Journal of Computer Science and Technology
Volume 4 ; Number 3 ; September 2026 ; Page 134-144
DOI : https://doi.org/10.59435/jocstec.v4i3.828
https://jurnal.padangtekno.com/index.php/jocstec
E-ISSN : 2985-4318
P-ISSN : 2985-5772

Procedural Level Generation For 2D Platformer Games Based On Graph And Chunk

Akmal Zidan Fahreza[1*], Chrystia Aji Putra[2], Yisti Vita Via[3]

[1-3] Study Program of Informatics, Universitas Pembangunan Nasional "Veteran" Jawa Timur
[1*]22081010163@student.upnjatim.ac.id

ABSTRACT

Level design is one of the key components in determining the gameplay experience in 2D platformer games. However, manual level design is time-consuming and difficult to scale. Graph theory can be used to represent the abstract structure of a level, with nodes as spatial units and edges as paths connecting those spaces. The problem is that this graph structure cannot always be directly translated into a valid and playable physical layout due to platformer constraints, such as jump height, safe distance, and gravity. This research proposes a chunk-based transformation approach, in which each graph node is represented as a fixed-size level chunk containing a platform layout, obstacles, and entry and exit points. The transformation of the graph into a physical level is carried out through the selection, arrangement, and alignment of chunks based on the graph's topology. This study aims to design a framework for transforming an abstract graph G=(V,E) into a chunk-based 2D platformer level layout. The method used is qualitative-descriptive, involving the formulation of a graph model, rules for mapping and arranging chunks, and playability criteria. The results of the study reveal three main stages: mapping vertices to corresponding chunks, arranging chunks according to the graph's structure, and validating reachability using an automatic path-finding agent.

Keyword : Procedural Content Generation, Level, 2D Platformer, Graph, Chunks

INTRODUCTION

Today's rapid technological advancements are driving the continued growth of the gaming industry. The evolution of the gaming industry has given rise to a wide variety of genres, each with its own distinct characteristics and concepts, one of the most popular and widely enjoyed genres is the 2D platformer. 2D platformers have become a popular genre because, despite their simple mechanics, they deliver an exciting and engaging gaming experience [1] [2]. Level design in the 2D platformer genre plays a crucial role because it directly influences the game's pacing, however, developers face challenges when confronted with such expansive level designs [3] [4]. Creating overly expansive levels manually consumes a great deal of time and resources, resulting in a significantly longer game development cycle [5] [6].

Procedural Content Generation (PCG) is an approach that simplifies game development by automating content creation [7]. PCG is a computational algorithm technique that automatically generates game content, such as items, characters, maps, or level layouts [8] [9]. By utilizing PCG, developers can create more engaging games by featuring dynamic, adaptive, and virtually limitless content [10] [11].

However, most PCG methods in the 2D platformer genre such as GANs and PRNGs (pseudo-random-number generators) still operate at the game's tile or pixel level. This can potentially result in poorly structured levels, making it difficult to ensure that the generated levels can actually be completed by players [12]. Furthermore, developers lack control over the narrative sequence and level

difficulty, since the generation process occurs without specific structural constraints, such as game logic.

Given these limitations, graph-based level generation offers a promising alternative for 2D platformer games, namely graph grammar [13]. In this approach, a level is first represented as a graph[14]. The graph contains nodes and edges, where nodes represent game components such as rooms or challenges, while edges represent connecting paths or relationships between game rooms [15].

Because the game layout and paths have been modeled before the level layout is constructed, using a graph-based approach will facilitate the validation and verification process for the resulting level structure [15]. This reduces the likelihood of creating unsolvable levels. Although various studies have adapted graph representation methods as tools for constructing level structures, most researchers stop at the graph generation process. Meanwhile, the process of transforming graph structures into the physical form of the game is rarely described in detail, particularly in 2D platformer games.

