

# Liquid Snake: a test environment for video game testing agents

Pablo Gutiérrez-Sánchez<sup>1,\*</sup>, Marco A. Gómez-Martín<sup>1,\*</sup>, Pedro A. González-Calero<sup>1,\*</sup>, Pedro P. Gómez-Martín<sup>1,\*</sup>

<sup>1</sup>Complutense University of Madrid, Madrid, Spain

## Abstract

In recent years, a number of benchmarks and test environments have been proposed for research on AI algorithms that have made it possible to evaluate and accelerate development in this field. There exists, however, an absence of environments in which to evaluate the feasibility of such algorithms in the context of games intended for continuous development, in particular in regression testing and automatic error detection tasks in commercial video games. In this paper we propose a new test-bed - Liquid Snake: a 3D third-person stealth game prototype, designed to conveniently integrate autonomous agent-driven quality control mechanisms into the development life cycle of a video game, based on the open source ML-Agents library in Unity3D. Focusing on the problem of regression testing on the potential unexpected changes induced in a game by altering the AI of enemies, we argue that this environment lends itself to be used as a sample test environment for automated QA methodologies thanks to the complexity and variety in the behaviors of NPCs naturally present in stealth titles.

## Keywords

Benchmark, QA, regression testing

## 1. Introduction

In the continuous development of a commercial title, it is common to find numerous enemies and NPCs being reused in different sections of the game that evolve over time. In this context, the evolution of the AI controlling these agents can end up inducing non-negligible modifications in the gameplay of a level, contrary to the intentions of the original designs. These changes are not always straightforward to detect: on the one hand development teams often do not have the resources to perform sufficiently exhaustive testing tasks, and on the other hand these modifications do not necessarily have to “break” sections of the game, simply altering the user experience in a more or less subtle way and thus may go unnoticed by testers with extensive experience in the game.

The tests involved in the task of determining whether a feature or a design that was correct in the past continues to work properly after some progress in the development of the project

---

*1 Congreso Español de Videojuegos, December 1–2, 2022, Madrid, Spain*


\*Corresponding author.

✉ pabgut02@ucm.es (P. Gutiérrez-Sánchez); marcoa@fdi.ucm.es (M. A. Gómez-Martín); pagoncal@ucm.es (P. A. González-Calero); pedrop@fdi.ucm.es (P. P. Gómez-Martín)

🆔 0000-0002-6702-5726 (P. Gutiérrez-Sánchez); 0000-0002-5186-1164 (M. A. Gómez-Martín); 0000-0002-9151-5573 (P. A. González-Calero); 0000-0002-3855-7344 (P. P. Gómez-Martín)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

are known as regression tests, and their cost of execution grows dramatically as the volume of the code base and elements included in the game increases, as they essentially entail the recurring repetition of previously executed test batteries or checks. These tests may raise functional questions (“does the enemy continue to approach the player when it encounters them at a distance below a certain threshold?”), or more abstract questions linked to the game design (“is it possible to complete the level in less than 10 minutes and without losing health points?” or “is it possible to traverse the level while picking up all the collectibles and without being detected by enemies in the process?”, to name a few examples). While the first types of questions can often be addressed through the use of tools such as unit tests, for which most commercial engines offer good support, this is typically not the case for the second type of problems, which generally require humans playing the game repeatedly trying to figure out an answer to the question posed by the test.

Since the latter poses both an economic and logistical bottleneck, different strategies have been proposed in recent years in an attempt to automate these checks and alleviate the burden they can place on a QA team. These include ideas such as replaying game traces recorded by human players during testing [1] or training autonomous agents based on AIs capable of interacting with the game in specific ways, each of them being more or less feasible to implement in a real development context [2, 3, 4].

While it is true that nowadays there are numerous standardized test-beds and benchmarks for machine learning techniques that are intended to act as “collective challenges” in the community to guide research efforts towards solving specific problems and to serve as environments for testing new algorithms, the same cannot be said for applications of such techniques to development cycles within commercial studios. Regression testing techniques described in the literature are often obtuse or inaccessible from the perspective of development teams, with proprietary use examples or technologies that are difficult to transfer to new environments.

