Journal of Artificial Intelligence and Engineering Applications

Website: https://ioinformatic.org/

June 2026. Vol. 5. No. 3; e-ISSN: 2808-4519

Implementing the Procedural Generation Method for Placing Dynamic Objects in a Roblox-Based Adventure Game

Muhammad Hiszat[1]*, Hotler Manurung[2], I Gusti Prahmana[3]

[1,2,3]*STMIK Kaputama*

hiszattamvan21@gmail.com[1*], manurunghotler0@gmail.com[2], igustiprahmana4@gmail.com[3]

Abstract

Procedural generation is a content-creation technique that has become increasingly important in modern game development. However, on the Roblox platform, dynamic object placement still faces challenges such as overlapping, illogical positioning, and blocked navigation paths when relying solely on pure random methods. This research implements the Rule-Based Random Generation algorithm to manage the automatic placement of dynamic objects (enemies, treasure chests, and traps) in a Roblox-based adventure game. The proposed method combines randomization with constraint validation, including boundary check, overlap check using Euclidean distance, restricted zone check, and cross-type relational constraints. The system was developed in Roblox Studio with the Luau scripting language using a prototyping methodology and a modular architecture comprising ObjectSpawner, ConstraintValidator, SpatialGrid, and DungeonGenerator. Functional testing was conducted across 10 game sessions on a 1000 × 1000 studs map with a configuration of 340 enemies, 10 chests, and 50 traps. The results show that the system successfully placed all objects without any constraint violation, produced significant spatial variation between sessions (ranging from 86.31 to 2358.00 studs), and maintained level playability in every session. The average spawning execution time was 336.62 ms per session (0.84 ms per object), demonstrating the computational efficiency of the proposed method.

Keywords: *Dynamic object placement; Procedural generation; Roblox Studio; Rule-based generation; Spatial constraint*

1. Introduction

The development of the digital game industry continues to encourage the need for more diverse and engaging content to enhance player experience. One approach that has been widely adopted to address this need is procedural generation, a technique for automatically creating content through algorithmic processes without direct manual intervention. Several studies have examined the application of Procedural Content Generation (PCG) for level design and object placement. Whitehead [1] modeled dungeon layouts as a constraint-satisfaction problem in which every spatial element must satisfy a set of constraints, implemented using an SMT solver to ensure that all placement rules are simultaneously satisfied and produce a valid level structure. Pereira et al. [2] used an evolutionary algorithm to generate dungeon maps and locked-door missions, and reported that players considered the generated content as enjoyable as classical level designs while allowing broader exploration.

The use of constraints in level design is essential for maintaining playability. Modern procedural level generators implement design constraints precisely when placing objects such as puzzle elements, keys, and obstacles. This is necessary to ensure that objects are distributed logically so that levels avoid dead-ends and remain solvable, while also being aligned with the intended instructional goals of the game [3].

In the context of the Roblox platform, the use of procedural generative techniques for game content is gaining significant attention. Several popular adventure games on Roblox, such as Dead Rails and 99 Nights in the Forest, serve as inspiration for this research. Dead Rails features a railway track system that is dynamically formed, while 99 Nights in the Forest presents forest maps that vary on every run. Both games demonstrate how procedural object placement and environmental elements can enhance the uniqueness and replayability of gameplay.

Although procedural techniques on the Roblox platform have begun to evolve, most existing implementations focus on terrain or path generation. The main challenge arises when these techniques are applied to dynamic object placement inside dungeons, such as enemies, treasure chests, and traps. If object placement is performed manually, the development process becomes time-consuming and the resulting content tends to be repetitive. On the other hand, the use of pure randomization frequently introduces technical issues such as overlapping objects, illogical positioning, or even blocked player paths, making levels unsolvable.

To address these problems, a method that balances random variation with game-logic rules is required. The solution proposed in this study is the application of the Rule-Based Random Generation algorithm. This method allows the system to place objects in a varied manner while still adhering to specific constraints so that gameplay remains balanced and valid. Based on the urgency of producing efficient yet playable content variations, this research focuses on implementing the procedural generation method to handle dynamic object placement automatically on the Roblox platform.

Based on the background described above, the formulated objectives of this research are: (1) to apply the procedural generation method as an automatic dynamic object placement technique in Roblox-based adventure games; (2) to design and build the adventure game in Roblox Studio as the main development platform; and (3) to evaluate the application of procedural generation in dynamic object placement and assess the effectiveness and quality of the resulting system.