Based on this background, this study will focus on designing and generating a game level through a graph transformation process, followed by the translation of nodes into game spaces (chunks) and edges as connectors linking those spaces.

METHODOLOGY

The Research Methodology section describes the steps taken by the author during the research process, which ranges from a literature review to performance testing of the resulting system.

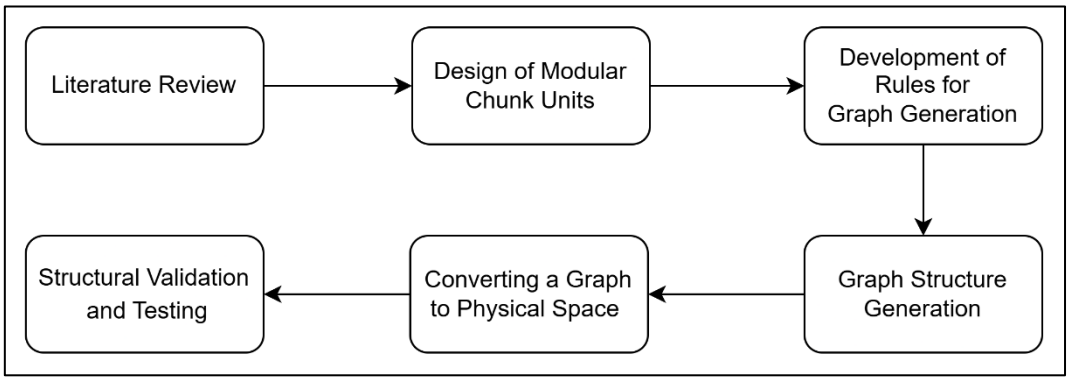


Figure 1. Research Phases

The first step in this research was to conduct a literature review on procedural content generation and graph representations in the physical space of a game, the literature review was based on various publications and journals published within the past five years. The research continued with designing and formulating a graph data structure as required. The graph structure consists of a set of nodes that correspond one-to-one with the game's physical layout specifically, chunks and a set of edges representing relationships between spaces. Once the design is complete, the system will proceed to generate a graph structure consisting of a set of nodes and edges.

The resulting graph structure will be directly used in the next process: conversion to a chunk-based physical game space. This step constitutes the core and primary focus of the research specifically, how the system transforms the graph structure into the game's physical layout while preserving its structure. In this process, the system will match node types with chunk types while simultaneously maintaining connectivity between chunks through a connector standardization mechanism. The final stage is conducted to ensure the validity of the generated level structure.

The testing phase was conducted using the Breadth-First Search (BFS) pathfinding algorithm as a validator, BFS traverses the generated graph to determine whether the resulting structure is playable by the player. In this study, BFS does not serve as a pathfinding simulation in Unity's tile-based system but focuses on level reachability to ensure that the end node (goal node/chunk goal) can be reached by the player. This is because the physical parameters of gravity and jump height have already

been validated during the manual development of the chunks. Each chunk is constructed in such a way that the player's movements (including jumps) are guaranteed to be physically feasible before they are finally included in the chunk library used by the system. In addition, a stress test was conducted by generating 100 levels using 10 iterations to evaluate the system's consistency and success in repeatedly generating levels.