It is therefore relevant, from our point of view, to propose a test-bed oriented to serve as a testing framework not only for machine learning algorithms, but also for their applications on quality control strategies in commercial games and the development of support tools for automated testing. Having a common environment also enables a shared vocabulary in the community: for instance, it becomes possible to compare the results of two ways of approaching a regression testing problem on specific and shareable modifications (such as “does my method detect that it is more difficult to evade level 2 enemies after altering a node of its behavior tree?”).

On the other hand, from the developers’ point of view, having a simple reference environment where they can try out automatic testing mechanisms allows them to experiment with different strategies in a smaller external project and validate them before taking the step of integrating them into their own games. At the same time, this test-bed can be used as an architectural reference for those teams that are considering undertaking automatic testing but are held back by the complexities and technical unknowns associated with the problem.

With this, in this paper we present Liquid Snake - a third-person 3D prototype belonging to the stealth genre and developed in Unity3D intended to act as a common test-bed for regression testing and automatic quality control, as well as an architectural reference for the integration of such methods in commercial projects. The rest of the paper is structured as follows. Section 2 discusses related work of interest in this field. Section 3 introduces Liquid Snake along with

the most relevant high-level dynamics within the game to provide a general understanding of the prototype. In turn, section 4 details the technical and architectural features that we argue make the project suitable for the uses described above, ending in section 6 with conclusions and future work.

## 2. Related work

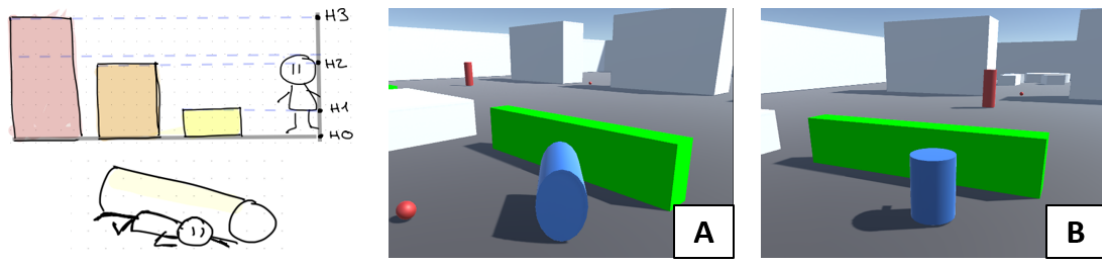
In view of the problem described above, in recent years a number of strategies have been proposed to implement automatic testing methods, typically making use of autonomous control agents capable of interacting with a level over a large number of simulations while collecting metrics that must fall within specific ranges to consider that the user experience has not been altered [5, 6]. This same principle has been used in platforms and engines such as Unity3D to introduce tools to support the design and balancing of video games [7].

To create these control agents, one of the most straightforward alternatives is simply to make use of segments recorded manually by a human performing the specified tasks, replaying them every so often over the original environment to check that the player's trace is still able to complete the set objective [8]. However, when the structure of the environment is modified, or the environment includes random elements that do not remain constant between runs, these strategies are no longer valid, motivating the need to create agents with a certain capacity to adapt to changes in their surroundings.

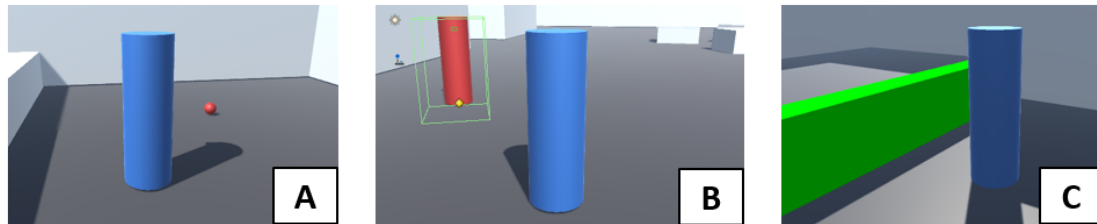
This is where methods based on AI-based strategies come into play, offering more reactive and adaptive policies to changing environments. Some examples of these techniques for automatic testing in video games adopt the use of machine learning algorithms based on Deep Reinforcement Learning (DRL) [9, 10], Imitation Learning (IL) [11], or even hybrid models of the previous strategies with control structures such as behavior trees (BTs) [5].

