APPLIED INFORMATION TECHNOLOGY AND COMPUTER SCIENCE
e-ISSN: 2773-5141

AITCS

Vol. 7 No. 1 (2026) 596-607
https://publisher.uthm.edu.my/periodicals/index.php/aitcs

Generating Test Cases in Unity Engine: A Case Study of My Indie Game – Cube

Gavin Chong Jia Hao[1], Mohd Zanes Sahid[1]*

[1] *Fakulti Sains Komputer dan Teknologi Maklumat,*
Universiti Tun Hussein Onn Malaysia, Parit Raja, Batu Pahat, 86400, MALAYSIA

*Corresponding Author: zanes@uthm.edu.my
DOI: https://doi.org/10.30880/aitcs.2026.07.01.032

Article Info

Received: 15 June 2025
Accepted: 21 June 2026
Available online: 30 June 2026

Keywords

Automated Test Case Generation, Random Search-Based Testing, Model-Based Testing.

Abstract

The development of video games requires extensive testing to ensure functionality and player satisfaction. However, manual testing of game mechanics can be time-consuming and prone to errors. This project focuses on addressing the inefficiencies in manual testing by introducing automated test case generation techniques for game mechanics validation. Traditional manual testing approaches often fail to cover all possible game states and transitions, leading to undetected bugs and suboptimal gameplay experiences. To address this, the project employs two different automated test case generation methods, which are random search-based testing and model-based testing to evaluate game mechanics with maximized path exploration and path logging. The generated test cases will then be executed using Unity's Test Framework to verify their accuracy. The approach is applied to Cube, an indie 3D puzzle-platformer game where the player navigates through a cube-shaped environment to reach specific targets while avoiding obstacles and enemies. The results show that model-based testing is better than random-based testing in terms of execution time and test coverage. However, it is recommended to have hybrid approach by combining both methods to ensure higher defect detection efficiency. Additionally, this project demonstrates the applicability of automated testing methods in indie game development, setting a foundation for broader adoption in similar domains.

1. Introduction

The rapid growth of the game industry, driven by engines like Unity, has made indie game development more accessible. A 3D puzzle platformer, "Cube", is developed where players navigate a red cube through a 3x3 grid to reach a green cylinder while avoiding obstacles. With increasingly complex mechanics throughout the stages, manual testing becomes time-consuming, error-prone, and ineffective at covering edge cases [1]. Automated testing offers a more efficient, consistent, and scalable solution [2]. Since every automated test case generation methods have their own pros and cons [3], this project will compare two automated test case generation techniques—random search-based testing and model-based testing—to evaluate their efficiency, test coverage, and defect detection capabilities, aiming to determine the best approach for Unity game testing and ensure "Cube" is thoroughly validated with minimal defects.

Game testing differs from traditional software testing and requires additional considerations to ensure a smooth user experience [4]. Testing complex game assets, mechanics, and interactions is time-consuming, especially for indie developers with limited resources [5]. Manual testing often misses edge cases and introduces

human errors, leading to potential bugs that affect gameplay [5]. Automated testing addresses these challenges by generating and executing test cases to cover all possible scenarios [6]. While random search-based testing explores a broad range of scenarios, it lacks precision and efficiency [7]. In contrast, model-based testing systematically tests predefined paths but requires time to develop accurate models [8]. This project compares these two approaches to identify the most effective method for testing *"Cube"*, aiming to improve testing efficiency, reliability, and ensure the final game has minimal defects.

There are three main objectives that need to be achieved, which are: (I) To study about different automated test case generation methods, random search-based testing and model-based testing. (II) To apply the test case generated by different automated test case generation methods on the indie game, "Cube." And (III) To evaluate the effectiveness of the two test case generation techniques in detecting faults in the indie game – "Cube."

This project evaluates and compares two automated test case generation techniques—random search-based and model-based testing—in the context of my indie 3D puzzle platformer, "Cube." The focus is on black-box functional testing to ensure critical mechanics, such as player movement, enemy collisions, and block detection, work as intended. The expected outcome is to determine which method offers better test coverage, bug detection, and implementation efficiency, providing insights into improving game testing processes. This research highlights the significance of automated testing for indie games, addressing challenges like time consumption, human errors, and edge case omissions in manual testing. By streamlining test case generation and enabling reusable models, the findings can assist developers in enhancing game reliability and quality while offering potential applications in more complex game scenarios.