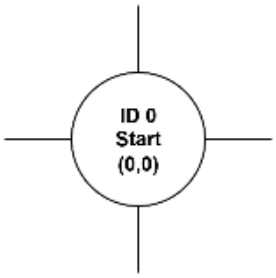
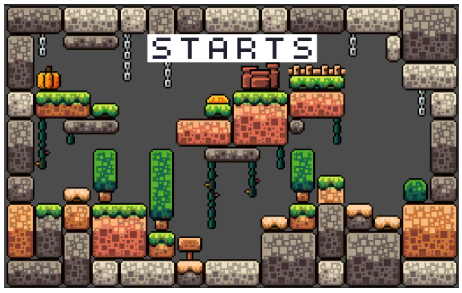
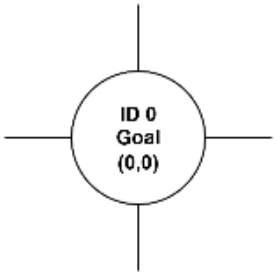
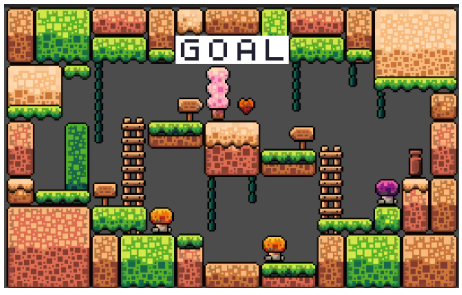
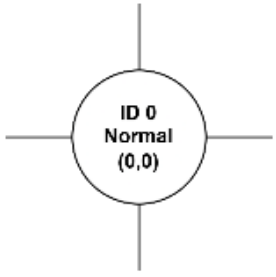
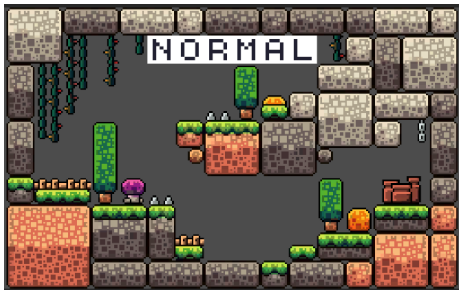
RESULTS AND DISCUSSION

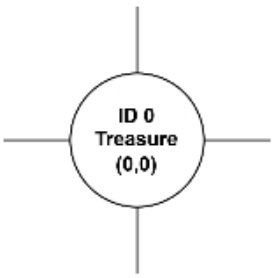
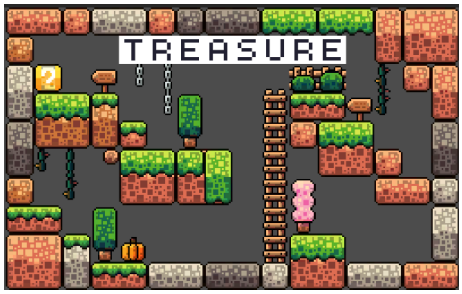
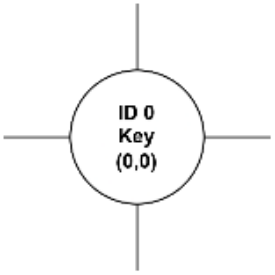
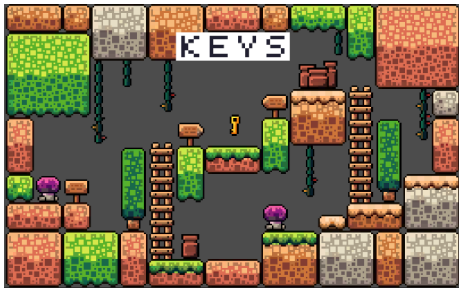
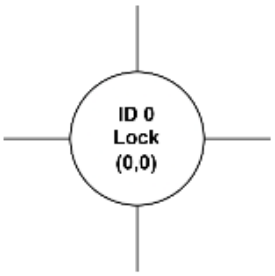
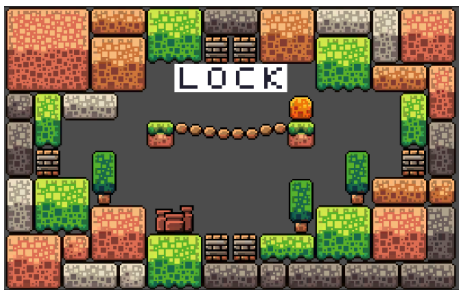
In conducting this study, the author used the same development environment and consistently used it from the beginning of the study through the final testing phase. This study was conducted on a laptop equipped with an AMD Ryzen 3 5300U processor, 8 GB of RAM, and a Radeon Graphics GPU (2.60 GHz). In addition, the author also used a game engine called Unity, version 2022.3.30f1.

