Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving

"For human beings, testing the same game for a longer period of time can be quite demanding of both their creativity and concentration. Since projects require different styles of testing at different times, such as simply playing through the game as quickly as possible or in-depth bug testing of various parts of the game, the testers often have to actively force themselves to leave the path their brains are used to and to come up with new creative ways of breaking the game. Additionally, even for a linear game, the number of possible combinations as well as the order they are made in during a play session can become extremely large."

- Maik Hildebrandt, Head of QA at Daedalic Entertainment [12]

Johannes Pfau
University of Bremen
Bibliothekstraße 1,
28359 Bremen, Germany

Jan David Smeddinck
ICSI, University of California, Berkeley
1947 Center St, Berkeley, CA 94704
jandavid@icsi.berkeley.edu

Daedalic Entertainment
Papenreye 51,
22453 Hamburg
Germany
jpfau@tzi.de

Rainer Malaka
University of Bremen
Bibliothekstraße 1,
28359 Bremen, Germany
malaka@tzi.de

Abstract
With *ICARUS*, we introduce a framework for autonomous video game playing, testing, and bug reporting. We report on the design rationale, the practical implementation, and its use in game development industry projects. With *ICARUS*, we introduce a framework for autonomous video game playing, testing, and bug reporting.

The definitive version was published as:
Pfau, J., Smeddinck, J. D., & Malaka, R. (2017). Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving. Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play, 153-164.
https://doi.org/10.1145/3130859.3131439

We report on the design rationale, the practical implementation, and its use in game development industry projects. The underlying solving mechanic is based on *discrete reinforcement learning* in a dualistic fashion, encompassing volatile short-term memory as well as persistent long-term memory that spans across distinct game iterations. In combination with heuristics that reduce the search space and the possibility to employ pre-defined situation-dependent action choices, the system manages to traverse complete playthrough iterations in roughly the same amount of time that a professional game tester requires for a speedrun. The *ICARUS project* was developed at Daedalic Entertainment. The software can be used to generically run all adventure games built with the popular *Visionaire Engine* [6] and is currently used for evaluating daily builds, for large-scale hardware compatibility and performance tests, as well as for semi-supervised quality assurance playthroughs.

The supplementary video depicts real-time solving with active control and observation via a *web control panel.*

Author Keywords
Automated game testing; quality assurance; reinforcement learning; automated bug reporting; con tinuous performance analysis; continuous integration testing

A	**Crashes/Freezes** Shutting down the game unexpectedly or preventing the screen from rendering any further.
B	**Blocker** Resulting in a game state from which no further game progress can be made.
C	**General** Graphical flaws, animation issues, typos, glitches.

Table 1. Common categories of bugs in video games [1, p. 178].

ACM Classification Keywords
D.2.5 [Software Engineering]: Testing and Debugging;
I.2.1 [Artificial Intelligence]: Applications and Expert Systems – Games;
K.8 [Personal Computing]: Games

INTRODUCTION

Continuo us and extensive quality assurance (QA) plays an important role in the video game industry. Modern games are often immensely complex software systems that offer a broad range of possible game experiences and are often immediately used by a large number of consumers. At the same time, bugs or game glitches can considerably harm the immersion, fun, and endanger the overall game experience. Thus, a large portion (typically ~10-20 %) [5] of the budget for a particular video game production is spent solely on finding and reporting bugs, testing traversability, compatibility, performance, and aesthetics. Such issues are usually broken down into three major categories of severity (**A**: Crashes/Freezes, **B**: Blocker and **C**: General. See: Table 1). While the order of severity is descending, the probability to miss a bug of the particular type is simultaneously ascending. Furthermore, the majority of missed bugs stems from error blindness (due to the habituation to the game procedures and the sticking to established action choice patterns), a specific form of change blindness [20], that testers grow more likely to fall victim to the more often and frequently they play-test the same game.

