Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

Performance Analysis of ECS Architecture in 2D Mobile Game Development: Ocean Hero

Raynaldi Irfansya Regar[1*], Benny Pinontoan[2], Christian A. J. Soewoeh[3]
[1,2,3]Information System Study Program, Mathematics Department, Faculty of Mathematics and Natural Sciences, Sam Ratulangi University, Indonesia
[1*]naldiregar.17@gmail.com, [2]bpinonto@yahoo.com, [3]christian.suwuh@unsrat.ac.id

Abstract: Mobile game development encounters performance bottlenecks when systems must simultaneously process large numbers of game objects on hardware-constrained Android devices. Conventional Object-Oriented Programming (OOP) produces elevated cache miss rates due to fragmented heap memory allocation. This research designs, implements, and evaluates an Entity Component System (ECS) architecture via Unity DOTS in a 2D Android educational arcade game titled Ocean Hero, developed using the Game Development Life Cycle (GDLC). ECS and OOP were comparatively evaluated on a Samsung Galaxy A15 4G through white-box testing and stress testing across six entity workloads from 500 to 3,000 entities. ECS maintained a stable 30 FPS throughout, while OOP degraded to 11 FPS and 90.73 ms frame time at 3,000 entities — a 172.7% performance improvement by ECS. These results confirm ECS as an effective architectural solution for scalable real-time mobile game development.

Keywords: Entity Component System; Object-Oriented Programming; Mobile Game; Stress Testing; Unity DOTS;

1. INTRODUCTION

The mobile gaming market has experienced substantial growth, with global revenue surpassing $100 billion and an installed base exceeding 2.5 billion active players worldwide [17]. Modern mobile games increasingly demand real-time management of hundreds of simultaneously active objects, dynamic difficulty escalation, and physics-based interactions — all within strict frame-rate constraints imposed by mid-range Android hardware. Mid-range processors such as the MediaTek Helio G99 class are constrained by limited L1/L2 cache sizes (typically 32 KB–2 MB), making memory access pattern efficiency a critical determinant of runtime game performance [11]. In this study, the game serves as a technical test bed: Ocean Hero is a 2D arcade game with a marine debris cleanup theme that integrates educational content while generating high numbers of dynamic entities during gameplay [1], [2].

Software architecture is a critical factor in mobile game development because the system must update rendering, input, collision, scoring, and object lifetime every frame while operating on limited mobile hardware. The conventional Object-Oriented Programming (OOP) paradigm encapsulates data and behavior inside objects and is useful for modular software construction. However, when many game objects are processed simultaneously, scattered object allocation and repeated per-object update calls increase memory access overhead and reduce CPU cache efficiency [11]. As the number of active

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267

entities increases, the likelihood of cache misses grows proportionally, resulting in degraded frame rates and reduced gameplay responsiveness.

The Entity Component System (ECS) architecture offers a data-oriented alternative by separating entities as unique identifiers, components as pure data containers, and systems as the exclusive holders of processing logic. The Entities package in Unity DOTS provides a practical implementation of ECS [12]. This approach stores entities with similar component compositions in contiguous memory blocks called chunks, enabling more efficient sequential iteration and improved data locality. Unity's Burst Compiler additionally compiles compatible C# code into highly optimized native CPU instructions [13]. Table 1 summarizes the principal architectural differences between OOP and ECS.

Table 1. Comparison between OOP and ECS architectures

Aspect	Object-Oriented Programming (OOP)	Entity Component System (ECS)
Paradigm core	Encapsulates data and behavior in objects.	Separates data in components from logic in systems.
Structure	Relies on class inheritance and object hierarchies.	Relies on composition through entity-component combinations.
Memory layout	Objects are stored in scattered heap memory.	Components are stored in contiguous chunks.
CPU efficiency	Random access can increase cache misses.	Data locality supports sequential processing.
Modularity	New features may require class hierarchy changes.	New features can be added using new components or systems.

On mobile hardware, the distinction between contiguous and scattered memory layouts has direct performance implications. Mid-range Android processors rely on small CPU caches (L1: 32–64 KB, L2: 256 KB–2 MB) to reduce access latency to main RAM. In OOP, each game object is allocated independently in heap memory at a non-adjacent address, so successive object updates require the CPU to fetch data from scattered locations — frequently causing cache misses that stall the pipeline. In ECS, components of the same type are stored contiguously in chunks, so iterating over thousands of entities of the same type requires far fewer cache-line fetches, directly translating to lower frame time and higher sustainable FPS on constrained mobile devices [11], [15].