2. Literature Review

2.1. Related Work

Several previous studies relevant to the application of PCG and object placement in game development have been reviewed. Hidayat et al. [4] applied the Cellular Automata method to form dungeon levels in the *Dungeon Diver* game. Their functional testing showed that a fill percentage of 45% was optimal for producing playable rooms with very fast generation times (0.08–0.3 seconds). Whitehead [1] approached dungeon layout from a different angle, treating it as a constraint-satisfaction problem in which each room placement must satisfy a system of linear constraints. This approach is relevant for ensuring that rooms do not geometrically overlap.

Pereira et al. [2] developed dungeon maps and locked-door missions using an evolutionary algorithm. Their validation showed that procedurally generated content was rated by players as enjoyable as human-designed levels, an important indicator of PCG quality. Selviana and Lugata [5] used the Perlin Noise algorithm to generate terrain maps in Unity. Although effective at creating environmental variation, their method required additional validation through the Floyd–Warshall algorithm to ensure that distances between portals met certain conditions. The fundamental difference with the present study lies in the use of the Roblox Studio platform and a rule-based approach for object placement, rather than a noise-based approach for terrain.

Kim On et al. [6] compared pure randomization with rule-based randomization in the generation of items for role-playing games. Their results showed that the rule-based method produced a more balanced item distribution that better matched player difficulty. This principle is adopted in the present study, but applied to physical 3D objects in Roblox rather than to item attributes. Sousa and Oliveira [7] developed a PCG system using biome seeds to populate large game worlds, emphasizing the importance of parameterization to prevent overly chaotic results.

Based on the review above, research on procedural generation that specifically combines dynamic object placement (enemies and traps) with playability rules on the Roblox platform is still rare. Most existing studies focus on room structure or terrain using other engines. Therefore, this research aims to fill that gap by applying the rule-based random generation algorithm in Roblox Studio to create dungeons that are varied yet playable.

2.2. Rule-Based Random Generation

In the development of PCG systems, randomization is often used to automatically generate content variation. However, pure randomization may produce uncontrolled conditions such as imbalanced object distribution, position conflicts, or invalid game structures. Therefore, an approach that combines the flexibility of randomization with rule-based control is required [8].

The Rule-Based Random Generation algorithm is an approach that combines randomization processes with a set of logical rules to control the generation outcome. Unlike pure-random methods, the algorithm evaluates certain conditions before an object is placed or generated. The applied rules may include spatial constraints, inter-object relations, or game design constraints designed to maintain system consistency. This approach allows the system to produce content variation without sacrificing the validity of the game structure [6].

In the context of object placement, rule-based algorithms play a crucial role in maintaining playability. Uncontrolled object distribution risks blocking the player's navigation routes or creating configurations that cannot be solved at all. Kumaran et al. [3] emphasized that procedural content generation must integrate design constraints that act as safety parameters to ensure that the resulting layout remains logical and solvable. This principle aligns strongly with the rule-based approach used in this study.

2.3. Roblox Studio And Luau

Roblox Studio is an Integrated Development Environment (IDE) used to design, build, and manage games on the Roblox platform. Roblox Studio simultaneously functions as a game engine that provides facilities for creating three-dimensional environments, configuring object properties, and arranging game logic through a programming language called Luau, which is a modified version of Lua [9].

Architecturally, Roblox Studio organizes the development environment into several main services, such as *Workspace* for storing physical game objects, *ReplicatedStorage* as a storage medium for data and assets accessible by both server and client, and *ServerScriptService* for storing scripts with high security. This separation of services enables the decoupling of game logic on the server from visualization on the client, supporting stability and security of the game system [9].

Luau is a derivative of Lua 5.1 that has been performance-optimized and enhanced with gradual typing. These modifications were made to handle complex computations on the Roblox engine that involve millions of physical and data interactions in real time [9]. In the present

study, Luau is used to implement the rule-based random generation algorithm on the server side, allowing the object generation process to be controlled centrally and consistently.

3. Method

3.1. Research Method

The research method used in this study is the prototyping method, a system development approach that focuses on building an initial model (prototype) which is then iteratively tested and refined until it meets system requirements [10]. This method was selected because it provides an early picture of the implementation of the PCG algorithm in dynamic object placement, allowing direct evaluation of the system in the game environment.