In this light, the introduction of ICARUS in professional video game development does not only aim at reducing labor costs for QA, but also at improving the bug tracking performance and at decreasing the cognitive load for human testers, assisting in all of the bug categories named above. Following a discussion of the current state of the art in game testing, automated testing, and the application of techniques from artificial intelligence / machine learning in these contexts, we present the rationale and architecture of ICARUS in detail, together with exemplary use cases in the form of an industry case study and a discussion that reflects on the value that such systems can currently provide in game development processes, as well as an outlook on future developments in the area of intelligent automated game testing. This technical framework description and the according case study provide a report on a novel system for automated game testing with adventure games. Readers from the scientific community will gain a better understanding of the extent to which the game industry is embracing applied artificial intelligence and machine learning in contexts beyond classic game AI, while readers with a background in the game industry can gain a better understanding of how similar approaches might benefit their own projects.

RELATED WORK

So far, automated frameworks for testing software or specifically video games have been developed. Automated approaches exist, for example for selected, discrete performance measurements, such as determining the FPS at which a game can run on a given system, or the CPU and memory load when starting or running the game using new games or saved game states [9]. While such systems can frequently detect issues in category A, blockers and especially more general flaws of non-technical nature, like unsolvable conditions in complex quests, remain undetected and require manual involvement. Other approaches simulate playthroughs, using manually

Left clicks	for each available target object
Right clicks	for each available target object
'Use' each item	with each available target object
'Use' each item	with each available target item
'Look at' item	for each available target item

Table 2. Common generic action categories for adventure games. Dialog options are handled separately, see: Dialog.

predetermined [3, 8, 11] or recorded [4, 10] action sequences. These systems can help with detecting many potential blockers and some more general issues. However, they require manual adaptation or re-recording of the action sequences whenever the procedure changes, which typically happens on a daily basis during the active game development of modern games. Furthermore, most of the time video games do not strictly constrain the player regarding the order in which a sequence of actions needs to be executed. Actions are not always mandatory to perform in order to progress in a game and often the player is given several choices on how to proceed. The former deterministic approaches thus require different manually defined (or recorded) action sequences. Even in games with just a few optional branches or decision points, the resulting combinatorial explosion clearly illustrates the limitations of such manually guided automatic testing. For some specific games, targeted automatic solvers exist that iterate over the whole possible action space of the game (e.g. in a brute force breadth-first search fashion [7]). However, these examples include a large number of actions that are repeated over and over again, although they often do not require validation in each iteration. Non-deterministic approaches were successful in spotting unwanted NPC behavior and glitches [13, 14], parameter tuning [15], testing formal core mechanics of multi-agent systems [16] or detecting every bug expressible in a proposed language [17], but in rather strictly limited situations, whereas our approach is tailored to the needs of traversing complete games. For a number of board games, complete, AI-guided play testing approaches exist [17,18,19], which clearly identified loopholes and design flaws, yet lack industrial application.

As the following section will show in further detail, the ICARUS system tackles a number of shortcomings of the systems that were discussed in this section. With an active and guided machine learning approach, it narrows the playthrough down to the most relevant actions, after having explored the complete game action set, highlighting potential yet less common blockers as well as general blockers, that - unlike crashes or freezes - could have easily gone undetected using more traditional automated testing. As Figure 7 shows, this can notably speed up the progress of QA evaluations.

ICARUS

The system for *intelligent completion of adventure riddles via unsupervised solving* (ICARUS) is a generic, platformindependent game solver written in *Lua* [2] and optimized for the *Visionaire Game Engine* [6]. ICARUS was developed at Daedalic Entertainment, a leading company in the development and publishing of adventure games. Hence, it is primarily focused on solving the main functionality and riddles of adventure games. However, the solver follows a more generic design rationale, allowing for the integration of many meaningful types of game actions that can be adapted to any similarly traversable game genre, since the solver system interacts with the game environment using the same commands as a human player would. In order to facilitate human supervision, the ability to start, stop and play in the meantime, as well as for the most accurate game representation and bug reproducibility, the games are played in real-time. However, soft acceleration methods, such as character speed modification or skipping dialog texts, menus, videos, etc. can be turned on and off at run-time via the *web control panel*. In comparison to the existing

