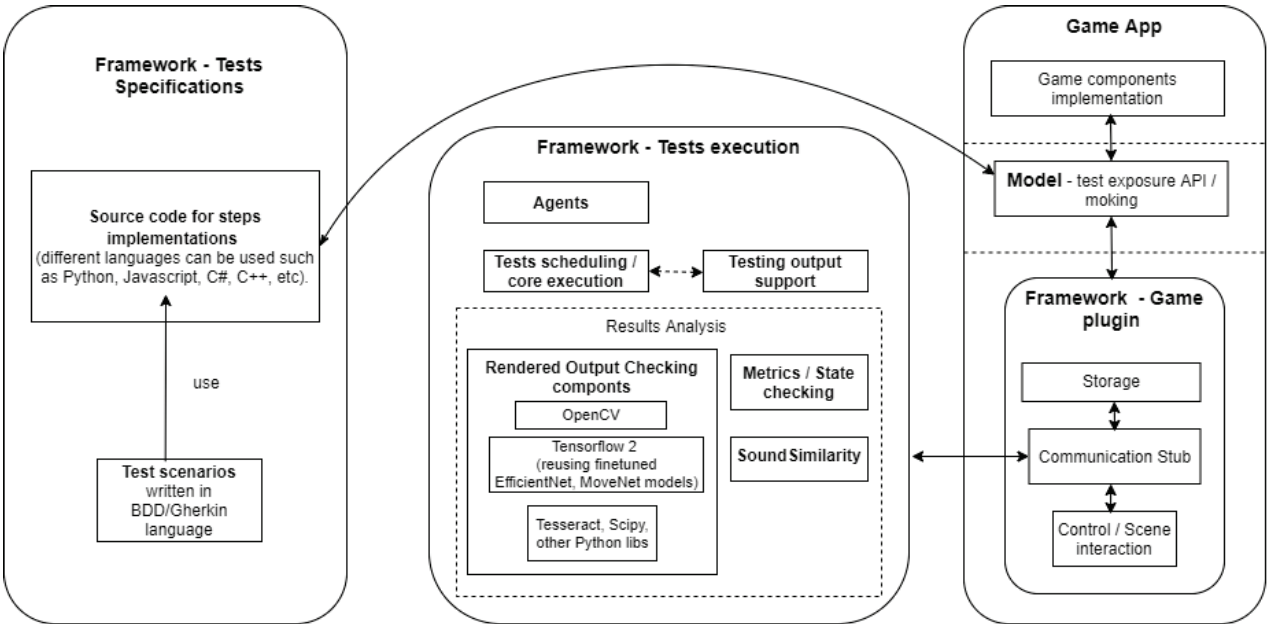

Fig. 8: Overview of the architecture of our framework and how its components interact with the game under test. The three main components (prefixed with *Framework*) are separate from each other, implement different concerns, and can be developed and deployed independently. For more details on each component, see Section V.

.

A. Interpretation of visual results evaluation

For the UI text and the detection of simple visual features recognition, such as the weapon cross tests given in Section IV-C, 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 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., a pistol instead of a 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 the default 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 in the tested frame contained too many visual effects, 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 tested vehicles even if the button was pressed, in the other half we mimicked a good test result (Fig. 4). In the interior, we used different camera angles and illumination models or intensities to ensure that the model did not overfit on certain features. In this case, the accuracy was 100% in both cases. Note that the model has to detect between three individual classes of vehicle interiors: a car, a plane, and a bus.

In the accuracy tests for movement control, we used the $Tanks$ demo to fire projectiles between enemies and then tested whether the visual representation of the fired projectile in each test confirmed the internal game state in terms of the direction of flight or the position of the projectile 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 calculating the transformations mentioned in SectionIV-C. 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 target detection 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

checking the visual results can be used with success to detect most cases. We also point out that human testers are also prone to errors due to various physical factors such as composure, stress, visual obstruction or attention problems, etc. In the 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 EfficientNet [4] model we obtained on average $\sim 15.37ms$ 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.

C. Sound testing evaluation

After evaluating the most common defects regarding sounds playing in computer games, we identified two main sources of defects on this topic: (a) Sounds that do not play at all due to incorrect triggers (b) Sounds that are interrupted because certain events occurred in addition to the previous trigger that generated the sound. For these two common problems, our method proved to correctly detect a $\sim 93\%$ out of 273 audio samples tested. The misclassified audio samples are mainly those that were interrupted a little earlier than they should have been, and which our thresholding system still considers to have played correctly. However, when testing voices using the method in [35], we found a false positive rate of 33% out of 27 voices in the $Start Wars\ Rogue\ Squadron$ game[5]. The low false positive rate is an important factor in trusting the system, since incorrectly reporting requires human effort. In order to make the system usable, we are leaving this as future work. Currently, we consider the idea to fine-tune the original model to game-specific voices and see if it produces better text transformation output.

VII. Lessons learned and future work

While developing the RiverGame tool, we gather several insights and new ideas to continue the research and implementation in the important and relevant field of game testing. In our previous position paper [9], we thought about improving the framework using game graphical blueprints [36] combined with model-based testing, symbolic execution, and fuzzing [37], but also reinforcement learning and Robotic Process Automation (RPA) bots [38] as test agents at the UI level of the game [39]. We provide below other interesting aspects that we discovered.