Despite the theoretical advantages of ECS, empirical studies that directly compare ECS and OOP performance in 2D mobile games running on Unity DOTS and tested on actual Android devices remain scarce. Existing work by Hatledal et al. [5] demonstrated ECS advantages in simulation environments using a custom framework, but did not address real-time game contexts on mobile hardware. This gap motivates the present research.

From a practical standpoint, the architectural choice between ECS and OOP directly affects the gameplay experience on the mid-range devices that represent the majority of the Android installed base. A system that cannot sustain a minimum of 24 FPS will produce visually perceptible stuttering and impaired input responsiveness, reducing both player engagement and the educational effectiveness of serious games [7], [9]. Establishing an evidence-based architecture recommendation for this device class therefore has direct utility for mobile game developers working under hardware constraints.

The main contributions of this research are: (1) a hybrid ECS architectural design for a 2D mobile arcade game implemented via Unity DOTS; (2) a functional prototype tested on a real mid-range Android device (Samsung Galaxy A15 4G); (3) a comparative stress-test evaluation across six entity workloads from 500 to 3,000 entities with quantified performance improvement; and (4) an empirical benchmark demonstrating that ECS

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

sustains 172.7% higher frame rate than OOP at the maximum tested entity count on actual mobile hardware.

2. RESEARCH METHODOLOGY

This research adopted the Game Development Life Cycle (GDLC) as the development framework. GDLC was used because it provides a structured flow for game projects, from concept formulation to testing and release [3]. The initiation stage identified the main technical problem, namely performance bottlenecks in mobile arcade games with many dynamic objects. It also established Ocean Hero as an educational arcade game with a marine debris cleanup theme.

The pre-production stage produced the game concept, Game Design Document, asset requirements, and the ECS architectural blueprint. The architecture used a hybrid model: the user interface, such as joystick, score, timer, tutorial panels, and scene transitions, remained in the Unity GameObject layer, while the core gameplay logic was processed in the ECS World. The ECS World contains player, trash, obstacle, spawner, and game data entities. Components include PlayerInputComponent, ScoreComponent, GameDataComponent, MoveSpeedComponent, MoveDirectionComponent, TrashTag, ObstacleComponent, LifetimeComponent, and TrashBufferElement. The detailed visualization of this proposed hybrid architectural model is illustrated in Figure 1.

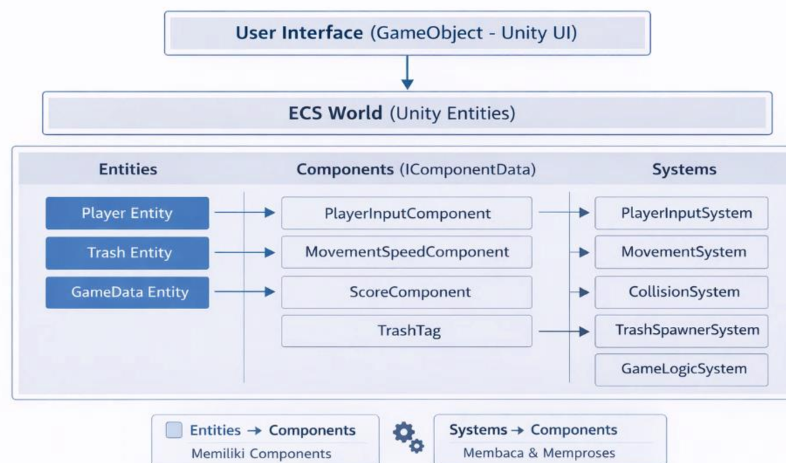

Figure 1. Hybrid Software Architecture Design

The production stage implemented the prototype in Unity 2022.3.x LTS using C#. Authoring scripts converted selected GameObjects into entities through the baking process. Component scripts were implemented as IComponentData structures, while system scripts processed runtime behavior. The implemented systems included PlayerInputSystem, MovementSystem, CollisionSystem, TrashSpawnerSystem, LifetimeSystem, GameLogicSystem, and DifficultySystem. The Entities and Burst packages were installed to support Unity DOTS and native-code optimization [12], [13].