Nonetheless, the reality of the matter is that at present the implementation of these methodologies in commercial developments suffers from several integration problems that discourage developers from employing them in their projects. First, the generation of autonomous agents capable of naturally playing a given game is complex and requires non-trivial knowledge in the field of machine learning to be implemented. This is cushioned to some extent by new tools such as ML Agents [12] in Unity3D or MindMaker [13] in Unreal Engine, which allow training policies in a more developer-friendly way, but these are mostly aimed at research or small-scale proof-of-concept creation. Secondly, to our knowledge there is a dearth of accessible benchmarks and environments that serve as showcases and examples for the use of automatic testing techniques in games intended for continuous development. Although there is a wide variety of benchmarks and environments designed to test different types of machine learning algorithms, such as DeepMind Lab [14], MineRL [15], OpenAI Gym [16] or Starcraft II Learning Environment [17], to name a few, all of them stem from a closed game in which the aim is to find a policy capable of fulfilling a certain fixed objective (or maximizing a given performance metric), as opposed to the problem that concerns us: starting from a game in open development and using agents to perform regression tests as it is modified.

In this paper we present a prototype stealth game implemented in Unity3D - Liquid Snake, as a contribution in progress to this field, intended to serve as an open source testing envi-



**Figure 1:** Height system in Liquid Snake. Captures from player in crawling (A) and crouching (B) states.



**Figure 2:** Sample interaction scenarios in Liquid Snake: picking up a collectible item (A), stealthily attacking an enemy from their back (B) and jumping over a log (C).

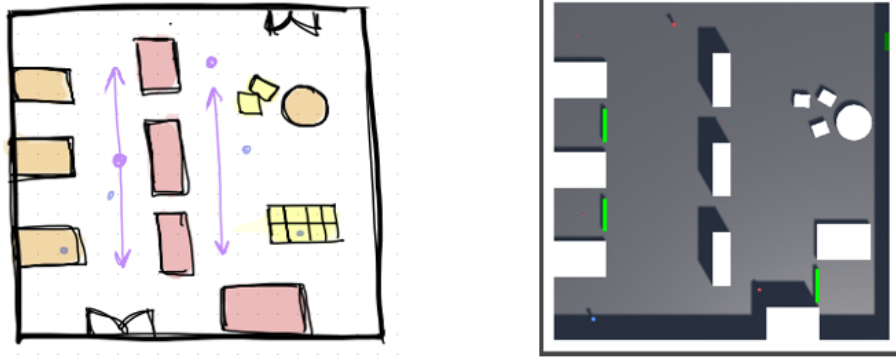
ronment that comes equipped with the necessary tools to experiment with automatic testing methodologies on a project under development.

### 3. Liquid Snake

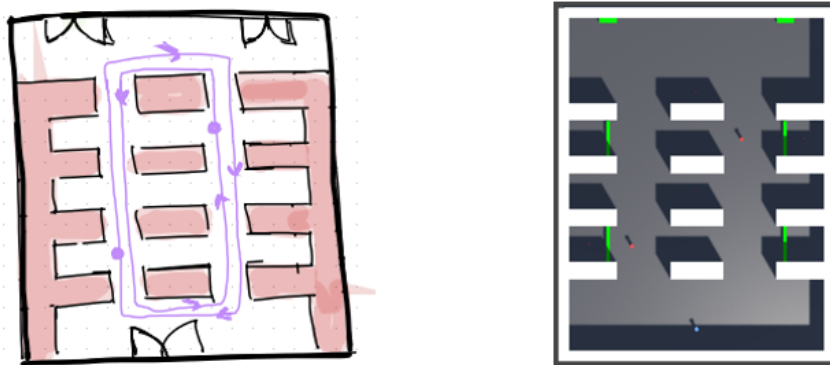
Liquid Snake is a game with mechanics inspired mostly by the stealth genre: in it, the player must navigate through different 3D rooms in an attempt to find the exit, while trying to avoid the enemies that patrol the area in search of intruders and collecting as much loot as possible as they move through the environment. The player has a finite number of health points that are reduced every time they are hit by an enemy projectile, being defeated when these are reduced to 0. Players may choose to walk, run, crouch, or crawl to navigate the room, thus modifying the height at which enemies perceive them, their movement speed, and the noise they produce (which influences the enemies' ability to detect them).