The prototyping approach allows the researcher to perform incremental testing of the Rule-Based Random Generation algorithm. The result of each test is used as a basis for system improvement in the next iteration until an optimal configuration that meets the playability aspect is obtained. The stages of the prototyping method applied in this study consist of: (1) problem identification regarding object overlap, distribution imbalance, and blocked paths; (2) requirement analysis covering hardware, software, user, and algorithm needs; (3) prototype development that implements the algorithm in a basic form; (4) prototype testing focusing on constraint validation, including inter-object distance, restricted zones, and path validity; (5) evaluation and refinement by adjusting parameters such as MIN_DIST, object count, and constraint rules; and (6) final implementation as the production system.

3.2. Rule-Based Random Generation Algorithm

The Rule-Based Random Generation algorithm controls the placement of dynamic objects through a combination of randomization and rule evaluation. The algorithm operates iteratively, where each randomly generated candidate position is validated against several constraints before the corresponding object is allowed to be placed in the game environment. Table 1 summarizes the seven sequential steps of the algorithm.

Table 1: Rule-based random generation algorithm steps

No	Step	Formula	Implementation
1	System parameter initialization	Area = [x_min, x_max] × [z_min, z_max] MIN_DIST = d totalObject = n	Game area boundaries, total number of objects (n), and minimum distance between objects (d) are predefined as configuration constants in ServerScript.
2	Random coordinate generation	x = rand(x_min, x_max) z = rand(z_min, z_max)	Candidate position (x, z) is generated using math.random() in Luau within the area. The y coordinate is fixed to the terrain surface.
3	Boundary check	$x_{min} \leq x \leq x_{max}$ $z_{min} \leq z \leq z_{max}$	Generated coordinates must lie within the valid game area. Out-of-bound coordinates trigger regeneration.
4	Overlap check (Euclidean distance)	$dist(P_1, P_2) = \sqrt{((x_2 - x_1)^2 + (z_2 - z_1)^2)}$	Euclidean distance is computed between the new candidate and all placed objects. If the distance is below MIN_DIST, the candidate is rejected and a new coordinate is generated.
5	Restricted zone check	$P_{new} \notin Z_{restricted}$	Candidate position is checked against restricted zones (e.g., spawn point and exit point). Objects cannot be placed in these zones to preserve playability.
6	Object spawning	pos_valid = (x, y_fixed, z) Object.Position = pos_valid list ← list ∪ {pos_valid} i = i + 1	If all constraints are satisfied, the 3D model is cloned from ReplicatedStorage and placed at the valid position in Workspace.
7	Iteration and termination	while i ≤ totalObject do repeat steps 2–6 if attempts > MAX_ATTEMPT skip object i	The process repeats until the target number of objects is placed. A safeguard MAX_ATTEMPT prevents infinite loops in densely populated areas.

The core mechanism for preventing overlap lies in the Euclidean distance calculation in step 4. The distance between a new candidate position and every placed object is computed using the formula $d = \sqrt{((x_2 - x_1)^2 + (z_2 - z_1)^2)}$. If the resulting distance is smaller than the predefined MIN_DIST threshold, the candidate position is declared invalid and the system triggers regeneration. To maintain stable performance and prevent infinite loops when the game area becomes dense, the algorithm includes a MAX_ATTEMPT safeguard that skips an object after a maximum number of failed attempts.

3.3. Cross-Type Relational Constraints

In addition to the same-type distance constraint (MIN_DIST), each object is also subject to a relational constraint that regulates the minimum distance to objects of different types. This rule is designed to prevent unbalanced game conditions, such as enemies standing directly in front of treasure chests, or traps placed too close to chests so that the player has no chance to avoid both simultaneously. The matrix of relational constraints used in this study is presented in Table 2.

Table 2: Cross-type relational constraint matrix (in studs)

Source \ Target	Enemy	Treasure chest	Trap
Enemy	15 studs	18 studs	—
Treasure chest	18 studs	20 studs	8 studs
Trap	—	8 studs	10 studs

Based on Table 2, the strictest relational constraint is applied between Enemy and Treasure Chest, with a minimum distance of 18 studs in both directions. This rule ensures that the player is not immediately attacked by an enemy when attempting to open a treasure chest. The constraint between Treasure Chest and Trap is set to 8 studs to prevent both hazardous elements from being placed too close to one another. There is no relational constraint between Enemy and Trap because both function as independent obstacles.