2. Related Work

Automated testing uses software tools to detect defects, validate performance, and ensure functionality in applications and games, addressing the limitations of traditional manual testing, which is inefficient for large-scale and complex games due to unpredictable player behavior and dynamic game states [4]. Game engines like Unity support automated testing through unit, integration, and system tests, enabling developers to verify mechanics without extensive manual effort [9]. Despite this, generating effective test cases remains challenging, leading to the adoption of techniques like random search-based and model-based testing, which improve coverage and efficiency while reducing manual intervention in game testing [6].

Random search-based testing generates test cases by randomly selecting inputs or parameters to create diverse scenarios. Its simplicity and low cost make it easy to implement, requiring no prior system knowledge [10]. This method is effective for covering edge cases and unexpected behaviors, especially in complex systems where modeling all possible behaviors is challenging [10]. However, its lack of structure can lead to inefficient test coverage, with some areas over-tested while others are neglected [3]. In Unity game development, random search testing explores player movements, AI actions, and environmental changes to identify bugs in mechanics or physics engines that manual testing might miss [10].

Model-based testing (MBT) uses abstract models, such as finite state machines (FSMs) or UML diagrams, to systematically generate test cases based on system behavior [11]. These models define states, transitions, and actions, ensuring structured and precise test coverage by validating expected behaviors in each state [12]. MBT excels at testing mechanics, AI behaviors, and interaction sequences in games, making it ideal for scenarios requiring systematic validation, such as combat systems modeled through FSMs [11]. Despite requiring significant effort to create and maintain models, MBT provides higher test coverage than random testing, identifying specific defects that random approaches might overlook [12].

Unity Engine is a versatile game development platform that supports automated testing through unit, integration, and system tests [9]. Its built-in testing framework enables developers to automate repetitive tasks, validate features, and check environmental criteria using third-party tools [4]. Unity's flexibility supports both random search-based and model-based testing, making it suitable for generating diverse test cases to assess player interactions, AI behaviors, and level progression [9]. However, Unity's tools alone may not provide sufficient coverage for dynamic and interactive games, requiring a combination of both testing approaches to address complex scenarios and improve overall quality [6].

This research has made some comparisons from the selected previous works, such as [7], [8] and [13] which discussed about the model based testing and random testing while using traditional testing as comparisons. The proposed research is going to compare the random search-based testing and model-based testing which have not been done in the previous research.

3. Methodology

This research uses the comparative research methodology to evaluate two different automated test case generation methods which are random search-based testing and model-based testing. Comparative research is particularly effective in software testing for validating techniques and assessing scalability, which is crucial as the game complexity increases [12]. For this research, test cases for both methods are defined with fixed parameters, and data are collected from game episodes. Metrics such as execution time, test coverage, and success rates are analyzed to compare the methods' efficiency, coverage, and reliability. Tests will be repeated to ensure consistency, and the project will involve setting up the game environment, applying both methods, and comparing their results across different aspects.

3.1 Random Search-Based Testing

Random testing has benefit of generating test case in low cost and its ease of automation [10]. The random-based testing approach generates test cases by randomly producing player movement sequences within the 3x3 cube environment, using inputs like W, A, S, and D. It is stateless, with each action independent of previous ones, simulating arbitrary movements until the player reaches the goal, encounters a collision, or exceeds the maximum moves. This exploratory method covers diverse paths, helping identify unexpected bugs or edge cases. Movement sequences are logged for analysis, allowing evaluation of test coverage and performance metrics.

3.2 Model-Based Testing

This research adopts the Unified Modeling Language (UML) state machine for implementing model-based testing in the 3D puzzle-platformer game, “Cube,” using UML class diagrams and state machines to model the player avatar's behavior and interactions with the environment as [13] do. A domain model is built to represent the conceptual aspects of the game [14], mapping game-specific elements to general concepts from a standard platform game profile stated in [13]. The Cube game domain model is shown in Fig. 1.