Development of Modular Chunk Units

A modular unit is an in-game object designed to be reused without repetitive development processes, modular units can be used for the development of characters, items, or other aspects of the game. This study applies this concept to the development of chunk-based game levels, each chunk will be designed manually (handmade) so that technical parameters such as jump distance and gravity can be adjusted during the development process. There are at least six types of chunks to be developed: chunk_start, chunk_goal, chunk_normal, chunk_treasure, chunk_key, and chunk_lock. The following shows a mapping of each chunk, representing each node type in a graph.

Table 1. Node Mapping Within a Chunk

No	Node Type	Chunk Representation
1		
2		
3		

4		
5		
6		

Based on the information in Table 1, each node type has its own representation in the form of a chunk-based game space, with each chunk having different designs, characteristics, and functions. The data in Table 1 will serve as a reference for the system in performing the graph structure conversion process, so that each node not only acts as a component in generating the graph structure but also plays a direct role in shaping the game space.

Chunk-to-Chunk Connector System

Chunk-based level generation in 2D platformer games presents several development challenges, one of which is how to create a system that can dynamically determine the direction of chunk connections based on edges in the graph. To address this, the author standardized the connectors for each chunk developed, ensuring that every chunk has doors of the same height and width.

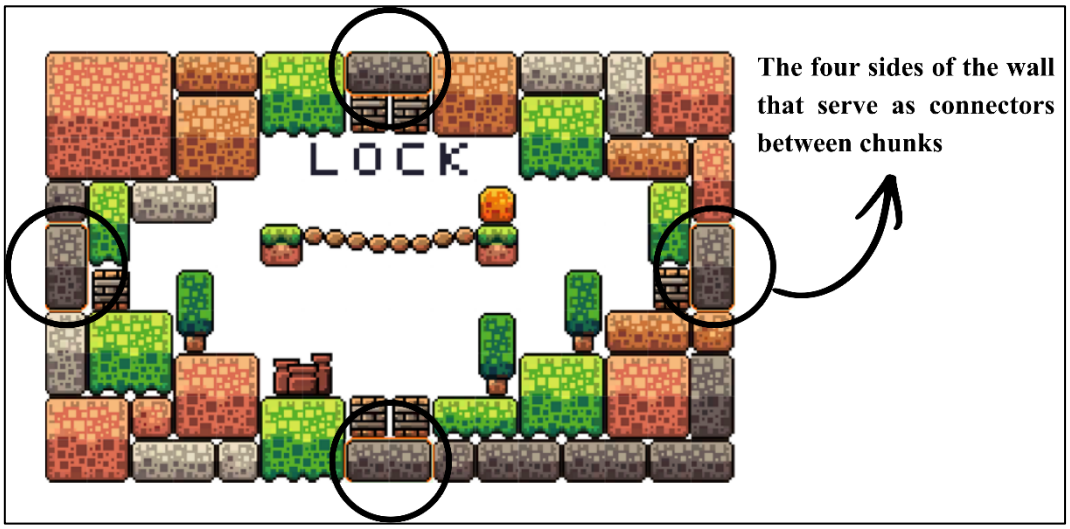


Figure 2. Connector Sides in Each *Chunk*

This method allows the program to simply identify which side corresponds to the graph structure, after which the system can open a connector in that direction. To determine the direction of a neighboring node in the current space, the program uses the concept of relative vectors based on the coordinate values of the current space. The program calculates the coordinate difference between two connected nodes to determine the direction of the neighboring node. The following is a simulation of the calculation to find the direction of a neighboring node between two connected nodes.

Given: n1 and n2 are neighbors of n1 with the following coordinate positions:

n1 (current node) = (5,0)

n2 (neighbor node) = (6,0)

Question : In which direction is n2, as a neighbor of n1?