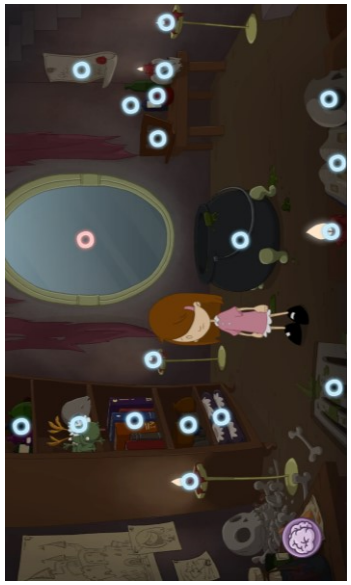

Figure 2. Example scene of the game *Anna's Quest*, containing 18 target objects, indicated by blue rings (for illustration purposes only). On each target, the actions of Table 2 can be applied.

approaches mentioned before, ICARUS can not only record performance metrics (FPS, RAM, CPU usage etc.) at single points of time, but it can track these measurements continuously over the whole span of a game iteration, recognizing crucial performance issues and pinning them down to concrete game situations and hardware constellations. For these iterations, it is not constrained to pre-determined sequences or recorded playthroughs, but it will dynamically explore the game state regardless if knowledge about the current situation is already given or not, using the solving process explained in the following section. The persistently learning nature of this setup allows ICARUS to combine the advantages of complete action testing and fast playthroughs, since it will start with a broad, explorative search over all possibilities of the game state and improve itself (in terms of number of executions per playthrough, thus also speed) with each further game iteration it traverses.

Solving Process

In most adventures, the actions that lead to progress are well-defined, generally consisting of (a) interacting with objects or characters, (b) collecting items, (c) combining items with other items, objects or characters, and (d) choosing from dialog options. Thus, as long as the acting character is not busy executing an action, ICARUS comes up with a representation of the game state by collecting the set of possible actions (Table 2) and stores it temporarily in a list of *currentActions*.

On these current actions, ICARUS remembers possible reward outcomes from previous choices that are stored in *currentRewards* $\in Z^{a\times4}$, the matrix mapping observed actions to reward values (where 0 is assigned to unobserved actions), which is a subset of *allRewards* $\in Z^{b\times4}$, containing short-term as well as long-term reward information (b being the amount of all actions observed in this and all previous game iterations in total, in 4 information dimensions about the action type, target, used item and reward).

Action selection

To choose an action, ICARUS performs a (random if configured to function probabilistically, consecutive if configured to use complete action iteration) selection of *maxCurrentRewards* $\subset$ *currentRewards*, which contains only the actions that yield the highest rewards among currentRewards. After the selection, ICARUS executes the corresponding action (e.g. a left click on a target T), waits until the completion of the action and evaluates the reward.

Reward learning

If the chosen action led to game progress (e.g. the inventory state changed, a quest progressed, targets appeared/disappeared, access to new areas is opened, etc.), a configurable, positive number is remembered persistently for this action in the long- and short-term memory. In general, a given state change can be considered positive if it is unrepeatable and leads to the enabling of formerly unavailable game actions. If no observable change happened, ICARUS punishes the last action by setting the action reward in the short-term memory to the respective punishment parameter that can equally be configured per action category. These configurations can take place in the script itself or via the *web control panel* at run-time, with +1 as the default for every positive game state change and -1 as the default for every action type in any other case. In

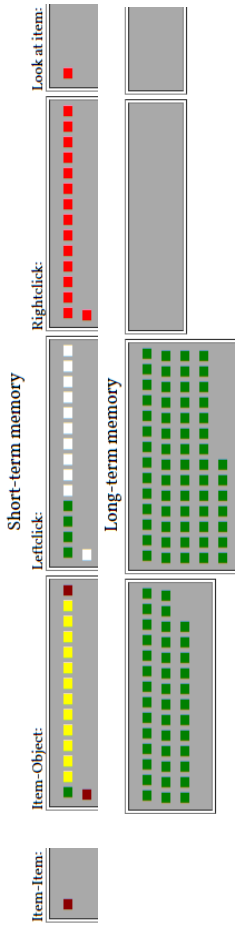

Figure 3. Reward map in a game state containing 1 item and 15 possible action targets, visualized in the *web control panel*.

that way, actions that rarely contribute to progress (e.g. looking at items) can be punished harder than important actions (e.g. left click). Once the set of current actions is iterated, i.e. every possible action has a negative temporary reward, the short-term reward map is *soft-reset (see: Soft-resetting the reward map)*.