The game considers two different heights for the player, as shown in figure 1. When deciding to walk or run, the player maintains the default height  $H2$ , the only difference between the two states being the speed of the character's movement and the noise produced (running being the faster but louder action). In the crouching and crawling states, the player adopts a height  $H1$ , which allows them to conceal themselves behind low obstacles such as the trunks in the figure so as to go unnoticed by enemies operating at a height  $H2$ . As with the walking-running pair, the only difference between the crouching and crawling states is the speed of the player's movement and the noise generated, with the crawling state being the slower but stealthier of the two.

The game also features interaction actions with objects in the environment and shooting



**Figure 3:** Sample level 1: disjoint enemy patrol paths separated by columns.



**Figure 4:** Sample level 2: clockwise overlapping enemy patrol paths.

with limited ammo to deal with certain vulnerable enemies. Some of the available interactions in the sample environments can be found in figure 2. This includes scenarios such as picking up a collectible item (figure 2-A), stealthily eliminating an enemy by interacting with it from its back before being detected (figure 2-B), or jumping over a low obstacle leaving the player positioned on the opposite side of the object (figure 2-C). In all cases, the player must approach the element with which they wish to interact and press an interaction key in order to execute the corresponding action. If there is more than one object with which an interaction can be performed, the one closest to the character at that time is selected.

The current project also includes some preconfigured levels that can be used as a reference for testing. All of them include at least one enemy with a predefined patrolling route and a number of collectibles hidden in various less accessible areas of the level, as well as other interactive elements such as jumpable logs. Some examples can be found in figures 3 and 4.

The decision to take a stealth game as a starting point for this test-bed is based on the fact that these environments are particularly appealing due to the presence of a number of mechanics that are very prevalent in a wide variety of genres and games and the presence of NPCs whose behaviors significantly determine the gameplay of the level. Indeed, in a game of this type it is

common to design enemies in very specific ways to motivate certain strategies on the part of the player, but also to increase the complexity of the AI so as to adapt it to new design needs, which is why we believe them to be an ideal candidate for regression testing.

The source code for the project, along with some usage examples, may be found in [18].

## 4. Testing environment features

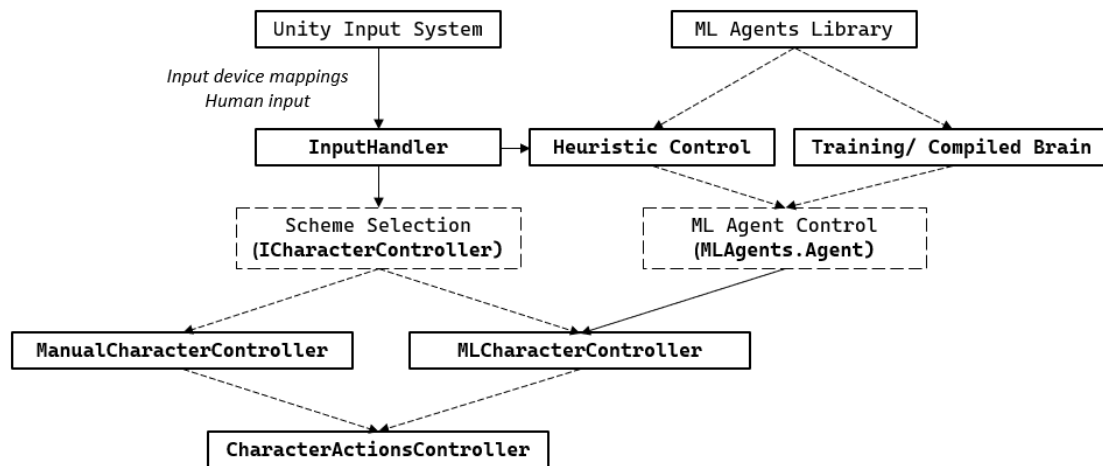
In this section we will summarize the most relevant features of the proposed testing environment: its native integration with the ML Agents library in Unity 3D, the chosen architecture for the input and actions schemes to conveniently switch between control mechanisms, and the use of Behavior Trees to support growing enemy policies.

