Automated Testing in Video Games
Challenges Faced and Solutions Presented

David Stevens

April 2020

1 Introduction

Wherever there is software, there are bugs. Writing code that functions correctly under all possible circumstances is a notoriously difficult task, as bugs can appear anywhere. Some of the most common types of bugs that affect every type of application are memory leaks, incorrect pointer usage, synchronization problems such as race conditions, and arithmetic or logical errors [19]. These issues are especially common in lower-level languages such as C or C++ [1] as they lack many abstractions that higher level languages provide.

The most powerful tool that the developer has to combat these bugs is rigorous software testing. While in conventional software this is often done in an automated manner through e.g. unit testing, video game development provide an interesting case study as the field frequently lacks automated testing [14].

This essay strives to explore what makes video game development different, and what strategies that are employed to overcome the difficulties that are unique to the field.

2 Challenges

This section will present some of the challenges that arise when attempting to apply conventional testing techniques within video game development.

2.1 State space

One of the most frequently cited reasons not to use automated testing within video game development is that the high degree of interactivity leads to an immense state space [14, 16, 13]. This is further amplified by the frequent use of multithreading in video games, leading to further complexity [4, 7]. Unit tests are based on setting up a particular state and ensuring that the functionality is correct, therefore exploring one part of the state space at a time. When the state space is large, it becomes unfeasible to write unit tests for everything.

2.2 Difficult reproducibility

A consequence of the previously mentioned large state space of video games is that it becomes difficult to reproduce bugs.

It is also difficult to quantify the interactivity between entities within video games. Gilbert [7] details how failure states are difficult to identify, as bugs can be as subtle as an AI character standing in the wrong place, or an animation not being aligned correctly with the actions on screen. Gilbert [7] and Murphy-Hill, Zimmermann, and Nagappan [14] also discuss the difficulty of identifying what caused a bug during gameplay. Bug reports are typically vague, and require much work to reproduce. Yarwood [21] goes as far as describing how user's bug reports occasionally are so difficult to reproduce that they simply wait for the bug to appear again with more information.

A difficulty in reproducing bugs directly leads to a difficulty in identifying the specific state that should be targeted in an automated test. If the exact circumstances that trigger a bug are not identified, the resulting tests that show the bug might rely on side effects. Changes to other systems may therefore cause the tests to break [14].

2.3 Non-determinism

The problem of reproducibility is complicated further by the frequent use of non-determinism in video games. Modern video games deliberately introduce non-determinism through the inclusion of randomized gameplay mechanics such as the behaviour of AI agents in the game [14]. However, they also include unintentionally non-deterministic behaviour due to threading or distributed computations [14]. Gilbert [7] describes a situation in which the persistent save data would become corrupted in a video game due to errors with multithreading in specific scenarios. Multithreaded behaviour is difficult to test in every field [9, 20], and video games are no exception.

2.4 Performance requirements

Video games have in many ways stricter performance requirements than many other types of software. The core of a video games structure is a centralized piece of code called a *game loop*, in which all logic is applied between the generation of each frame that is shown to the user [17]. Most games strive to reach a consistent frame rate of between 30 and 60 frames per second [17], meaning that all calculations have a deadline before which they must be complete. A desired frame rate of 60 frames per second equates to all logic being handled within approximately 16 milliseconds, lest the game starts stuttering [17].

Because of this, video game development places a heavy emphasis on performance. While the code's efficiency is one of the top priorities, it can be difficult to test. This is both due to the previously mentioned large state space, which can have a massive impact on the performance, but even more critically due to the large variance in hardware that the game should run on. Modern

games are often cross-platform, running with shared code on desktop computers, game consoles and on hand-held devices or phones [5]. Ensuring consistent frame rates across multiple resolutions and hardware configurations is critical, as the performance of the game has been shown to have a direct impact on the performance of the player [5].

However, writing performance intensive code requires avoiding unnecessary branching, in order to avoid memory accesses and branch prediction misses [16]. This makes it difficult to write the modularized code that would otherwise lend itself to better testing. Using functionally pure subroutines is often not an option either, as they require prohibitively expensive copying of data [4].

2.5 Unclear requirements

Video games differ from conventional software in that it is difficult to define what the correct behaviour is [11, 14]. This is primarily due to the difference in goals with the software. While conventional software is intended to provide a specific functionality, video games are intended to provide enjoyment and evoke emotions [11]. Bugs that exist in games could manifest in behaviour such as an enemy being defeated in fewer hits than the developer had intended, but this is not necessarily a problem as the true requirement is simply that the enemy is defeated in an enjoyable way [14].