Testing was conducted through white-box functional verification and performance-oriented stress testing. White-box testing was used to validate internal execution flow and system logic [8]. Stress testing was used to evaluate system stability and performance behavior under workloads exceeding normal operational conditions [10]. Unity Profiler was used to monitor runtime performance [14]. The stress-test scenario was structured as follows: the game was started from a clean state with an initial active entity count of approximately 10–50 objects (player entity, normal trash, and obstacle entities). A dedicated in-game debug button triggered the TrashSpawnerSystem, which instantiated

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

500 additional entities per activation directly into the ECS World. All spawned entities were rendered on-screen simultaneously with full scene rendering active — no culling or background loading was applied. The entity count was increased incrementally from 500 to a maximum of 3,000 entities across six steps, with each scenario observed over a 20-second tracking period and repeated three times to ensure reliability.

Table 2. Testing environment and stress scenario

Item	Specification
Device	Samsung Galaxy A15 4G
Processor	MediaTek Helio G99
RAM	8 GB
Operating system	Android
Display refresh rate	60 Hz
Game engine	Unity 2022.3.x LTS
Stress-test increment	500 entities per debug-button activation
Maximum tested entities	3000 entities
Number of repetitions	3 times per scenario
Rendering condition	Full scene rendering — all active entities visible on screen (no culling)
Measured metrics	FPS, frame time (ms), CPU usage (Unity Profiler)
Observation duration	20 seconds per scenario

To quantify the performance difference between the two architectures, a relative performance improvement metric was applied. According to Gregg [16], performance comparison between two systems should be conducted using quantitative benchmarking metrics that allow objective and reproducible measurement under identical workload conditions. The formula applied is:

$$\text{Performance Improvement (\%)} = (FPS_ECS - FPS_OOP) / FPS_OOP \times 100$$

Where FPS_ECS is the frame rate produced by the ECS architecture and FPS_OOP is the frame rate produced by the OOP architecture, which serves as the baseline reference.

3. RESULT AND DISCUSSIONS

During normal gameplay, the active entity count ranged from approximately 10 to 50 objects, including trash entities, obstacle entities, and the player entity. Player interaction used a virtual joystick, while score and timer information were displayed through the GameObject-based UI. Figure 3 shows the normal gameplay display, where the ECS systems processed movement, collision, spawning, lifetime deletion, game rules, and difficulty scaling every frame.

Figure 2. Ocean Hero Application Interface: Main Menu (left) and Tutorial Panel (right)

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

Figure 3. Normal Gameplay Display (ECS Architecture — 30 FPS, 33.3 ms)

White-box testing verified that all internal gameplay systems operated according to their designed logic. The PlayerMovementSystem correctly updated the player entity position from joystick input. The CollisionSystem destroyed trash entities and incremented the score by 10 points. The ObstacleCollisionSystem applied time and score penalties when appropriate. The EntitySpawnerSystem generated 500 entities simultaneously upon debug-mode activation. The LifetimeSystem correctly destroyed entities upon expiration, and the GameLogicSystem triggered win or loss states according to the target score and remaining time. All tested internal processes produced expected results.

The ECS stress test demonstrated stable performance across all six entity levels. Figure 4 shows the game environment during the benchmark mode at 3,000 entities. Despite the extreme entity count visible on screen, the system maintained a consistent 30 FPS with a frame time of 33.48 ms, confirming that the ECS architecture sustained maximum achievable performance without degradation.

Figure 4. ECS Stress Test — Benchmark Mode at 3,000 Entities (30 FPS, 33.48 ms)

The frame time variation across all ECS scenarios was less than 0.2 ms (from 33.29 ms at 500 entities to 33.48 ms at 3,000 entities), demonstrating negligible computational overhead scaling. Additional extreme spawning beyond the standard scenario reduced performance only slightly to approximately 26–30 FPS, which remained playable.