The segmentation into short- and long-term memory is important since the completion performance increases with *short-term memory action selection* and the *final ICARUS action selection* as illustrated in Figure 7, which is mainly caused by the inclusion of long-term rewards. Entries of the long-term reward map are loaded whenever the respective action is currently available and thus strongly determine the sequence of the action selection. Nevertheless, the short-term memory is still needed, since the rewards for the actions in the long-term memory are learned from a very specific game state that has to be the same (or similar) to the current game state in order to yield actual game progress. If an action is remembered positively from the long-term reward, but the current game state yields no reward for this action, the respective punishment does not overwrite the positive reward in the long-term memory, but it will store a negative reward in the short-term memory, which ICARUS will use in the end for the action selection.

For example, ICARUS might record a positive reward if the game object *door* is opened, where the underlying game state had a precedent action that unlocked the door. In the next game iteration, ICARUS will remember the positive reward and prioritize attempting to open the door, even if it is still locked. The punishment reward will be recorded in the short-term memory and ICARUS will proceed with other actions (most likely the collection of a *key* and the unlocking action *key-with-door*) before it will reconsider opening the door. This reconsideration process is realized by the following *soft-resetting*.

Soft-resetting the reward map

In this process, every negative value of the map is increased by 1, so that -1 rewards result in 0 (“unobserved”) and even lower values are coming one step closer to a possible re-observation. That means that an action from an action type that is configured to be punished with -5 requires 5 soft-resets to be considered for execution again. If entries that have a positive long-term reward turn to 0 (“unobserved”) in this process, they are set to the respective original instead to ensure that ICARUS prefers the execution of them again, after the soft-reset. Figure 3 visualizes the reward map right after a soft-reset, where several actions containing long-term rewards are reset into positive values (green), some actions were punished harder and thus yielded a high negative reward (red), and some actions had a low negative reward which were reset to 0 in this step (white). In total, the technique of soft-resetting results in a normalization of the reward space, so that negatively rewarded actions have a chance to be executed again, but strictly after positively and new actions are tested.

Educated Guessing

The majority of adventure games are composed using puzzles that challenge the human power of deduction and creative combination by demanding the correct usage of items with object targets or other items. In theory, every possible *item-item* and *item-object* combination has to be considered in the process of action collection. However, most of these combinations

Figure 4. Example inventory containing 4 items, resulting in 16 combinations. Two actions are actually beneficial for game progress (X), two have also to be tested (?) and 12 can be discarded via educated guessing (X).

Figure 6. Inventory in the example game state of *Anna's Quest*. 11 items are held that can be combined with each other or used on the target objects of the scene (Figure 2).

are evoking only standard responses and the set of items that lead to progress in combination with each other or with objects is most often small in comparison to the possible set of all combinations.

For example, an inventory containing a *red key*, a *red box*, a *book* and a *blue box* already has 16 possible item-item combinations, where only the combination *red key-with-red box* (or vice versa) leads to actual game progress (see: Figure 4). Educated guessing (comparable to pruning a search space in classical AI search) is the process of discarding options that do not make sense to evaluate in the first place, e.g. combining each item with itself, combining the *book* to any of the items or trying to open one of the boxes with another box. It will leave *red key-with-red box* and *red key-with-blue box* as possible actions, even if the latter won't have any positive effect.

The decision why *red key-with-blue box* is still considered an action that could yield reward and e.g. book-with-blue box is not, is made on the type of response that the respective action evokes. *red key-with-blue box* will trigger an evaluation about whether the key fits the box (or a comment that is precisely defined for this situation, e.g. "This key doesn't fit."), where book-with-blue box will only trigger a generic standard response, e.g. "This doesn't work". Once this generic response is triggered and it's execution therefore part of a test run, it does not need to be triggered by the remaining zero-effect item combinations again, and the solving process can thus be sped up using educated guessing without sacrificing the validity and reliability of a test run. This distinction between action types works by assessing engine information about the particular action, which cannot explicitly tell the reward or the outcome, but is able to discard purely cosmetic or commentary actions that often trigger random default phrases. Every action that falls under certain categories, invokes standard functions, is tagged with default codes, or can be parsed for yielding no meaningful game progress can thus be strictly excluded from repeated executions in order to drastically reduce the amount of combinations for active execution checks.