### 4.1. Natural integration with ML Agents

Liquid Snake is natively integrated with the ML Agents library for the training of autonomous agents that act as automatic testers and, as will be described later, allows to switch smoothly between manual control of the character and control by means of agents already trained or in training. Additionally, each level of the project is wrapped by a `LevelManager` to orchestrate the initialization and reboot processes of the environment between training episodes. To coordinate the resetting of the components within the environment, an `IResettable` interface is provided with a single `Reset` method, to be implemented by whichever level constituents require a return to their initial configuration at the beginning of a new training episode, or when respawning the player after dying. The `LevelManager` is responsible for subscribing to all events in the environment that may result in a level reset (player death, time limit exceeded, etc) and maintains references to all elements adhering to the `IResettable` interface to call their corresponding `Reset` methods each time one of these events is triggered in-game. This way, it is sufficient to implement this interface to ensure that a new level element is restarted when necessary, without the need to establish an explicit dependency between components, thereby facilitating the scalability of the project.

A `CharacterEvents` component is also provided for global notification of the different events of interest involving the character, which is particularly useful when linking occurrences with training rewards (for instance, granting a positive reward after the event of reaching the room exit) or with logistical operations on the scene (such as restarting the level from the `LevelManager` after the event of character death). In practice, all the events triggered by the various components that define the character's behavior (health, interactors, movement managers, etc.) end up being intercepted by the `CharacterEvents` component, which lifts and unifies them to provide a single access point from the outside to the catalog of events exposed by the character. As a result, it is only necessary to subscribe to the events of this component rather than having to access each component of interest individually.

Once a trained model is available, it is possible to perform simulations on the level of interest while collecting different customizable metrics related to the performance of the agents. At the moment, the project provides a set of Unity 3D play mode tests that can be used as a guideline to load scenes automatically, instantiate a standalone controller, associate it to the character object and then run the corresponding scene for a fixed number of times collecting event-driven



**Figure 5:** Control scheme in Liquid Snake.

metrics. This allows to gather simulation metrics from a fair number of levels in a convenient way, avoiding manual switches and executions. One limitation at the moment, however, is that no historical record is kept of the metrics collected in the tests, which makes it necessary to keep an external storage in which to place and analyze them retrospectively.

## 4.2. Decoupled input-actions scheme

One of the main bottlenecks when integrating a machine learning model with a test environment is the difficulty to conveniently switch between control schemes. The most classic examples of this situation are found in the need to alternate between manipulating the character manually, using a policy trained by reinforcement learning, executing traces prerecorded by human demonstrators, or applying hand-scripted routines to specify desired behaviors.

With this in mind, we consider it essential to start from a model in which the command modes mentioned above are kept strictly decoupled from the actual control logic of the character, by means of mechanisms natural to a programmer familiar with the engine. The proposed scheme is summarized in figure 5, for the case in which it is intended to alternate between manual control and control through an ML Agents library agent (either via heuristics, through a trained model or a model undergoing training).

In the previous scheme, there exists an abstraction layer that is always present within the character and acts as an interaction API with its set of supported actions, given by a component `CharacterActionsController` attached to the object to be handled. On top of this layer it is possible to place specific controllers that make direct use of this component to manipulate the character (`ManualCharacterController` and `MLCharacterController` in figure 5).