3.4. System Architecture

The system was implemented in Roblox Studio with a modular architecture comprising several scripts. *MainScript* serves as the game-loop controller and is placed in *ServerScriptService*. *DungeonGenerator* is responsible for procedurally generating the terrain using a noise-based approach with two biomes (Forest and Plains). *ObjectSpawner* implements the seven steps of the rule-based random generation algorithm. *ConstraintValidator* encapsulates all constraint-validation logic, including boundary check, overlap check, restricted-zone check, and the cross-type constraint matrix. *SpatialGrid* is used as a performance optimization that reduces distance-search complexity from O(n) to approximately O(1) using a 20-stud grid cell size.

The configuration of each dynamic object type is defined in the CONFIG table inside *ObjectSpawner*, containing the parameters *total* (object count), *minDist* (minimum same-type distance), and area boundaries (areaX and areaZ). The spawning order follows the sequence Enemy → Chest → Trap. For each object type, the spawning loop generates a candidate position, calls *ConstraintValidator.validateAll()* to verify all constraints, and places the object only when the candidate is valid. If validation fails, a new coordinate is generated up to MAX_ATTEMPT = 50 times.

4. Result and Discussion

4.1. Functional Testing

Functional testing was conducted by running the system across 10 game sessions and observing the behavior of the rule-based random generation algorithm in each iteration. Each session was executed with a different random seed on a 1000 × 1000 studs map. The tested object configuration was 340 Enemy, 10 Chest, and 50 Trap. The aspects evaluated include placement success, the number of retry attempts required, and the presence or absence of constraint violations. The summary of functional testing across the 10 sessions is presented in Table 3.

Table 3: Functional testing results across 10 sessions

Run	Enemy placed	Chest placed	Trap placed	Boundary violation	Overlap violation	Status
1	340	10	50	0	0	**PASS**
2	340	10	50	0	0	**PASS**
3	340	10	50	0	0	**PASS**
4	340	10	50	0	0	**PASS**
5	340	10	50	0	0	**PASS**
6	340	10	50	0	0	**PASS**
7	340	10	50	0	0	**PASS**
8	340	10	50	0	0	**PASS**
9	340	10	50	0	0	**PASS**
10	340	10	50	0	0	**PASS**

Table 3 shows that across all 10 sessions, the system successfully placed every object according to its configuration (340 Enemy, 10 Chest, and 50 Trap) without any boundary or overlap violations. This confirms that the rule-based random generation algorithm operates correctly and consistently, regardless of the random seed used in each session.

4.2. Cross-Type Constraint Testing

To verify that the relational constraint matrix is correctly enforced, the actual minimum distance between every pair of dynamic objects was measured across the 10 sessions. The measurement results are summarized in Table 4.

Table 4: Inter-object distance verification (average of 10 sessions)

Object pair	Min constraint (studs)	Min observed (studs)	Avg distance (studs)	Status
Chest – Chest	500	501.14	1172.96	**PASS**
Trap – Trap	100	100.30	1032.83	**PASS**
Trap – Chest	100	100.13	1059.70	**PASS**
Enemy – Enemy	50	50.00	1041.17	**PASS**
Enemy – Chest	50	50.01	1062.83	**PASS**
Enemy – Trap	50	50.00	1035.56	**PASS**

Table 4 shows that all six object pairs satisfy the predefined minimum-distance constraint. The smallest margin is observed for Enemy–Enemy (exactly at the 50-stud boundary), while the largest margin is observed for Chest–Chest (1.14 studs above the minimum). This confirms that the *ConstraintValidator* module correctly enforces both same-type and cross-type relational constraints designed in Section 3.3.

4.3. Playability Testing

Playability testing was conducted to ensure that every level produced by the system can be completed by the player without structural obstacles such as blocked paths or objects covering access to the dungeon exit. Across all 10 sessions, the path from the spawn point to the exit was always open, the spawn area was always free from enemies within the predefined restricted zone (25 studs), and all treasure chests were reachable without structural obstruction. No placement failure exceeded the maximum attempt limit. These results confirm that the constraint mechanism preserves playability in every procedurally generated session.