In a real world example, the combinatorial complexity can become very severe. Figure 5 displays the extent of combinations of a game state containing 11 available items (Figure 6) and 18 available targets (as in Figure 2). Having many items in the inventory leads to exponential growth of the number of possible actions, which becomes even worse in scenes with many objects. This is where educated guesses come into play in order to avoid exponential growth in execution times that can cause considerable costs even with automated testing. As in the examples mentioned before, red cubes in the figures represent actions that were already tested and only yielded negative reward. However, a great portion of these actions (marked in dark red) will never be chosen, since ICARUS discards them using educated guessing. The resulting subset of item-item combinations still contains a number of combinations that do not result in immediate game progress or only display (specifically chosen) comments. However, in this example the method of educated guessing reduced the search space from 121 possible combinations to 35, i.e. by over 70%. The same 11 items could also be used on 18 different object targets in this scene, but over 80% do not require active execution checking for game progress, since they are already filtered out by educated guessing. However, when fully exhaustive exploratory game testing is required, developers and testers can disable this filter at run-time.

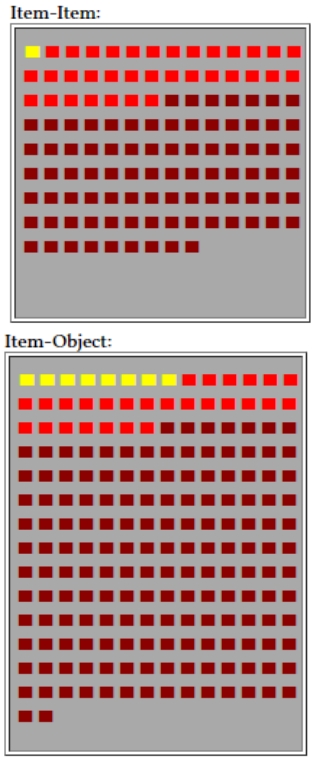

Figure 5. Example reward map subset in a scene of the game *Anna's Quest*. Yellow/white actions are unobserved, red actions were tried out and yielded negative reward. The majority of actions however are discarded by educated guessing (dark red), since they are classified as yielding no reward beforehand.

Dialog

The dialog choice system differs systematically from the previously mentioned actions. Dialogs are temporary, usually occur only in situations in which no other actions are possible and the set of options is very limited, namely to the number of dialog alternatives that are available at any give step. In each frame on the main loop, ICARUS assesses whether the game state is in a dialog or not before choosing an action or a dialog part. If the game is in a dialog state, a dialog option is chosen at random or in a traversing fashion, depending on the configuration.

Hints

Before ICARUS selects an action, but after observing all scene targets and inventory items, it will check if the situation fits to one of the hints (scripted actions) that can be manually defined in the configuration. Each of the two action archetypes (dialogs and actions) has its own hint table. E.g., *MGHints* contains walkthrough-like actions for puzzles that have an extremely low probability of being solved without context-sensitive, graphical, or time-dependent comprehension. ICARUS will favor an action above all else if the situation at hand matches the situation specified in a *MGHint* with the following general structure:

The current scene name is exactly "*SCENE*".
The target "*TARGET_OBJECT*" exists.
All conditions in the table "*CONDIS*" are met. No condition in the table "*NEG_CONDIS*" is true.
All values of "*LIST OF VALUE-CONSTRAINTS*" are met.
All items of "*LIST OF NECESSARY ITEMS*" are held.
All items of "*LIST OF FORBIDDEN ITEMS*" are not held.

Using this mechanism, in extreme cases, a complete game iteration can be executed deterministically by a hard-coded sequence of hints, since ICARUS will not explore the remaining game actions as long as the current situation fits to the execution of a hint. This can help with troubleshooting fixed processes as every game iteration has the same action sequence (e.g. in system compatibility or performance testing).