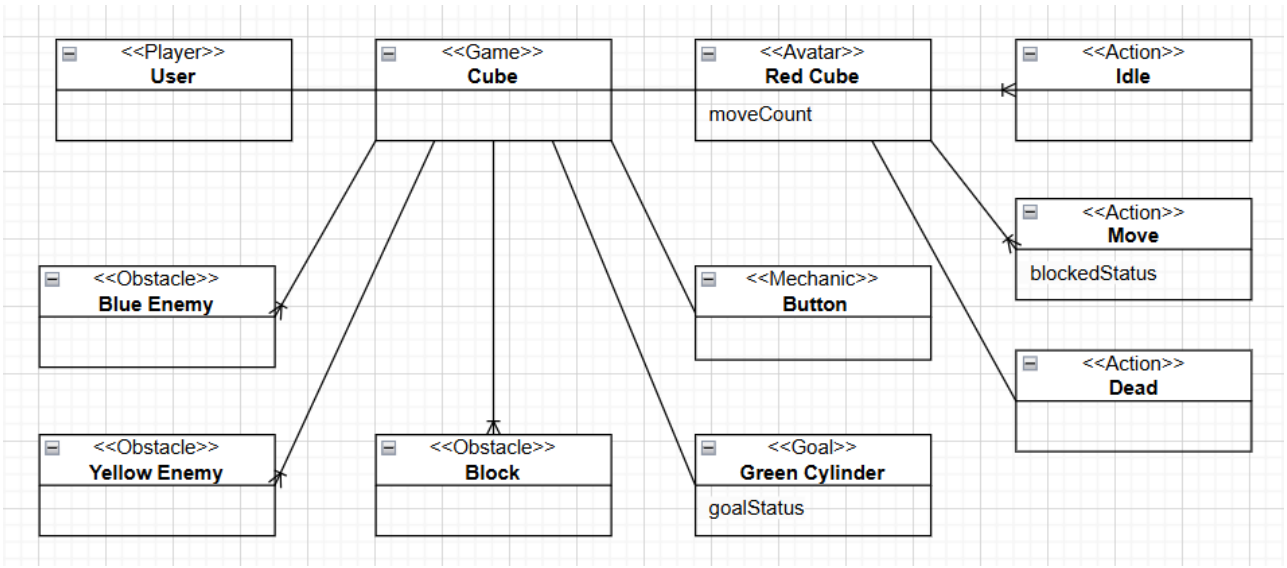

Fig. 1 *Cube Game Domain Model*

Besides that, the expected behaviors of the game are captured using UML state machine. State machines have been widely used in multiple fields for modeling runtime behavior [15]. An Avatar State Machine based on the game behavior and states is built and shown in Fig. 2. While for the movement state machine, since the game logic for movement in four direction which is move forward, move back, move left and move right is exactly the same, a move forward state machine as shown in Fig. 3. is built which can represent movement in all direction.

Applied Information Technology and Computer Science Vol. 7 No. 1 (2026) p. 596-607