Answer :

diff = (Coordinates node n2) - (Coordinates node n1)

diff = (6,0) – (5,0)

diff = (1,0)

diff.x = 1; diff.y = 0

To determine the position of a neighboring node relative to the current node, the system first calculates the coordinate difference between the two nodes by subtracting the current node's coordinates from those of the neighboring node. For example, suppose the current node n1 is at coordinates (5,0), while the neighboring node n2 is at coordinates (6,0). The coordinate difference is calculated using the formula diff = position of n2 − position of n1, resulting in diff = (6,0) − (5,0) = (1,0). Since x = 1, node n2 is to the right of node n1. Conversely, if x = −1, the neighboring node is to the left. The same principle applies to the y-axis, where a value of y = 1 indicates the upward direction and y = −1 indicates the downward direction. Here is a view of the level with its chunks connected.

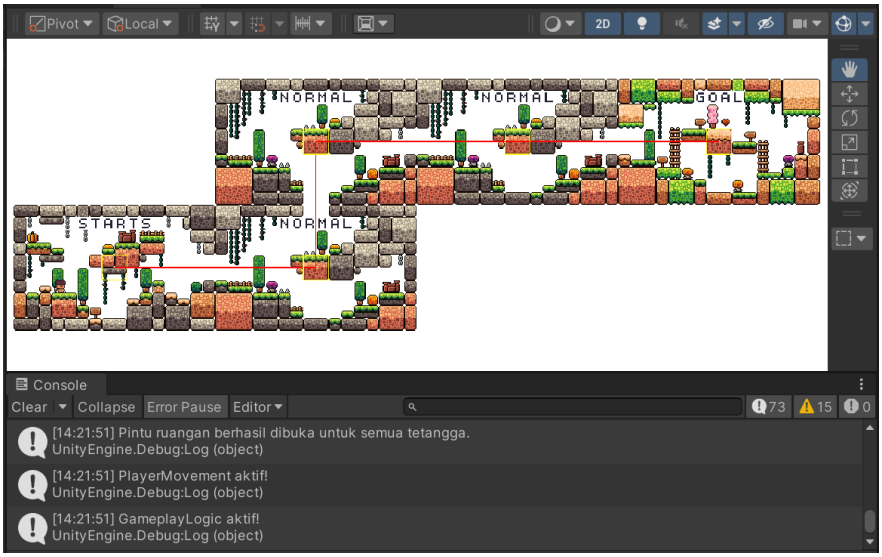


Figure 3. Game Levels That Are Connected Across Chunks

Figure 3 shows the result of the level generation, in which each chunk is successfully connected by creating door openings in the walls according to the specified direction. This result ensures that the connections between chunks are properly implemented in the generated level.

Designing Rules for Graph Grammar

In this stage, several graph grammar production rules will be designed to facilitate graph transformation. Each rule will have a Left-Hand-Side (LHS) the condition to be satisfied in a graph and a Right-Hand-Side (RHS) the process performed when the LHS condition is met. There are three rules to be developed: Level Expansion, Branching, and Key Lock.

1. Level Expansion
   Level expansion is a rule that serves to expand the level structure by adding a normal node to the graph. In this process, the system will search for a node located before the goal node and then add a new normal node between the two nodes. The following is an illustration of the LHS and RHS for the Level Expansion rule.