Web Control Panel

Since ICARUS is written completely in the script language *Lua* that is implemented by the engine, it can be applied to games running from the game engine editor as well as to complete builds, without the need of external programs or tools. However, to achieve more comfortable control over the most important parameters at run-time, to visualize the technical view of target objects, items, and possible item-item or item-object combinations, to provide a current snapshot of the debug log, and to provide access to the complete shortand long-term memory reward map, the system comes with a *web control panel* (see Figures 3, 5, and the supplementary video figure). It is implemented running a local web server on the testing machine, which can be assessed while the same computer is actively testing or remotely, to simplify the observation, control, and management of multiple testing iterations on several machines.

Completion time

Figure 7 depicts the time different agents required for one game iteration of the game *The Pillars of the Earth* that is currently under development. The complete and random action selection versions of ICARUS which do not include *educated guessing* or *reinforcement learning* serve as a baseline for the more elaborate solving

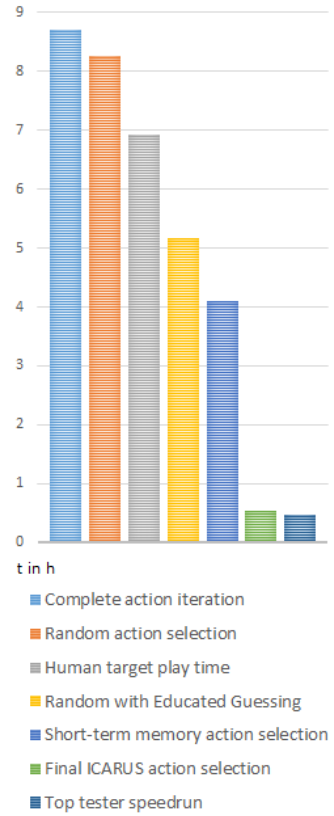

Figure 7. Completion times of the game *The Pillars of the Earth* by ICARUS employing several different features and approaches, compared to human playthroughs.

algorithms. They complete the game in about the time a human player needs who has no prior experience with the game, its puzzles, and is exposed to the content for the first time. The heuristic filtering of *educated guessing* cuts away about 40 % of the time required, whereas using *educated guessing* together with *short-term reinforcement learning* cuts the time required for completing the game by more than than half. Combining all of the introduced features (*educated guessing*, both *short-term* and *long-term reinforcement learning*, and *hints*), ICARUS can achieve a playthrough of the game in about 30 minutes, which is on the same level as the fastest speedruns of expert QA testers with prior experience of the game. Given that the game actions are supposed to be carried out at real-time, in an in-game, situated manner, these completion times can be considered near optimal and they are well suited for regular application.

Performance Tracking

In order to assess performance data about the game while playing, ICARUS can be configured to log the usage of RAM, VRAM, the time required to render the last frame (see: Figure 8), and further information of arbitrary kind into a persistent csv file. This tracking can take place simultaneously or independently from the basic solving process. If enabled, ICARUS will record one entry of performance data per frame, but save only the most extreme values of a given time window. In practice, each of the entries contains a time stamp, the name of the current scene, the current chapter, the last action and target that were chosen from ICARUS, and performance data: the time needed to render this frame in ms, as well as the amount of RAM and VRAM used in this frame in MB, before they get chunked to only the most extreme values, segmenting in steps of 1000ms.

This implementation of performance tracking is novel in the sense that it is integrated in the process of automated solving, as well as being able to detect and report critical performance issues immediately (e.g. significantly high frame load time, or exhaust of available RAM/VRAM) in continuous comparison of the same game situation over many different hardware constellations and development versions, which proved to be of great use in its first application during the development of *The Pillars of the Earth*.

Bug detection

Returning to the initial issue of bug detection, the ICARUS system can support the detection and reporting in all of the major bug categories:

A	Crashes/Freezes **Fully autonomous:** The tracking component of ICARUS will report immediately when the game crashes or stops rendering, thus it won't record any further data entries, displaying the exact time, scene and action that lead to the defect.
B	Blockers **Fully autonomous:** When a predefined progress timeout is defined (e.g. 5 minutes) or the action space is empty at a non-busy point of time, ICARUS can detect if it is stuck in a game state that can not proceed any further.
C	General **Semi-autonomous:** Aesthetic graphical, animation, sound, or spelling issues cannot be detected automatically using a logical solving algorithm. However the generic setup can be employed to play any adventure game, while human testers no longer have to concentrate on executing game actions.