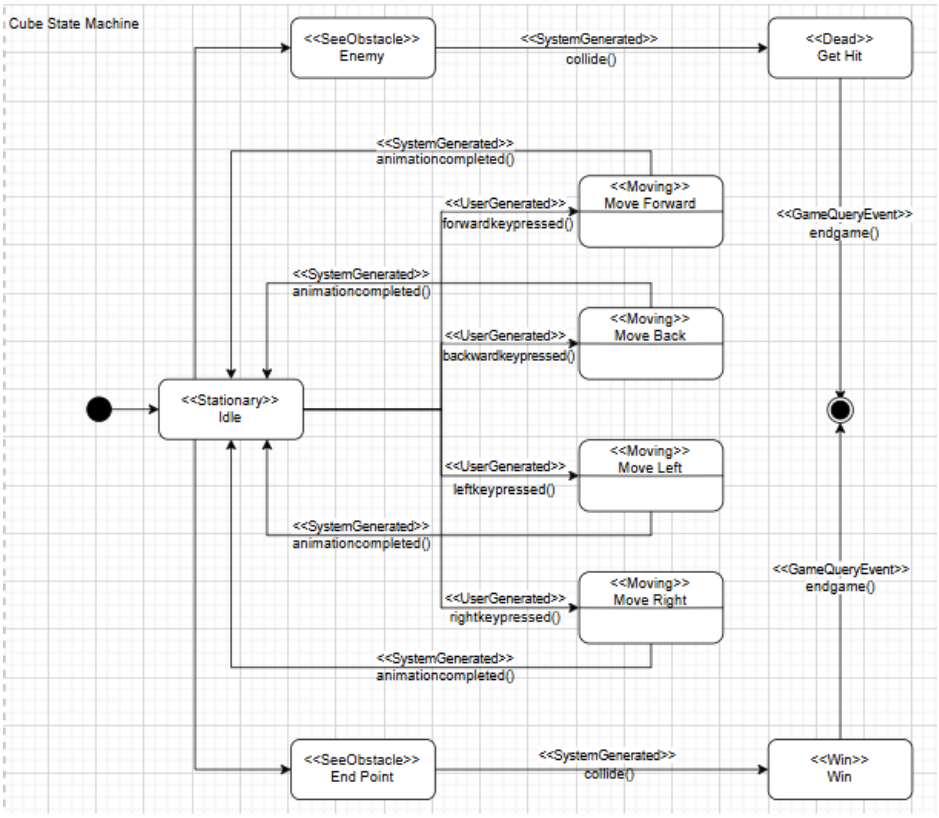

Fig. 2 *Cube's Avatar State Machine*

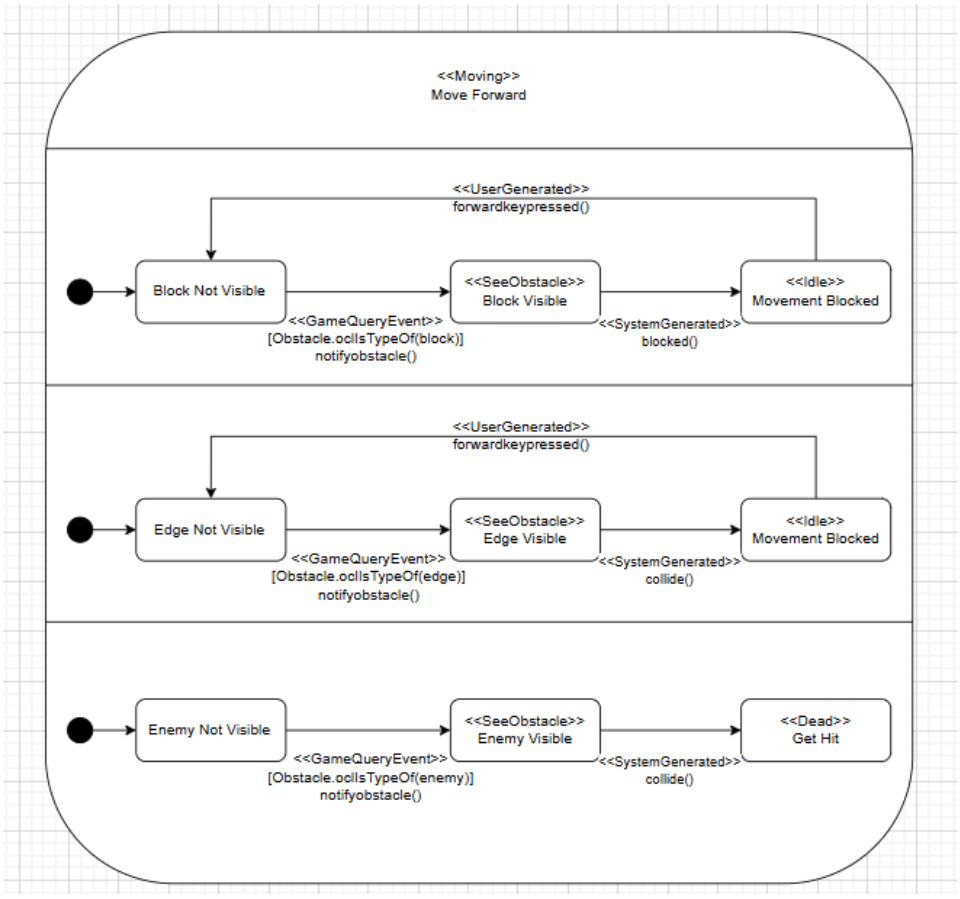

Fig. 3 *Move Forward State Machine*

4. Analysis

This research proposes the implementation and evaluation of two automated test case generation techniques—Random Search-Based Testing and Model-Based Testing (MBT)—to test the indie game "Cube." The methods focus on generating test cases efficiently, with Random Search simulating diverse player inputs to detect edge cases and MBT using UML state machines to model game states and transitions. The research includes data processing stages to ensure consistent and usable datasets, with both techniques implemented in Unity's testing framework. Performance metrics will be evaluated after that to compare the effectiveness of the approaches in defect detection.