For those cases where one wishes to implement a controller that requires human input, such as in conventional controls or in the heuristic mode of ML Agents, we introduce an `InputHandler` object in the scene, responsible for transforming human input registered in Unity's input system into methods specific to any controller that implements the `ICharacterController` interface

(methods such as “Move”, “Shoot” or “Interact”, which are more manageable and understandable than raw input). It is worth mentioning here that the heuristic mode differs from traditional control in that in the former case the agent must receive the commands represented following the encoding used for the RL model, in an array of actions. This enables the collection of human demonstrations that can be later used to train learning-by-demonstration models or to reproduce historical traces. In this way, we are able to run a single input system, but operate it in different ways depending on our needs (for instance, one would not expect to use heuristic control as the main control in a commercial game).

To implement and deploy a character controller, we opt for a separation between the character actor object itself (a game object in the scene that exposes a `CharacterActionsController`) and the controller itself, which is nothing more than an empty object in the Unity scene with a component that implements the `ICharacterController` interface and is provided with a reference to the object that represents the character. In this way the character object acts as a “pawn”, and the controller can be modified as desired without further restructuring references from other objects in the scene to the player. This is convenient because in a game of these characteristics it is very common to have a multitude of elements that reference the player in a fairly direct way, being the enemies perhaps the most typical case, as they need to keep a reference to the player in order to know what to chase and attack, and to gather information from the player at run-time. If the controller were present in the character object itself, this would imply that to change the scheme it would be necessary either to replace the object in use with another one with different control components, or to allow the coexistence of numerous controllers in the same object and implement some kind of manager dedicated to enable and disable them as required, with the corresponding clutter and logistical complication that this may cause.

Additionally, when using a controller derived from the `Agent` class of ML Agents, it is common to introduce different sensor components associated to the agent to enable more sophisticated forms of perception such as visual observations from a camera, or spatial observations by means of sensors based on raycasts around the object or grids centered on the agent to detect the relative position of nearby elements in the scene. These sensors must be attached as components to the object that hosts the controller and will always dispatch their observations without the ability to disable them. Since it is usually desirable to experiment with various configurations of sensors and parameters, it seems natural to keep each controller as an independent object that can be easily replaced whenever one wishes to enforce a new perception system. One thing to note here is that if one intends to use a sensor centered on the player’s object, then it becomes necessary to ensure that the position of the controller matches that of the character, which is typically rather straightforward, but important to note nonetheless.

### **4.3. Behavior trees for rich NPCs**

As we mentioned before, in the development of commercial games with enemies and other NPCs, it is common to start with a relatively simple AI that meets the design needs of the initial levels and then evolve and expand it to adapt to new requirements as the project progresses. These AIs are usually coded using behavior trees (BTs) or state machines, which is why in *Liquid Snake* we include the Behavior Bricks library [19] for specifying the behavior of enemies in the form of BTs. From this library it is possible to design arbitrarily complex flows that can



be expanded during development.

The current project features a suite of pre-designed behavior trees to define the flows of several enemies present in the example levels that may be used as a reference for changes or expansions. Included here are nodes encoding common conditions and checks such as “is target in sight?” or actions such as “advance to the next patrol point” or “shoot at target”.

## 5. Testing process

Following the descriptions given in the previous sections, the process of creating a battery of tests on one of the project’s environments would be as follows:

1. Set up a level on which one wishes to perform a regression test as a scene within the game. Currently the levels in figures 3 and 4 are provided as testing samples (for which pre-trained controllers and Play Mode tests that make use of them are provided).
2. Configure a controller that is able to automatically perform the desired behavior at the created level. This can be done in a number of ways, either with a manually programmed AI, with traces of human inputs or, as in the case of the examples in the project, by training an agent through the ML-Agents library to take control of the player. As for the behavior to be generated, this need not be limited to successfully completing the level, but can specify more precise tasks, such as defeating a particular enemy or reaching a sequence of points in order. In the example cases, the goal of the level is always to reach an escape point before dying or running out of time, with a reward function that penalizes enemy deaths and damage, and awards positive stimuli for approaching the goal, reaching an exit point, or retrieving a collectible item. The implementation details for these controllers (observations, reward functions, configuration of the underlying neural network, etc) can be consulted in the project repository, where a quick start guide explaining a simple training process is also included.
3. Write a Play Mode test in Unity’s Test Runner that instantiates the implemented controller on the scene to be tested and binds it to the player object in the environment. In general, a test runs the controller simulation over the scene for a set number of times (which should be high enough to be in proportion to the variability of the level in order to collect as many occurrences as possible) and collects execution metrics configured by the test designer, such as number of objects retrieved or number of times that the player is detected by the enemy. These metrics can make use of the player’s CharacterEvents component to be constructed as certain standardized events are received. In the example cases, damage to the player and goal reached events are used to compute the metrics per episode of health remaining at the end of the level and time taken to escape from the room. The test executor is also responsible for compiling the execution metrics and dumping them into a log file, thus enabling subsequent analysis.
4. After applying a structural change on the scene, re-run the previous test to obtain a new set of simulation metrics with the same agent. At this point it is possible to perform a comparative analysis of the distributions of the metrics (applying, for example, a non-parametric test such as Mann-Whitney to contrast the equality of pre- and post-change

distributions) or, if preferred, simply check if they are within acceptable limits given by the design team.

The use of a Test Runner such as the one integrated in Unity allows to configure and launch large numbers of tests on an arbitrary number of scenes in an automated way, so that it is possible to repeat the capture of metrics at the end of a development day as a first verification mechanism to check whether the metrics are maintained at appropriate values.

## 6. Conclusions and future Work

In this paper we present a test environment for automatic regression testing in Unity 3D as a contribution to the field of quality control in commercial video games, arguing that it can be used both as a reference for the integration of automatic testing methodologies in new projects, and to evaluate new testing algorithms in a controlled and prepared environment before deciding to proceed to incorporate them into a proprietary project. In particular, we conclude that the architecture proposed in this paper offers a number of features that make it particularly suitable for automated testing, such as its native integration with machine learning libraries such as ML-Agents, a decoupled scheme of character controllers that allows seamless switching between input and control mechanisms, and reference scenarios that exemplify its use in Unity's Play Mode Tests for the automated execution of multiple batteries of simulations.

The test-bed described in this article is under active development, and we aim to incorporate new features that make it as easy as possible to perform tests on it, as well as to develop thoroughly documented application examples of automatic testing techniques such as those described in [5] as a showcase of the environment. In the short term, the highest priority is to improve the tools to conveniently configure and perform the automatic execution of a large number of tests with autonomous agents, as well as to collect the execution results of such tests in a structured and manageable way. This will enable a good degree of control over the evolution of the set of game sections avoiding the tedium of launching each regression test manually.

## 7. Acknowledgments

This work was supported by the Ministry of Science and Innovation (PID2021-123368OB-I00).

## References

- [1] M. Ostrowski, S. Aroudj, Automated Regression Testing within Video Game Development, *GSTF Journal on Computing (JoC)* 3 (2013) 10. URL: <http://www.globalsciencejournals.com/article/10.7603/s40601-013-0010-4>. doi:10.7603/s40601-013-0010-4.
- [2] J. Pfau, J. D. Smeddinck, R. Malaka, Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving, in: *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*, ACM,