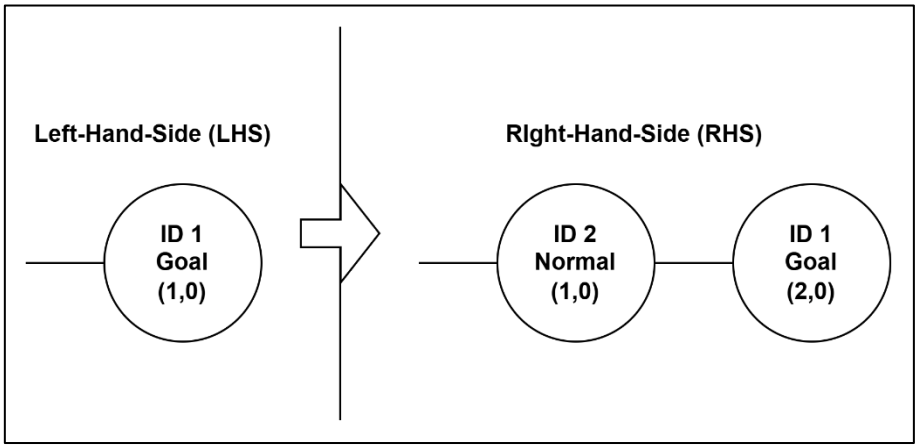


Figure 4. LHS and RHS of Level Expansion Rule

2. Branching
   This rule adds a new “treasure” node to the graph structure, this addition serves to introduce branching variations in the generated level. The system searches for a normal node with fewer than three connections, when this condition is met, a treasure node is added and placed at that node. The following is an illustration of the LHS and RHS for the Branching rule.

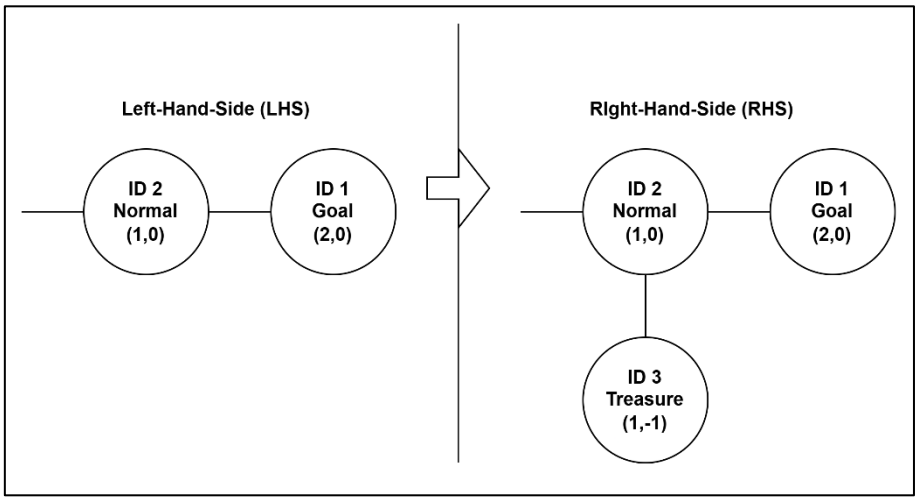


Figure 5. LHS and RHS of Branching Rule

3. Key-Lock
   These rules will create key nodes and lock nodes in the level structure to add variety to the level using simple puzzle elements. In the process, the system searches for two adjacent normal nodes (n1, n2) so that a lock node can be inserted between them, then, a key node is placed as the node connected to node n1.