Figure 8. An example result listing of frame time tracking of *The Pillars of the Earth*. Different scenes are distinguished by coloring. Frame time is recorded in ms.

They can instead focus on spotting bugs of all categories more closely, monitoring multiple game sessions that are being played automatically at the same time. Furthermore, the explorative nature of ICARUS leads to the execution of actions that are potentially undiscovered by the regular testing procedures, often because they seem to be not intuitive or promising to lead to game progress, while still potentially containing or causing bugs.

DISCUSSION AND SITUATED USE

ICARUS is currently used for continuous integration by daily build validation at Daedalic Entertainment for all new adventure titles. The application of ICARUS supports the development teams in staying up to date with recent game alterations and content implementation through integrated testing. If any complete feature updates are committed to the shared repository, ICARUS will automatically test the build provided from the internal game build server, reporting issues if necessary. ICARUS is even more frequently applied in the continuous QA processes and test runs, where it aids testers by reducing workload, using the semi-autonomous approach. Furthermore, ICARUS is employed in the gold mastering of finalized games to check for changes of traversability and performance after games are completed to a shippable version. Finally, even large-scale hardware compatibility tests that are using remote hardware can be executed through ICARUS, as the first, external test of the game *The Pillars of the Earth* on 61 different hardware constellations and platforms demonstrated successfully. This does not only help with determining minimum hardware requirements, but also provides general insights into the impact of game mechanics, graphics and scene staging across a large number of systems.

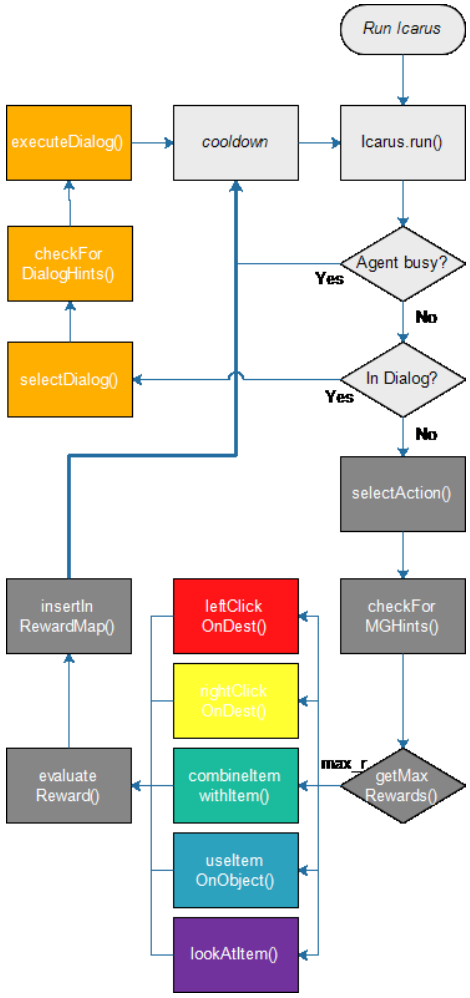

Figure 9. Flowchart of ICARUS

"Having a script like ICARUS that will simply try out different combinations and paths without getting exhausted or used to a specific way of playing the game helps focus QA resources on more complex tasks (testing of visuals and audio, game logic, etc.), making QA testing of the project more efficient and effective. Obviously, automated QA of parts or the entirety of a game can never completely replace human testers […]. However, it is definitely a valuable addition to our QA methods and helps to improve the overall quality of our games."

- Maik Hildebrandt, Head of QA at Daedalic Entertainment [12]

Limitations and Future Work

In order to widen the field of applications for ICARUS and to be no longer constrained to a single game engine, the system is currenlty being extended to support further game engine environments, namely Unreal Engine and Unity.