The OOP implementation showed a different pattern. At 500 to 1,500 entities, OOP maintained 30 FPS and approximately 33.3 ms frame time, comparable to ECS. Beyond this threshold, performance degraded substantially. At 2,000 entities, the frame rate dropped to 23 FPS with a frame time of 43.42 ms. At 2,500 entities, the frame rate declined further to 17 FPS and 58.50 ms. At 3,000 entities, OOP reached only 11 FPS with a frame time of 90.73 ms, causing severe loss of gameplay smoothness. The accelerating frame time increase — from +10 ms at the 1,500-to-2,000 step to +32 ms at the 2,500-to-3,000 step — indicates that the CPU was approaching saturation, causing non-linear performance collapse. Table 3 presents the full comparative results.

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

Table 3. Comparative stress-test performance results

Number of entities	Architecture	FPS	Frame time	Status
500	ECS	30 FPS	33.29 ms	Stable
500	OOP	30 FPS	33.33 ms	Stable
1000	ECS	30 FPS	33.31 ms	Stable
1000	OOP	30 FPS	33.29 ms	Stable
1500	ECS	30 FPS	33.35 ms	Stable
1500	OOP	30 FPS	33.31 ms	Stable
2000	ECS	30 FPS	33.40 ms	Stable
2000	OOP	23 FPS	43.42 ms	Degraded
2500	ECS	30 FPS	33.42 ms	Stable
2500	OOP	17 FPS	58.50 ms	Degraded
3000	ECS	30 FPS	33.48 ms	Stable
3000	OOP	11 FPS	90.73 ms	Degraded

To clearly illustrate the performance divergence between the two architectures, Figure 5 presents the FPS comparison graph across all six entity workloads.

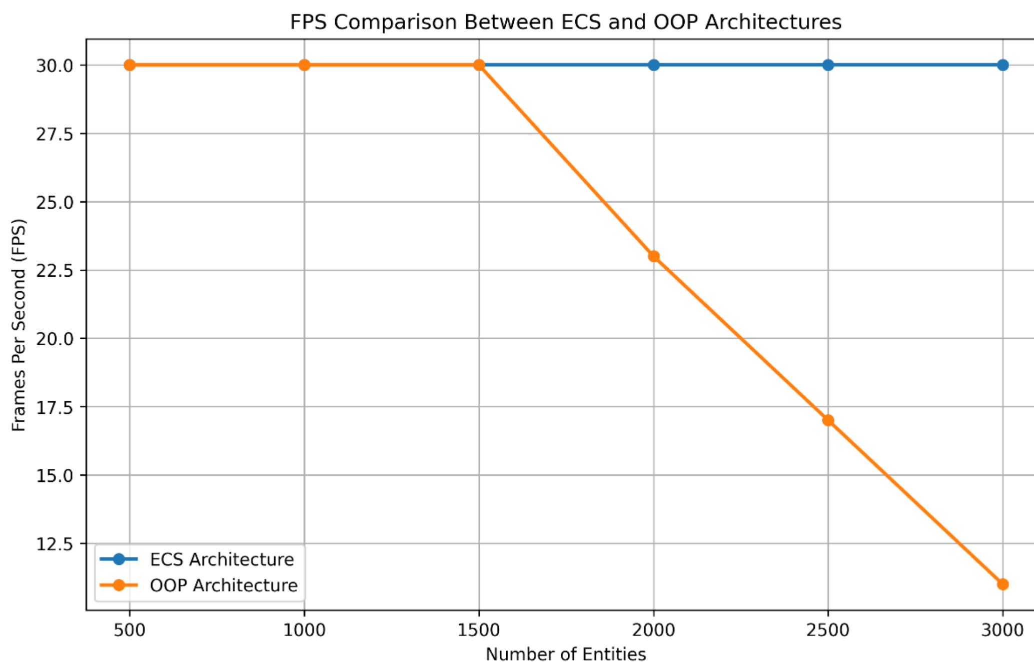

Figure 5. Frames Per Second (FPS) Comparison Graph Between ECS and OOP Across Six Workloads

At the maximum workload of 3,000 entities, ECS maintained 30 FPS while OOP recorded 11 FPS. Applying the relative performance improvement formula [16]:

$$(30 - 11) / 11 \times 100 = 172.7\%$$

The ECS implementation achieved approximately 172.7% higher runtime performance than OOP under the maximum stress-test workload. This result is consistent with the theoretical advantage of ECS: systems iterate over component data in batches stored in contiguous memory rather than executing many independent per-object update calls. The reduced memory fragmentation and improved data locality decrease CPU overhead under

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

large entity workloads [11], [15]. Unity's Burst Compiler further amplifies this advantage by compiling ECS system code into native machine code [13].