4.4. Object Placement Variation

The application of procedural generation aims to produce non-repetitive object layouts in every session. As an evaluation parameter, the system recorded the coordinates of the first five objects of each type. The coordinates were then compared across three different game sessions (Run 1, Run 2, and Run 3). The Euclidean distance between corresponding objects was used to quantify the spatial difference between sessions. A summary of the variation per object type is presented in Table 5.

Table 5: Spatial variation summary across three sessions

Object type	MIN_DIST (studs)	Range of position differences (studs)	Result
Enemy	50	450.16 – 1514.15	All positions distinct
Treasure chest	500	86.31 – 2358.00	All positions distinct
Trap	100	522.67 – 1947.94	All positions distinct

Table 5 indicates that the placement of dynamic objects (Enemy, Chest, and Trap) changes substantially each time the game is launched. The differences in position between sessions span a wide range, from 86.31 to 2358.00 studs, indicating high spatial dispersion. Since none of the compared objects occupies an identical coordinate across the three sessions, the randomization mechanism is highly effective in producing non-repetitive content for each session and increases the replayability of the game.

4.5. Performance Evaluation

Performance evaluation was carried out to measure the computational efficiency of the implemented procedural generation system. Execution time was measured using the *tick()* function in Luau, called before and after each main process. The execution time results across 10 sessions are presented in Table 6.

Table 6: Object spawning execution time per session (in ms)

Run	Chest (ms)	Trap (ms)	Enemy (ms)	Total (ms)	Avg per object (ms)
1	3.57	2.15	196.77	202.49	0.5062
2	7.18	5.29	358.06	370.53	0.9263
3	3.22	2.51	167.57	173.30	0.4332
4	10.55	5.45	497.65	513.65	1.2841
5	4.95	3.58	324.77	333.30	0.8332
6	7.36	4.50	313.79	325.65	0.8141

Run	Chest (ms)	Trap (ms)	Enemy (ms)	Total (ms)	Avg per object (ms)
7	7.81	3.86	349.98	361.65	0.9041
8	8.80	3.39	287.67	299.86	0.7497
9	10.73	6.80	366.02	383.55	0.9589
10	11.45	7.31	383.53	402.29	1.0057
Average	**7.56**	**4.48**	**324.58**	**336.62**	**0.8416**

Table 6 shows that the average total spawning time of dynamic objects (Enemy + Chest + Trap) is 336.62 milliseconds per session. The largest contribution comes from Enemy spawning, with an average of 324.58 ms, due to its higher object count (340 objects). The average per-object spawning time is approximately 0.84 ms, far below the threshold typically perceptible to users (above 100 ms per object). This confirms that the proposed method has high computational efficiency and does not significantly affect the player's experience during the loading process.

4.6. Advantages And Limitations

Based on the implementation and testing results, the proposed method offers several advantages: (1) *playability guarantee* through layered constraint mechanisms (boundary, overlap, restricted zone, cross-type), with a 100% success rate across 10 test sessions; (2) *high computational efficiency* (0.84 ms per object) supported by the SpatialGrid optimization; (3) *significant content variation* between sessions (86.31–2358.00 studs); (4) *modular architecture* that simplifies maintenance and extension; and (5) *explicit designer control* over key parameters, in contrast to black-box methods such as machine learning or evolutionary approaches.

However, the method also has several limitations: (1) *local path validity*, because the restricted-zone validation only evaluates static distances from key points and does not detect global path obstructions; (2) *manual parameter tuning*, since values such as MIN_DIST, zone radius, and attempt limits must be configured manually; (3) *uniform probability distribution*, which does not yet account for spatial context such as room corners or hidden areas; (4) *no level reproducibility*, because seed values are not stored persistently; and (5) *scalability of relational constraints*, as the cross-type constraint matrix grows quadratically with the number of object types.

5. Conclusion

The application of the procedural generation method using the Rule-Based Random Generation approach in Roblox-based adventure games has been successfully implemented. The method is able to perform automatic dynamic object placement without manual intervention, increasing development efficiency and providing flexibility in game design. The constraint mechanism applied in the algorithm has been proven to preserve playability: the system always keeps the main path open, prevents object overlap, and maintains a logical and balanced object distribution. This demonstrates that the rule-based approach is superior to pure randomization.

The system is also capable of producing variation in object placement in every game session, increasing replayability. Integration into the Roblox Studio environment runs well, particularly for the main components such as *DungeonGenerator*, *ObjectSpawner*, and *ConstraintValidator*. The use of a server-side architecture supports the consistency and stability of the system in performing object generation. The application of *SpatialGrid* as an optimization mechanism improves system performance, especially in the inter-object distance validation process.