CONCLUSION

The ICARUS solver for adventure games has a proven track record as a significant enhancement for the quality assurance at Daedalic Entertainment. It can support developers and QA staff with tedious workflows, simplifying daily tasks and enabling performance comparisons across game iterations, game versions, and hardware constellations that might otherwise be prohibitively costly to execute. It can detect or aid the detection of all major bug categories, by either fully autonomous reporting or by allowing testers to focus closely on occurring bugs instead of being busy with executing game action sequences. The time needed for a complete game iteration is on the same level as professional game testers, thus no delays compared to prior development and QA processes are caused when using the system, while the time for implementing ICARUS in a completely new game project is also reasonable. Although some related work on automated non-technical testing solutions in games exists for clearly defined, template or macro-based scenarios, the generic nature, the ability to iterate through complete game iterations reliably, and the manifold features of tracking, visualizing and reporting, allow ICARUS to support game studios with establishing novel standards of QA, providing benefits to developers, publishers, and gamers alike.

REFERENCES

1. Bob Bates. 2004. "Game Design".
2. Retrieved February. 2017. "The programming language Lua". https://www.lua.org/
3. Fazeel Gareeboo and Christian Buhl. 2012. "Automated Testing: A Key Factor For Success In Video Game Development. Case Study And Lessons Learned". http://www.uploads.pnsqc.org/2012/papers/t-26_Gareeboo_paper.pdf EA Sports.
4. W.P. Judd and W.L. Heinz. 1997. "Universal automated training and testing software system". https://www.google.com/patents/US5602982 US Patent 5,602,982.
5. Mathieu Lachance. 2016. "How much people, time and money should QA take?". Retrieved March 29, 2017 from http://www.gamasutra.com/blogs/MathieuLachance/20160113/263446/How_much_people_time_and_money_should_QA_take_Part1.php.
6. Retrieved March. 2017. "Visionaire Studio". http://www.visionaire-studio.net/
7. Cyril Marlin. 2011. "Automated Testing: Building A Flexible Game Solver". http://www.gamasutra.com/view/feature/134893/automated_testing_building_a_.php
8. Jim Merrill. 2016. "Automated testing for League of Legends". https://engineering.riotgames.com/news/automated-testing-league-legends Riot Games.
9. K. Peterson, S. Behunin, and F. Graham. 2012. "Automated testing on multiple video game platforms".

https://www.google.com/patents/US20120204153 US Patent App. 13/020,959.

10. G.M. Pope, J.F. Stone, and J.A. Gregory. 1994. "Automated software testing system". https://www.google.com/patents/US5335342 US Patent 5,335,342.
11. U.H.H. Wild and M.I. Jabri. 1997. "System and method for automated testing and monitoring of software applications". https://www.google.com/patents/US5671351 US Patent 5,671,351.
12. Johannes Pfau. 2017. "Personal interview with Maik Hildebrandt"
13. B. Chan, J. Denzinger, D. Gates and K. Loose. 2004. "Evolutionary behavior testing of commercial computer games" In Evolutionary Computation, 2004. CEC2004. Congress on (Vol. 1, pp. 125-132). IEEE.
14. 14. Southey, Finnegan, Gang Xiao, Robert C. Holte, Mark Trommelen, and John W. Buchanan. 2005. "Semi-Automated Gameplay Analysis by Machine Learning." In AIIDE, pp. 123-128.
15. Alexander Zook et al.. 2014. "Automatic playtesting for game parameter tuning via active learning". Proceedings of the International Conference on the Foundations of Digital Games.
16. Martens, Chris. 2015. "Ceptre: A language for modeling generative interactive systems." In Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference.
17. Smith, Adam M., Mark J. Nelson, and Michael Mateas. 2009. "Computational Support for Play Testing Game Sketches." In AIIDE.
18. Osborn, Joseph Carter, April Grow, and Michael Mateas. 2013. "Modular Computational Critics for Games." In AIIDE.
19. Fernando de Mesentier Silva et al.. 2107. "AI as Evaluator: Search Driven Playtesting of Modern Board Games". Proceedings of the AAAI 2017 Workshop on What's Next for AI in Games.
20. Daniel J.Simons and Ronald A.Rensink. 2005. "Change blindness: past, present, and future". Trends in Cognitive Sciences, Volume 9, Issue 1 [p.16-20].