- Amsterdam The Netherlands, 2017, pp. 153–164. URL: <https://dl.acm.org/doi/10.1145/3130859.3131439>. doi:10.1145/3130859.3131439.
- [3] S. Ariyurek, A. Betin-Can, E. Surer, Automated Video Game Testing Using Synthetic and Humanlike Agents, *IEEE Transactions on Games* 13 (2021) 50–67. doi:10.1109/TG.2019.2947597.
- [4] J. Bergdahl, C. Gordillo, K. Tollmar, L. Gisslén, Augmenting Automated Game Testing with Deep Reinforcement Learning, in: *2020 IEEE Conference on Games (CoG)*, 2020, pp. 600–603. doi:10.1109/CoG47356.2020.9231552, iSSN: 2325-4289.
- [5] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, P. P. Gómez-Martín, Reinforcement Learning Methods to Evaluate the Impact of AI Changes in Game Design, *Proceedings of the AAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 17 (2021) 10–17. URL: <https://ojs.aaai.org/index.php/AIIDE/article/view/18885>.
- [6] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, P. P. Gómez-Martín, A proposal for combining reinforcement learning and behavior trees for regression testing over gameplay metrics, *VII Congreso de la Sociedad Española para las Ciencias del Videojuego 2021* (2021). URL: <http://ceur-ws.org/Vol-3082/paper13.pdf>.
- [7] Optimize your game balance with Unity Game Simulation, 2020. URL: <https://blog.unity.com/technology/optimize-your-game-balance-with-unity-game-simulation>.
- [8] M. Ostrowski, S. Aroudj, Automated Regression Testing within Video Game Development, *GSTF Journal on Computing (JoC)* 3 (2013) 10. URL: <https://doi.org/10.7603/s40601-013-0010-4>. doi:10.7603/s40601-013-0010-4.
- [9] J. Bergdahl, C. Gordillo, K. Tollmar, L. Gisslen, Augmenting Automated Game Testing with Deep Reinforcement Learning, in: *2020 IEEE Conference on Games (CoG)*, IEEE, Osaka, Japan, 2020, pp. 600–603. URL: <https://ieeexplore.ieee.org/document/9231552/>. doi:10.1109/CoG47356.2020.9231552.
- [10] J. Pfau, J. D. Smeddinck, R. Malaka, Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving, in: *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*, ACM, Amsterdam The Netherlands, 2017, pp. 153–164. URL: <https://dl.acm.org/doi/10.1145/3130859.3131439>. doi:10.1145/3130859.3131439.
- [11] S. Ariyurek, A. Betin-Can, E. Surer, Automated Video Game Testing Using Synthetic and Humanlike Agents, *IEEE Transactions on Games* 13 (2021) 50–67. URL: <https://ieeexplore.ieee.org/document/8869824/>. doi:10.1109/TG.2019.2947597.
- [12] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, D. Lange, Unity: A General Platform for Intelligent Agents, *arXiv:1809.02627 [cs, stat]* (2020). URL: <http://arxiv.org/abs/1809.02627>, arXiv: 1809.02627.
- [13] A. Krumins, Mind maker, 2020. URL: <https://github.com/krumiaa/MindMaker>.
- [14] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg, S. Petersen, DeepMind Lab, 2016. URL: <http://arxiv.org/abs/1612.03801>. doi:10.48550/arXiv.1612.03801, arXiv:1612.03801 [cs].
- [15] A. Kanervisto, S. Milani, K. Ramanauskas, N. Topin, Z. Lin, J. Li, J. Shi, D. Ye, Q. Fu, W. Yang, W. Hong, Z. Huang, H. Chen, G. Zeng, Y. Lin, V. Micheli, E. Alonso, F. Fleuret, A. Nikulin,

- Y. Belousov, O. Svidchenko, A. Shpilman, MineRL Diamond 2021 Competition: Overview, Results, and Lessons Learned, 2022. URL: <http://arxiv.org/abs/2202.10583>. doi:10.48550/arXiv.2202.10583, arXiv:2202.10583 [cs].
- [16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016. arXiv:arXiv:1606.01540.
- [17] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekeremo, J. Repp, R. Tsing, StarCraft II: A New Challenge for Reinforcement Learning, 2017. URL: <http://arxiv.org/abs/1708.04782>. doi:10.48550/arXiv.1708.04782, arXiv:1708.04782 [cs].
- [18] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, P. P. Gómez-Martín, Liquid Snake, 2022. URL: [https://github.com/UCM-GAIA/Liquid\\_Snake](https://github.com/UCM-GAIA/Liquid_Snake).
- [19] PadaOne Games, BehaviorBricks, 2021. URL: <http://bb.padaonegames.com/>.