Perhaps the most famous example of this is *rocket jumping*, a technique in multiple first-person shooter games consisting of using the blast from an explosion from a weapon such as a rocket launcher to launch the player forward. While the technique is often implemented intentionally in modern games, it was popularized in 1996's *Quake* where it was an unintended consequence[1] of how the physics engine functioned [10]. Oftentimes the bugs become features, as long as they are enjoyable. However, enjoyment is of course difficult to quantify which eliminates the use of assertions to catch this type of bug, and testing therefore becomes difficult to automate.

3 Strategies

Despite the numerous challenges that video games present, the industry has found novel ways to automate some of the testing process. This section will present some of the techniques that are employed.

3.1 Unit tests

While it oftentimes is difficult to unit test video games, there are components of video games which can be unit tested naturally. There are also components that can be rewritten in such a way that they facilitate testing.

[1] Which is a nice way to say bug.

Industry legend John Carmack[2] discusses the benefits of increased use of functional programming at length in an article republished by industry publication Gamasutra [4]. Carmack [4] argues for functional programming for the purpose of preventing bugs, but also for the purpose of writing testable code. Whenever a method is considered complicated, Carmack [4] suggests refactoring it in a functionally pure manner and adding unit tests to it. Even if it cannot be made entirely pure due to performance reasons, restricting the amount of side effects allows the developer to reap most of the benefits of pure functions without complete purity [4].

Unit testing can also naturally be incorporated into backend engine functions such as texture loading or memory management systems [7]. It is also natural to unit test common functionality such as procedural level generation, mathematical calculations and in-game hint systems [8].

3.2 Runtime monitoring

Many video game developers employ strategies based on monitoring the state of the software during execution. This can be done in multiple ways.

Assertions While not fully automated, video game developers make frequent use of assertions in developmental builds to fail fast when something goes awry [16]. This helps developers spot errors as soon as they happen, reducing the risk of side effects and easing reproducibility.

Assertions can also be formalized into temporal logic constraints. While this is not commonly practiced within the industry, an example of this is found in Varvaressos et al. [18] who define constraints for the gameplay mechanics of an open source reimplementation of *Super Mario World*. When the constraints are violated, appropriate logs can be generated and stored in a database. This method has also been extended to games in other genres by Varvaressos et al. [17], showing potential for more widespread use.

Simulated playthroughs Another common approach that is fully automated is the use of simulated playthroughs. The basic idea is to program a script or bot that plays the actual game, and compares the game state with what is expected. This is a powerful concept that can be implemented in a variety of ways to target a variety of errors.

Yarwood [21] describes how game studio *Frontier Developments* makes heavy use of simulated playthroughs. One example is automating tasks such as placing every type of object on the map in roller coaster building game *Planet Coaster* to ensure that every asset can be placed successfully without crashing [21]. Frontier also test that games can be built and started without crashing on all supported platforms and various hardware configurations [21]. However, they also use automated testing of more complex gameplay, such as flying a space

[2]Co-founder of *id Software*, makers of games such as *Doom* and *Quake*

ship to a station in the space exploration game *Elite Dangerous* and ensuring that both flight and the information presented works as it should [21].

Bécares, Valero, and Martín [2] present a novel approach where instead of tracking failed playthroughs through crash reports, successful human playthroughs of video game levels are traced and logged. By modelling the state of game using *Petri nets* the test suite can then use the games AI system in conjunction with the user's logged inputs to replay the successful playthrough and ensure that the modelled expected state still holds [2].

Gilbert [7] uses a playthrough technique that is similar to chaos engineering, by developing a bot that would randomly play the game, jumping from room to room and occasionally finding crash bugs. This was also helpful for finding silent build errors such as missing assets or bad packaging [7].

Artificial Intelligence One more recent development within the field of simulated playthroughs is the use of machine learning for automated testing. Reinforcement learning has been successfully applied in both commercial and academic situations [3, 13]. Both studies train agents to solve simple tasks, and measure program correctness not only based on if the agent was successful but also based on how long it took for the agent to learn an optimal strategy and the runtime performance of the software [3, 13]. This way, potential changes that unintentionally harm frame rate or make the game world more difficult to traverse can be detected in an automated manner [3].

Nantes, Brown, and Maire [15] present a framework for testing the interface layer of video games by utilizing computer vision. Their approach is to use an AI agent that navigates the game by replicating user actions that were tracked in an earlier version and checking for visual anomalies by comparing the rendered output with the logged output [15]. This can be used to for instance detect bugs with the rendering of shadows in the game world [15]. Applying computer vision in testing is a way to solve the grievances voiced by Gilbert [7], as the otherwise difficult to test graphical bugs become very naturally testable.

3.3 Data mining

Bugs can also be tracked and identified automatically through large scale analysis of human gameplay. Telemetry systems can be implemented to identify where performance suffers in a video game [6]. Drachen, El-Nasr, and Canossa [6] also mention a case where a bug with enemies that weren't supposed to be able to damage each other were doing so through analysis of logs.