The decision to use the BDD methodology and Gherkin to specify input in a natural language was a well-studied decision after much trial and error. In game development, multiple non-technical stakeholders (e.g., artists, producers, quality assurance, managers) should ideally describe the usability of the features they want in the game and their correctness from their perspective. A natural language description of the specifications, combined with a reusable set of templates to describe them, facilitates the management and decoupled development of tests throughout the life of the project. It is difficult to express the benefits of one method or another in numbers, but with our proposed method we have found that more people with different perspectives have been able to write specifications correctly, whereas previously only people with programming skills could write correct and valuable specifications.

While developing the interaction between the BDD and the game, we found out that by using RestAPI is generally difficult to maintain states correctly and communicate bidirectionally between the game side and testing side. This is one of the reasons why we concluded that we should use *websockets* and a collection of persistent logic mechanisms in our implementation. Separation of concerns in the architecture is also an important factor that we have noticed during the project development so far. For example, one of our plans for the future is to collaborate with the industry to incorporate RPA [39] into the testing process to possibly replace BDD in some scenarios.

Prioritizing inputs is also a valuable thing, as developers typically complain about the time it takes to quickly test between successive versions of a game. Our visual tool addresses these concerns in an initial phase, but more work should be done in this direction. For example, we should have an automatic agent that detects the areas of the game that are affected by either the new source code changes or the changed assets and change

the priorities automatically, rather than relying on humans to change the priorities within the visual tool.

As our tool will mature, we hope that it will be useful to practitioners from the game industry and at the same time to provide a playground to academics to experiment and transfer the latest research ideas to the field of 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] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. of CVPR'16*. IEEE, 2016, pp. 779–788.

[4] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. of ICML'19*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 6105–6114.

[5] 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.

[6] R. Votel and N. Li, "Next-generation pose detection with MoveNet and TensorFlow.js," May 2021. [Online]. Available: https://blog.tensorflow.org/2021/05/next-generation-pose-detection-with-movenet-and-tensorflowjs.html

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

[8] 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 ICECCS'19*. IEEE, 2019, pp. 170–179.

[9] C. Paduraru, M. Paduraru, and A. Stefanescu, "Automated game testing using computer vision methods," in *Proc. of the 1st Int. Workshop on Automated Software Engineering for Computer Games (ASE4Games'21), in conjunction with ASE'21*. IEEE, 2021, in press.

[10] N. Tziortziotis, K. Tziortziotis, and K. Blekas, "Play ms. Pac-Man using an advanced reinforcement learning agent," in *Proc. of SETN'14*, ser. LNCS, vol. 8445. Springer, 2014, pp. 71–83.

[11] 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.

[12] 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.

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

[14] A. Khalifa, A. Isaksen, J. Togelius, and A. Nealen, "Modifying MCTS for human-like general video game playing," in *Proc. of IJCAI'16*. AAAI, 2016, p. 2514–2520.

[15] S. F. Gudmundsson *et al.*, "Human-like playtesting with deep learning," in *Proc. of CIG'18*. IEEE, 2018, pp. 1–8.

[16] B. Tastan and G. Sukthankar, "Learning policies for first person shooter games using inverse reinforcement learning," in *Proc. of AIIDE'11*. AAAI, 2011, pp. 85—90.

[17] 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.

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

[19] 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.

[5]https://www.ea.com/games/starwars/squadrons

[20] 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.

[21] 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.

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

[23] 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.

[24] 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 ASE'19*, 2019, pp. 772–784.

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

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

[27] I. S. W. B. Prasetya and M. Dastani, "Aplib: An agent programming library for testing games," in *Proc. of AAMAS'20*. ACM, 2020, pp. 1972–1974.

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

[29] M. Irshad, R. Britto, and K. Petersen, "Adapting behavior driven development (BDD) for large-scale software systems," *J. Syst. Softw.*, vol. 177, p. 110944, 2021.

[30] T. R. Silva and B. Fitzgerald, "Empirical findings on BDD story parsing to support consistency assurance between requirements and artifacts," in *Proc. of EASE'21*. ACM, 2021, pp. 266–271.

[31] E. C. dos Santos and P. Vilain, "Automated acceptance tests as software requirements: An experiment to compare the applicability of fit tables and gherkin language," in *Proc. of XP'18*, ser. LNBIP, vol. 314. Springer, 2018, pp. 104–119.

[32] L. Porzi, S. R. Bulo, A. Colovic, and P. Kontschieder, "Seamless scene segmentation," in *Proc. of CVPR'19*, 2019, pp. 8277–8286.

[33] D. Baráth and L. Hajder, "Novel ways to estimate homography from local affine transformations," in *Proc. of VISIGRAPP'16*. SciTePress, 2016, pp. 434–445.

[34] B. McFee, C. Raffel, D. Liang, D. Ellis, M. Mcvicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in python," in *Proc. of SciPy'15*, 2015, pp. 18–24.

[35] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, "wav2vec 2.0: A framework for self-supervised learning of speech representations," in *Proc. of NeurIPS'20*, 2020.

[36] 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.

[37] C. Paduraru, M. Paduraru, and A. Stefanescu, "RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing," in *Proc. of ICST'21*. IEEE, 2021, pp. 430–435.

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

[39] M. Cernat, A.-N. Staicu, and A. Stefanescu, "Improving UI test automation using robotic process automation," in *Proc. of ICSOFT'20*. SciTePress, 2020, pp. 260–267.