Data processing stages are significant to increase the value of information and facilitate decision-making [16]. Two data processing stages are applied on the two approaches to normalise and make sure the data generated is clean. The data normalization is the practice of organizing data entries to ensure they appear similar across all fields and records [17]. Data normalization was performed by applying both testing methods under identical conditions. The starting positions of the player cube, target cylinder, and obstacles such as enemies and blocks were fixed across all test cases. This approach guarantees that the datasets generated are normalized and directly comparable. Additionally, data preprocessing included outlier management to filter unusable data [18]. Specifically, constraints were set for the player's maximum movements (25 moves) and execution time (20 seconds) for both methods. These parameters ensure that the generated datasets remain valid and within practical limits. Furthermore, for model-based testing using the depth-first search (DFS) algorithm, only paths leading to the target cylinder were logged, eliminating unnecessary paths that do not reach the goal. This refinement further improves the quality and relevance of the generated data.

Both approaches leverage Unity's built-in testing framework to execute test cases and log results, ensuring efficient evaluation of gameplay scenarios while balancing exploratory and systematic testing methods. The evaluation framework uses three key performance metrics—execution time, test coverage, and defect detection efficiency (DDE)—to measure the scalability, reliability, and accuracy of the proposed techniques. Execution time evaluates the speed and resource efficiency of test case execution [19], considering path traversal and decision-making processes [20]. Two equation is generated to calculate the average execution time (1) and total execution time (2) which is as below:

$$\mathrm{T}avg = \frac{\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{T}exec(\mathrm{i})}{\mathrm{N}} \quad (1)$$

Where:

N = Total number of test cases

$T_{\mathrm{exec}}(\mathrm{i})$ = Execution time for the i^{th} test case

$$\mathrm{T}total = \sum_{\mathrm{j}=1}^{\mathrm{M}} \mathrm{T}exec(\mathrm{j}) \quad (2)$$

Where:

M = Total number of test cases executed

Test coverage metrics are qualitative measures used to assess the completeness and effectiveness of the testing process, ensuring that all parts of a software application are thoroughly tested for quality and reliability [21]. It is often measured by using McCabe's Cyclomatic Complexity [22], focusing on path coverage to ensure that critical gameplay scenarios are tested. The formula for McCabe's Cyclomatic Complexity is as (3):

$$\mathrm{V(G)} = \mathrm{E} - \mathrm{N} + 2\mathrm{P} \quad (3)$$

Where:

E = Number of edges in the control flow graph

N = Number of vertices in the control flow graph

P = Program factors, such as independent modules or disconnected subgraphs

While DDE quantifies the effectiveness of each approach in detecting defects relative to total errors identified during testing [23]. These metrics provide comprehensive insights into the performance and reliability of the automated test case generation methods [24]. The defects detected can be either the player Cube moves out from the map, no die triggered when player collides with the enemy, no win triggered when reaching target cylinder and anything that is unexpected to happen. If no defect is detected for certain stage, the DDE will not be compared for that stage since both of the DDE will be 0%. The formula for DDE Is stated as below:

$$DDE = \left(\frac{\text{Defects found in phase}}{\text{Total defects}}\right) \times 100 \quad \text{.....(4)}$$

5. Result and Discussion

The experimental results and analysis comparing Model-Based Testing (MBT) and Random-Based Testing (RBT) in Unity for the indie game *Cube* are generated. The findings are structured based on three key metrics—execution time, test coverage, and defect detection efficiency (DDE)—across nine test stages (Stage1-1 to Stage1-9).

The stage 1-1 is a normal 3x3 map with player at left bottom and target cylinder at right top, while the player only need to move to the target cylinder to pass the stage. Stage 1-2 is similar with Stage 1-1, the only different is that there is an blue colored enemy moving in the middle of the map to disturb the movement of the player. Stage 1-3 contains one enemy move is a square shape and one blocked floor which is black in color in the map and cannot passed by player. The player have to bypass the enemy within the narrow path to pass this stage. Stage 1-4 introduce new mechanism called yellow enemy. It will always follow the movement of the player, and surely it will blocked by the air wall or blocked floor as well. Thus player have to trap the yellow enemy into a corner to be able to bypass them and pass the stage. Stage 1-5 introduce another mechanism named yellow floor which have to all be stepped to unlock the target cylinder. It has one extra mechanic that after the player stepped on it for the first time, they cannot move back to it again after it, which means that the floor becomes blocked floor. Thus player have to plan they path to pass the stage in one stroke. While for stage 1-6, it combines both new mechanism introduced just now which are the yellow enemy and yellow floor. It means that player have to step on all the yellow floor to activate the target cylinder and at the same time avoid collide with the yellow enemy, which is a challenge to the player. Stage 1-7 continue to introduce new mechanism named button, where the target cylinder will be surrounded by gate and player has to step on the button to remove the gate. While Stage 1-8 has three yellow floor in 3 corners where player has to step on all of them at the same time avoid the blue enemies patrol around the map. Lastly, we have 1-9 which is a big combination of the mechanisms introduced before, as a final stage for the player. Player have to solve the stage by stepping on both yellow floor and button to be able to reach the target cylinder, at the same time they have to avoid both yellow and blue enemy which is ready to kill them.