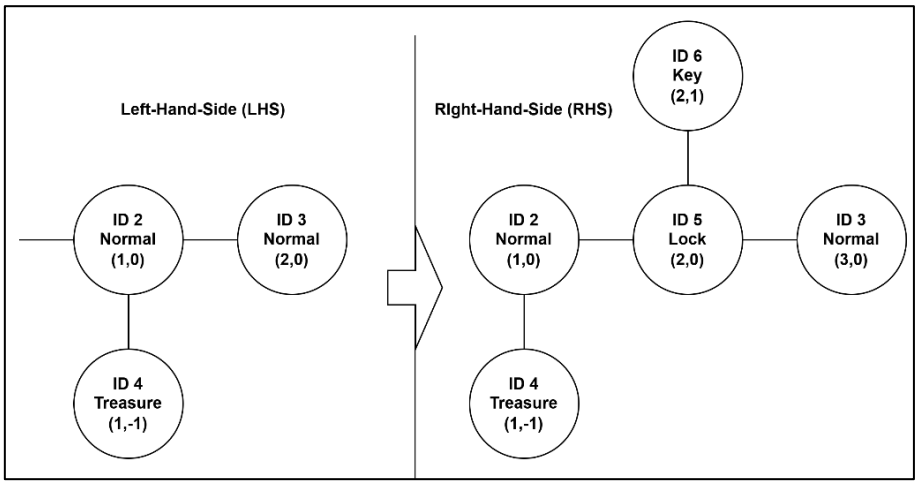


Figure 6. LHS and RHS of Key-Lock Rule

Applying this rule will introduce a unique mechanic into the level, so adjustments are needed when validating using BFS. The system will create a boolean variable named `hasKey` to handle the door-and-key logic, the `hasKey` variable will have a default value of `FALSE`. When BFS reaches a lock node before finding a key node, `hasKey` remains FALSE, and the lock node is treated as a solid wall that cannot be added to the queue or the visited list. However, when BFS successfully finds a key node, `hasKey` changes to TRUE, and the lock node is treated as a regular node that can be traversed, allowing the algorithm to continue its search for the goal node.

Graph Structure Generation

At the beginning of each level, the system creates a simple graph structure consisting of two basic nodes the start node and the goal node and connects them with a set of edges. These two nodes form the initial structure of the graph, which is then rewritten as the graph transformation process proceeds. The following is an illustration of the start and goal nodes as the initial structure of a graph.

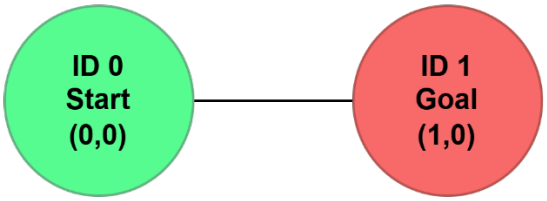


Figure 7. Initial Graph Structure Containing 2 Basic *Nodes*

As the initial structure, the start node is placed at the origin (0,0) and the goal node is placed at (1,0), directly to the right of the start node. Once the initial graph structure is formed, the system continues the generation process by expanding the graph structure according to predefined rules. This expansion process is performed iteratively until the specified number of iterations is reached. Each iteration can result in additions or changes to the graph structure, so the complexity and variation of the level structure can increase as the number of iterations increases. The following is an illustration of the graph structure generated after the generation process has been performed for 10 iterations.