Finally, Lin, Bezemer, and Hassan [12] present a method for automatically classifying gameplay videos posted by users online as containing bugs based on metadata such as video titles and keywords. They propose using a random forest classifier to classify videos from both YouTube and the video game distribution platform Steam based, and achieved accuracies that were significantly higher than a simple keyword search [12]. This is particularly significant as a video provides more information for reproducing the bug than a log.

4 Conclusions

The challenges faced in automated testing of video games provides a fascinating case study for anyone that is interested in more complex forms of software testing. Issues such as massive state spaces, non-determinism and strict performance requirements make it difficult to employ conventional automated testing strategies such as unit testing within the field, forcing the developers to investigate more elaborate testing strategies. Commonly, runtime monitoring is employed as the primary strategy through liberal use of assertions, but also through simulated playthroughs of the video games in question. Within academia, there has been a recent trend of employing machine learning to perform more powerful testing. While this trend has not become mainstream within the industry yet, it is a clear indication of the massive potential that exists for solving many of the challenges that have so far been preventing a more widespread adoption of automated testing. The coming years will therefore be an interesting time to follow developments within the field of video game testing, as the academical approaches start trickling down into the industry itself.

References

[1] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. "Are there language specific bug patterns? Results obtained from a case study using Mozilla". In: *2009 Fourth International Conference on Software Engineering Advances*. IEEE. 2009, pp. 210–215.

[2] Jennifer Hernández Bécares, Luis Costero Valero, and Pedro Pablo Gómez Martín. "An approach to automated videogame beta testing". In: *Entertainment Computing* 18 (2017), pp. 79–92.

[3] Matthew Bedder. *AI tools for automated game testing*. 2019. URL: `https://www.prowler.io/blog/ai-tools-for-automated-game-testing` (visited on 04/28/2020).

[4] John Carmack. *In-depth: Functional programming in C++*. Apr. 30, 2012. URL: `https://gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php` (visited on 03/26/2020).

[5] Mark Claypool and Kajal Claypool. "Perspectives, frame rates and resolutions: it's all in the game". In: *Proceedings of the 4th International Conference on Foundations of Digital Games*. 2009, pp. 42–49.

[6] Anders Drachen, Magy Seif El-Nasr, and Alessandro Canossa. *Game Analytics: Maximizing the Value of Player Data*. Springer, 2013, p. 510.

[7] Ron Gilbert. *Unit Testing Games*. 2018. URL: `https://grumpygamer.com/unit_testing_games` (visited on 04/28/2020).

[8] Francois Guibert. *Unit testing in video games*. 2017. URL: `https://www.frozax.com/blog/2017/06/unit-testing-in-video-games/` (visited on 04/28/2020).

[9] Vilas Jagannath et al. “Improved multithreaded unit testing”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 223–233.

[10] Lee Killough. *Doom Level History*. 2002. URL: https://web.archive.org/web/20131217024351/http://www.rome.ro/lee_killough/history/doomqna.shtml (visited on 04/26/2020).

[11] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. “What went wrong: a taxonomy of video game bugs”. In: *Proceedings of the fifth international conference on the foundations of digital games*. 2010, pp. 108–115.

[12] Dayi Lin, Cor-Paul Bezemer, and Ahmed E Hassan. “Identifying gameplay videos that exhibit bugs in computer games”. In: *Empirical Software Engineering* 24.6 (2019), pp. 4006–4033.

[13] D Loubos. “Automated testing in virtual worlds”. MA thesis. 2018.

[14] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. “Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?” In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 1–11.

[15] Alfredo Nantes, Ross Brown, and Frederic Maire. “A Framework for the Semi-Automatic Testing of Video Games.” In: *AIIDE*. 2008.

[16] Chris Parsons. *Why video game coders don’t use TDD, and why it matters*. 2015. URL: http://chrismdp.com/2015/03/why-games-coders-dont-use-tdd-and-why-it-matters/ (visited on 04/28/2020).

[17] Simon Varvaressos et al. “Automated bug finding in video games: A case study for runtime monitoring”. In: *Computers in Entertainment (CIE)* 15.1 (2017), pp. 1–28.

[18] Simon Varvaressos et al. “Runtime monitoring of temporal logic properties in a platform game”. In: *International Conference on Runtime Verification*. Springer. 2013, pp. 346–351.

[19] V Vipindeep and Pankaj Jalote. “List of common bugs and programming practices to avoid them”. In: *Electronic, March* (2005).

[20] Cheer-Sun Yang and Lori L Pollock. “The Challenges in Automated Testing of Multithreaded Programs’”. In: (1997).

[21] Jack Yarwood. *A look at how different-sized studios approach the challenges of QA*. Mar. 13, 2020. URL: https://gamasutra.com/view/news/359240/A_look_at_how_differentsized_studios_approach_the_challenges_of_QA.php (visited on 04/28/2020).