Both testing methods are applied on all of the nine stage separately, with model-based testing having a path generation script and path execution script while random-based testing just move the player randomly over the nine stage. The data is summarized in tables shown below. The analysis explores implications for game testing practices and aligns results with the study's objectives.

5.1 Tables

Table 1 *Comparison Table for Stage 1-1*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	4.96s	12.68s	MBT
Test Coverage	70.6%	58.8%	MBT
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

Table 2 *Comparison Table for Stage 1-2*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	3.36s	6.46s	MBT
Test Coverage	70.6%	5.9%	MBT
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

Table 3 *Comparison Table for Stage 1-3*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	3.43s	11.50s	MBT
Test Coverage	11.8%	5.9%	MBT
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

Table 4 *Comparison Table for Stage 1-4*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	3.77s	11.87s	MBT
Test Coverage	17.6%	41.2%	RBT
Defect Detection Efficiency (DDE)	66.7%	70%	RBT

Table 5 *Comparison Table for Stage 1-5*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	4.93s	14.38s	MBT
Test Coverage	18.2%	18.2%	Tie
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

Table 6 *Comparison Table for Stage 1-6*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	9.46s	19.20s	MBT
Test Coverage	72.2%	0%	MBT
Defect Detection Efficiency (DDE)	7.14%	10%	RBT

Table 7 *Comparison Table for Stage 1-7*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	9.11s	19.36s	MBT
Test Coverage	65%	0%	MBT
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

Applied Information Technology and Computer Science Vol. 7 No. 1 (2026) p. 596-607

Table 8 *Comparison Table for Stage 1-8*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	9.43s	18.28s	MBT
Test Coverage	100%	5.9%	MBT
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

Table 9 *Comparison Table for Stage 1-9*

Metric	Model-Based Testing	Random-Based Testing	Better Method
Avg. Execution Time	8.37s	19.41s	MBT
Test Coverage	54.5%	4.5%	MBT
Defect Detection Efficiency (DDE)	0% (no defects found)	0% (no defects found)	Tie (no defects)

5.2 Discussion

The experimental results demonstrate clear performance differences between Model-Based Testing (MBT) and Random-Based Testing (RBT) across all test stages. MBT consistently exhibited superior efficiency, with execution times 2-3 times faster than RBT in all scenarios (e.g., 4.96s vs. 12.68s in Stage 1-1). This advantage is attributed to MBT's structured approach, which systematically explores predefined game states without redundant operations.

In terms of test coverage, MBT achieved significantly higher rates in 7 out of 9 stages, including 100% coverage in Stage 1-8. This aligns with expectations, as MBT's methodical design ensures comprehensive traversal of critical game paths. However, RBT outperformed MBT in Stage 1-4 (41.2% vs. 17.6% coverage), suggesting that randomness can occasionally uncover untested scenarios missed by rigid models.

Defect Detection Efficiency (DDE) revealed mixed outcomes. While most stages reported 0% DDE for both methods—indicating either robust game code or limitations in test case design—RBT detected slightly more defects in Stage 1-4 (70% vs. 66.7%) and Stage 1-6 (10% vs. 7.14%). This implies that RBT's unpredictability may help identify edge-case bugs, though its overall reliability remains inconsistent due to 0% coverage in Stages 1-6 to 1-9.

Based on the observation from the result, MBT's high coverage doesn't always guarantee better bug detection, while RBT's sporadic successes highlight the value of unpredictability in testing. Besides that, performance gaps widened between both testing methods when more complicated mechanics is introduced in later stages, where MBT maintained dominance, suggesting RBT struggles with complex game states. It is suggested to have hybrid testing in future game testing method by combining MBT's efficiency with RBT's exploratory strengths to optimize both coverage and defect detection.

These findings underscore MBT's suitability as a primary testing method for indie games, while RBT could serve as a complementary tool for edge-case validation. Future work should address DDE limitations through enhanced test case design or AI-assisted model generation.