From a data-oriented design perspective [15], the performance gap between ECS and OOP can be explained through two mechanisms. First, cache locality: in ECS, all component data for 3,000 entities is stored as a contiguous array within a single chunk, so a single cache-line fetch loads sequential data covering approximately 8–16 entities. In OOP, each entity's data is embedded inside a MonoBehaviour object allocated at a random heap address, so the CPU must issue a separate memory request per object, resulting in cache miss rates that scale linearly with entity count [11]. Second, instruction throughput: the Burst Compiler translates ECS system code — which operates on struct arrays with no virtual dispatch — into SIMD-optimized native instructions, enabling the CPU to process multiple component values in parallel within a single clock cycle. MonoBehaviour Update() calls, by contrast, use virtual dispatch through the Unity scripting layer, adding per-frame overhead that accumulates with entity count [13]. These two mechanisms together explain why the OOP frame time increase accelerated non-linearly — remaining stable at 500–1,500 entities but rising by +32 ms at the 2,500-to-3,000 step — while ECS remained essentially flat throughout.

The CPU profiler data reinforced these findings. The ECS profiler graph remained relatively flat across all entity levels, while the OOP graph showed a steep and progressive increase in CPU workload, with irregular spikes at higher entity counts indicating frequent garbage collection and scattered memory access. These observations confirm that the OOP architecture's performance limitations are architectural rather than incidental.

To contextualize these results against prior work, Table 4 compares the findings of this research with related studies.

Table 4. Comparison with related research

Author (Year)	Focus	Platform / Tool	Key Outcome
Hatledal et al. (2021)	ECS in simulation systems	Custom framework	ECS improves modularity and data processing efficiency
This Research (2026)	ECS vs OOP in mobile arcade game	Unity DOTS + Samsung Galaxy A15 4G	ECS sustained 30 FPS at 3,000 entities; 172.7% improvement over OOP

As shown in Table 4, this research provides a direct empirical comparison of ECS and OOP on identical hardware, extending prior findings by Hatledal et al. [5] on ECS in simulation environments to the specific context of 2D real-time arcade games on actual mobile hardware. The results confirm that ECS is substantially more scalable than OOP for real-time 2D mobile games that dynamically spawn many similar objects. The hybrid implementation model is also confirmed as practical: the developer can retain conventional GameObject elements for UI while moving high-frequency gameplay logic to ECS, supporting both maintainability and performance.

4. CONCLUSION

This research successfully designed and implemented an ECS-based architecture for the 2D Android game Ocean Hero using the GDLC framework. The architecture separated identity, data, and logic into entities, components, and systems, producing a modular structure that is more suitable for high-volume entity processing than a conventional OOP approach. The hybrid Unity implementation allowed the GameObject-based UI and ECS-

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

based core gameplay to operate together without requiring a complete replacement of conventional Unity elements.

Functional verification confirmed that all internal systems operated according to their designed logic, including player movement, trash collection, obstacle penalties, score updates, entity spawning, and lifetime deletion. Comparative stress testing demonstrated that ECS maintained 30 FPS and frame times between 33.29 and 33.48 ms across all entity levels from 500 to 3,000, while OOP degraded to 11 FPS and 90.73 ms at 3,000 entities. Based on the relative performance improvement calculation, ECS achieved approximately 172.7% higher runtime performance than OOP under the highest tested workload.

These findings carry direct practical implications for mobile game developers. Indie developers and educational game studios targeting mid-range Android devices — which represent the majority of the global mobile gaming market — can adopt a hybrid ECS architecture to sustain playable frame rates even when scenes require hundreds to thousands of simultaneously active objects. The architecture documented in this research (Unity DOTS with a hybrid GameObject UI layer) provides a replicable blueprint that does not require full migration away from conventional Unity workflows, lowering the adoption barrier. For developers of arcade-style serious games specifically, where progressive difficulty increases entity density over time, ECS provides a scalable foundation that maintains gameplay responsiveness and, by extension, user engagement and learning continuity.