For further development, several directions are suggested: (1) developing more adaptive algorithms capable of dynamically adjusting difficulty based on player behavior; (2) integrating artificial intelligence (AI) approaches to enhance the complexity of dynamic object behavior, especially enemies; (3) expanding constraint variations to also cover gameplay aspects such as difficulty balance, reward distribution, and progression flow; (4) further optimizing system performance for larger-scale environments through more efficient data structures; and (5) adding more item variations to enrich the procedurally generated content.

References

[1] J. Whitehead, "Spatial layout of procedural dungeons using linear constraints and SMT solvers," in ACM International Conference Proceeding Series, 2020, doi: 10.1145/3402942.3409603.

[2] L. T. Pereira, P. V. de S. Prado, R. M. Lopes, and C. F. M. Toledo, "Procedural generation of dungeons' maps and locked-door missions through an evolutionary algorithm validated with players," Expert Systems with Applications, vol. 180, 2021, doi: 10.1016/j.eswa.2021.115009.

[3] V. Kumaran, D. Carpenter, J. Rowe, B. Mott, and J. Lester, "Procedural level generation in educational games from natural language instruction," IEEE Transactions on Games, vol. 16, no. 4, pp. 937–946, 2024, doi: 10.1109/TG.2024.3392670.

[4] E. W. Hidayat, E. N. F. Dewi, and I. S. Ramadhan, "Application of procedural content generation system in forming dungeon level in Dungeon Diver game," Jurnal Teknik Informatika (Jutif), vol. 5, no. 3, pp. 873–881, 2024, doi: 10.52436/1.jutif.2024.5.3.1465.

[5] R. Selviana and D. B. F. Lugata, "Implementasi generate map dan pemunculan objek secara acak pada game 3D menggunakan bahasa C# dan metode Perlin Noise di Unity," vol. 15, no. 1, 2023.

[6] C. Kim On, N. W. Foong, J. Teo, A. A. A. Ibrahim, and T. T. Guan, "Rule-based procedural generation of item in role-playing game," vol. 7, no. 5, 2017.

[7] J. P. Sousa, G. Oliveira, B. I. Borges, B. Boas, R. Tatto, and R. P. Lopes, "Implementation and playtesting for a world adventure game's procedural content generation system," in ArtsIT, Interactivity and Game Creation, A. L. Brooks, Ed. Springer Nature Switzerland, 2023, pp. 187–197.

[8] N. Shaker, J. Togelius, and M. J. Nelson, Procedural Content Generation in Games. Springer Cham, 2016, doi: 10.1007/978-3-319-42716-4.

[9] M. Fahrizal Afni Romadhan, K. Raditya Pratama, M. Syaiful Anam, M. Alfis Sholikhin, and T. Wicaksono Hermawan, "Penerapan model project-based learning (PjBL) dalam pengembangan modul pembelajaran pembuatan game obby berbasis Roblox Studio untuk siswa sekolah dasar," JMA, vol. 3, 2025, doi: 10.62281.

[10] E. Meilinda, R. Sabaruddin, and D. Fitriani, "Model prototype sebagai metode pengembangan perangkat lunak pada sistem informasi pengaduan umum," Jurnal Khatulistiwa Informatika, vol. 9, no. 2, pp. 86–91, 2021.

[11] I. Lupiani, Procedural Content Generation for Games. Apress, 2025, doi: 10.1007/979-8-8688-1787-8.

[12] T. Pricillia and Zulfachmi, "Perbandingan metode pengembangan perangkat lunak (Waterfall, Prototype, RAD)," Jurnal Bangkit Indonesia, vol. 10, no. 1, 2021, doi: 10.52771/bangkitindonesia.v10i1.153.

[13] E. Adams, Fundamentals of Game Design, 3rd ed. Pearson Education, 2013.

[14] R. Ierusalimschy, Programming in Lua, 4th ed. Roberto Ierusalimschy, 2016.

[15] A. A. Wulandari, A. Fahrudin, and A. Rahman, "Peran Roblox dalam pembentukan identitas generasi muda: sebuah tinjauan literatur," INTERACTION: Communication Studies Journal, vol. 2, no. 2, 2025, doi: 10.47134/interaction.v2i2.4777.