6. Conclusion

This study evaluated and compared random search-based and model-based testing techniques for automated test case generation in the 3D puzzle platformer Cube. Random search-based testing proved effective in exploring diverse scenarios due to its simplicity and low cost but lacked precision and systematic coverage. In contrast, model-based testing provided structured and comprehensive coverage by leveraging formal models, ensuring higher accuracy in validating game mechanics despite requiring more effort to implement. The findings suggest that combining both techniques can improve testing efficiency by addressing gaps in coverage and reliability, making them suitable for indie game development with limited resources. Future research should focus on optimizing these approaches, exploring scalability in dynamic game environments, and integrating AI-driven strategies to further enhance automation and efficiency in game testing.

Acknowledgement

The authors would like to thank the Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia for its support.

Conflict of Interest

Authors declare that there is no conflict of interests regarding the publication of the paper.

Author Contribution

This journal requires that all authors take public responsibility for the content of the work submitted for review. The contributions of all authors must be described in the following manner:

The authors confirm contribution to the paper as follows: ***study conception and design:*** *Gavin Chong Jia Hao, Mohd Zanes Bin Sahid;* ***data collection:*** *Gavin Chong Jia Hao;* ***analysis and interpretation of results:*** *Gavin Chong Jia Hao, Mohd Zanes Bin Sahid;* ***draft manuscript preparation:*** *Gavin Chong Jia Hao, Mohd Zanes Bin Sahid. All authors reviewed the results and approved the final version of the manuscript.*

References

[1] SureTest Inc. (2023). "Why Now Is the Perfect Time to Focus on Test Automation." https://suretest.health/blog/why-now-is-the-perfect-time-to-focus-on-test-automation/.

[2] WeTest, (2023). "The Benefits and Challenges of Test Automation in Game Development." https://www.wetest.net/blog/the-benefits-and-challenges-of-test-automation-in-game-development-656.html.

[3] Shayma Mustafa et al. (2014). "Systematic Mapping Study in Automatic Test Case Generation," ResearchGate. https://www.researchgate.net/publication/266143055_Systematic_Mapping_Study_in_Automatic_Test_Case_Generation

[4] Stephan Petzl. (2024). "Understanding the Unique Challenges of Game Testing," *Repeato*. Different strokes for athletic hearts. *The New York Times*, D4. https://www.repeato.app/understanding-the-unique-challenges-of-game-testing/

[5] Rohan Singh. (2022). "Game Testing Challenges And Essential Metrics," *HeadSpin*. https://www.headspin.io/blog/game-performance-metrics-that-matter

[6] Elly Johnson. (2024). "Top 5 Autonomous Testing Tools - 2024 Review," *Test Automation Tools*. https://testautomationtools.dev/autonomous-testing-tools/

[7] Nicklas Johansson. (2023). "A comparison between random testing and adaptive random testing." https://www.diva-portal.org/smash/get/diva2:1784320/FULLTEXT01.pdf

[8] Arthur Marques and Franklin Ramalho. (2014). "Comparing Model-Based Testing with Traditional Testing Strategies: An Empirical Study," *ResearchGate*. https://www.researchgate.net/publication/269304315_Comparing_Model-Based_Testing_with_Traditional_Testing_Strategies_An_Empirical_Study

[9] Asmo Jurvanen. (2023). "Automated Testing with Unity," *Theseus*. https://www.theseus.fi/bitstream/handle/10024/792842/Jurvanen_Asmo.pdf?sequence=3

[10] Andrew F. Tappenden and James Miller. (2009) "A Novel Evolutionary Approach for Adaptive Random Testing," *IEEE Xplore*. https://ieeexplore.ieee.org/document/5338642.

[11] Larry Apfelbaum and John Doyle. (1997). "Model-Based Testing," *CiteseerX*. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=381ed6dd3e695a38c93cba9a7265ed68a03caab2.

[12] Frank Esser and Rens Vliegenthart. (1997) "Comparative Research Methods," *Wiley Online Library*, pp. 1–22 https://doi.org/10.1002/9781118901731.iecrm0035.

[13] Sidra Iftikhar et al. (2015). "An automated model based testing approach for platform games," *IEEE Xplore*, 2015. https://ieeexplore.ieee.org/document/7338274/authors#authors

[14] Craig Larman. (2005). *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Pearson Education India.