The limitation of this research is that the evaluation focused on technical performance and did not measure educational impact, usability, or player experience. Future research can extend this work with user testing, learning-effectiveness evaluation, broader hardware variation, and additional optimization analysis using memory profiling data.

5. ACKNOWLEDGMENT

The author expresses sincere gratitude to the Information System Study Program, Faculty of Mathematics and Natural Sciences, Sam Ratulangi University, for the institutional support provided throughout this research.

6. REFERENCES

[1] E. Adams, Fundamentals of Game Design, 3rd ed. Berkeley, CA, USA: New Riders, 2014.
[2] T. M. Connolly, E. A. Boyle, E. MacArthur, T. Hainey, and J. M. Boyle, "A systematic literature review of empirical evidence on computer games and serious games," Computers & Education, vol. 59, no. 2, pp. 661-686, 2012, doi: 10.1016/j.compedu.2012.03.004.
[3] F. Febriyanto, R. P. Sari, and S. Rahmayuda, "Implementasi Metode Game Development Life Cycle (GDLC) pada Perancangan Game Edukasi Pra Kemerdekaan Kalimantan Barat," Jurnal Teknologi Informasi, vol. 5, no. 3, 2024, doi: 10.46576/djtechno.
[4] M. Foxman, "United We Stand: Platforms, Tools and Innovation With the Unity Game Engine," Social Media + Society, vol. 5, no. 4, 2019, doi: 10.1177/2056305119880177.
[5] L. I. Hatledal, Y. Chu, A. Styve, and H. Zhang, "Vico: An entity-component-system based co-simulation framework," Simulation Modelling Practice and Theory, vol. 108, 2021, doi: 10.1016/j.simpat.2020.102243.
[6] B. Isik, G. E. Isik, and M. Zilka, "Game-based learning for industrial maintenance: A Unity 3D educational game of compressed air system training," Procedia Computer Science, vol. 253, pp. 784-793, 2025, doi: 10.1016/j.procs.2025.01.140.

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.

Journal of Artificial Intelligence and Technology Information (JAITI)
Website : https://ejournal.techcart-press.com/index.php/jaiti
E-ISSN 2985-6396
P-ISSN 2985-5306
Volume 4, Number 2, June 2026
https://doi.org/10.58602/jaiti.v4i2.267
Page 236-244

[7] Y. Li, D. Chen, and X. Deng, "The impact of digital educational games on student's motivation for learning," PLOS ONE, vol. 19, no. 1, 2024, doi: 10.1371/journal.pone.0294350.
[8] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 9th ed. New York, NY, USA: McGraw-Hill Education, 2020.
[9] L. Rodriguez-Calzada, M. Paredes-Velasco, and J. Urquiza-Fuentes, "The educational impact of a comprehensive serious game within the university setting," Heliyon, vol. 10, no. 16, 2024, doi: 10.1016/j.heliyon.2024.e35608.
[10] I. Sommerville, Software Engineering, 10th ed. Boston, MA, USA: Pearson, 2016.
[11] J. Gregory, Game Engine Architecture, 3rd ed. Boca Raton, FL, USA: CRC Press, 2018.
[12] Unity Technologies, "Entities overview | Entities 1.0.16," Unity Manual, 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html
[13] Unity Technologies, "Burst compiler | Burst 1.8.29," Unity Manual, 2026. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html
[14] Unity Technologies, "Profiler overview," Unity Manual 2022.3, 2026. [Online]. Available: https://docs.unity3d.com/2022.3/Documentation/Manual/Profiler.html
[15] Unity Technologies, "Introduction to the Data-Oriented Technology Stack for advanced Unity developers," Unity Resources, 2024. [Online]. Available: https://unity.com/resources/introduction-to-dots-ebook
[16] B. Gregg, Systems Performance: Enterprise and the Cloud, 2nd ed. Hoboken, NJ, USA: Pearson, 2020.
[17] Newzoo, Global Games Market Report 2024, Newzoo BV, 2024. [Online]. Available: https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2024-free-version

Raynaldi Irfansya Regar: *Corresponding Author
Copyright (c) 2026, Raynaldi Irfansya Regar, Benny Pinontoan, Christian A. J. Soewoeh.