[15] Muhammad Uzair Khan et al. (2014). "A Heuristic-Based Approach to Refactor Crosscutting Behaviors in UML State Machines", Software Maintenance and Evolution (ICSME) 2014 IEEE International Conference on, pp. 557-560. https://ieeexplore.ieee.org/document/6976138

[16] Astera Analytics Team. (2024). "What Is Data Processing? Definition and Stages", Astera. https://www.astera.com/knowledge-center/what-is-data-processing-definition-and-stages/

[17] Chrissy Kidd. (2024). "Data Normalization Explained: An In-Depth Guide", Splunk. https://www.splunk.com/en_us/blog/learn/data-normalization.html

[18] Kristie Sequitin. (2023). "What Is an Outlier?", Career Foundry. https://careerfoundry.com/en/blog/data-analytics/what-is-an-outlier/

[19] Paul Pocatilu. (2002). "Automated Software Testing Process" https://economyinformatics.ase.ro/content/EN2/pocatilu.pdf

[20] Sahar Tahvili et al. (2017). Towards Execution Time Prediction for Test Cases from Test Specification https://www.researchgate.net/publication/316969151_Towards_Execution_Time_Prediction_for_Test_Cases_from_Test_Specification

[21] Mehedi Hasan Shoab. (April 2024). Test Coverage Metrics: What is, Types & Examples https://www.practitest.com/resource-center/blog/test-coverage-metrics/

[22] GeeksforGeeks (Jan 2024). Path Testing in Software Engineering https://www.geeksforgeeks.org/path-testing-in-software-engineering/

[23] Tuskr. (2024). Defect Detection Efficiency https://tuskr.app/learn/defect-detection-efficiency

[24] Per Runeson et al. (2006). What Do We Know about Defect Detection Methods? https://www.researchgate.net/publication/3248375_What_Do_We_Know_about_Defect_Detection_Methods

Appendix A: Gantt Chart

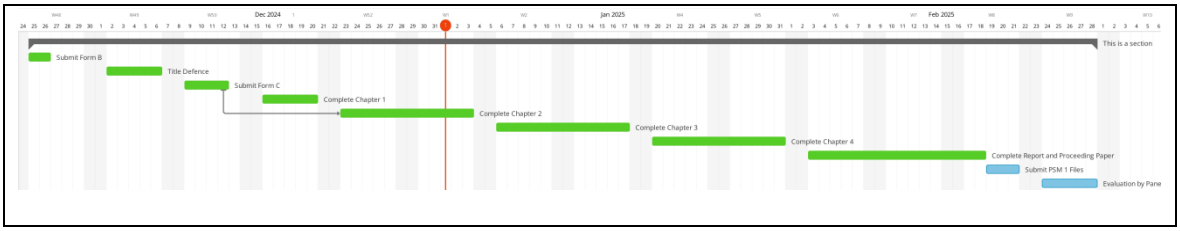

Figure A: Gantt Chart of Final Year Project 1

Appendix B: Code Segment for Test Cace Generation

```
[UnityTest]
    public IEnumerator RandomMovementTest()
    {
        Vector3 initialPosition = player.transform.position;
        logFile.WriteLine("Initial Player Position: " + initialPosition);

        for (int i = 0; i < maxMove; i++)
        {
            if (ReachTarget())
            {
                logFile.WriteLine("Reached target at: " +
target.transform.position);
```

```
                break;
            }

            MovementDirection direction = GetRandomDirection();
            logFile.WriteLine("Move #" + (i + 1) + ": " + direction);

            yield return SimulateMovement(direction);

            logFile.WriteLine("Player New Position: " +
player.transform.position);
            logFile.Flush();
        }

        // Add assertions as needed to check final state
        Assert.Pass();  // Placeholder, you can validate position or player
state here
    }
```

Figure B1: Random Testing Code Segment

```
private void DFS(Vector3 pos, string currentPath, int depth)
    {
        if (depth > maxDepth)
        {
            return;
        }

        // Check if the position is at the goal
        if (pos == goalPosition)
        {
            paths.Add(currentPath);  // Save valid path
            return;
        }

        if (pathPosition.Contains(pos))
        {
            blockposition.Add(pos);
            //visited.Clear();
        }

        // Mark the current position as visited
        visited.Add(pos);
```

```
        // Try moving in all possible directions (W, S, A, D)
        for (int i = 0; i < directions.Length; i++)
        {
            Vector3 newPosition = pos + directions[i];

            // Check if this new position is valid (inside map and not
visited)
            if (IsValidMove(newPosition))
            {
                // Continue DFS with the new position and updated path
                DFS(newPosition, currentPath + moveChars[i], depth + 1);
            }
        }

        visited.Remove(pos);
    }
```

Figure B2: Depth First Search Code Segment