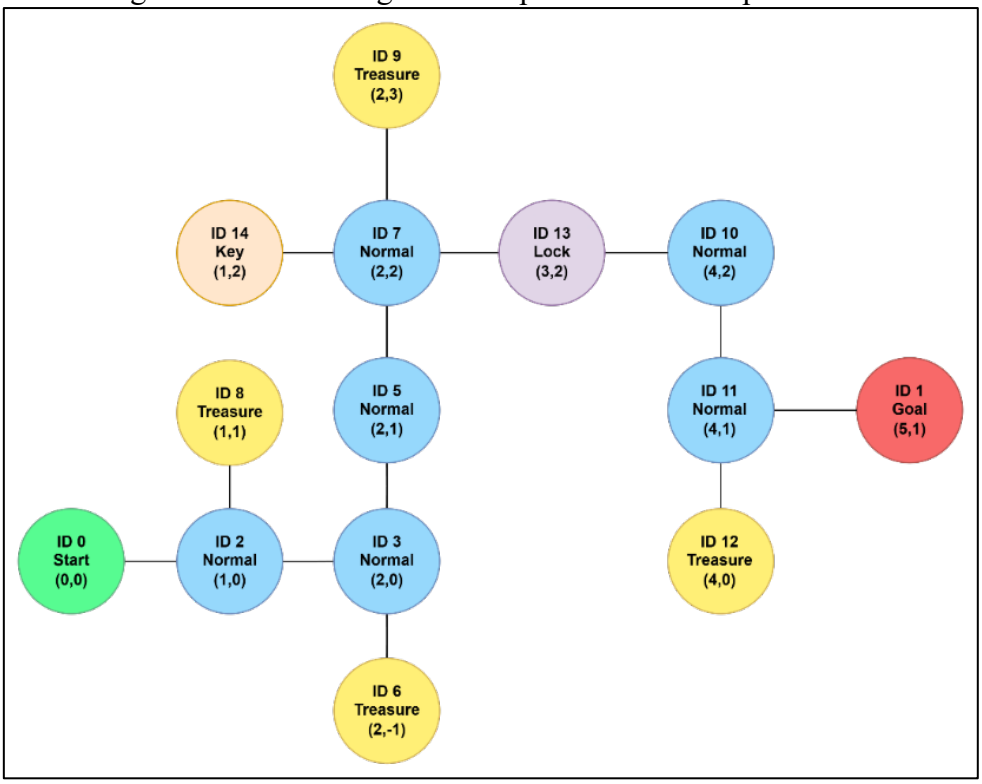


Figure 8. Graph Structure After 10 Iterations of Generation

Conversion to Game Level Layout

After the graph structure has been successfully generated through a certain number of iterations, the level generation process continues by converting the graph structure into the physical layout of a 2D platformer game level. To execute this process, the program maps each node in the graph to a chunk of the appropriate type. In addition to mapping each node, the system also interprets edges as connectors linking the chunks, thereby forming a path that the player character can traverse. Below is a game level resulting from the conversion of the graph structure shown in Figure 8.

Figure 9. Results of Converting the Graph Structure to a Game Layout

System Testing Using BFS as a Validator

After the procedural level-generation system was successfully developed, a system test was conducted using the BFS algorithm as the path-finding agent. The test was performed using a stress test method, which aimed to evaluate the system's performance by repeatedly generating a large number of levels. BFS applies level reachability criteria to determine whether a generated level is valid or invalid, a level is deemed valid if BFS is able to reach a node of the "goal" type. If this criterion is not met, BFS will mark the level as invalid. The following data shows the results of 100 level generation tests, with a total of 10 iterations at each level.

Table 2. System Test Results

Iteration	Status	Total Nodes	Generation Time (ms)
1	Success	14	50.370
2	Success	14	40.090
3	Success	14	41.843
4	Success	12	34.747
…	…	…	…
…	…	…	…
…	…	…	…
97	Success	14	39.809
98	Success	14	41.493
99	Success	13	36.308
100	Success	14	47.796
Success Rate		100%	
Fail Rate		0%	
Average Time		39.842 ms	

System testing was conducted by repeatedly generating levels 100 times, with the tests focusing on the validity of the levels and the time required by the system to generate a level. Based on the test results in Table 2, all level generation experiments were deemed valid and successful, with a success rate of 100%. The test data also shows the generation time for each iteration, with an average generation time of 39.842 ms. These results indicate that the system successfully generated levels 100 times in a row without any failures, with a relatively stable generation time.

CONCLUSION

The developed system successfully translates graph structures into physical representations of 2D platformer game levels using a chunk-based approach. During the transformation process, each node in the graph structure is translated into a chunk representing the game space, while edges are used as references to form physical connections between chunks. These connections are realized through a path-opening mechanism on the sides of interconnected chunks. With this approach, an abstract graph structure can be translated into a physical level layout that forms an interconnected, playable game space.

REFERENCE

[1] N. Aditiya and D. Laily Fithri, "ANALISIS GAME 2D 'MONKEY BANANA SURVIVAL' SERTA PENGUJIAN MENGGUNAKAN METODE BLACKBOX," *Jurnal PROSISKO*, vol. 11, no. 2, Sep. 2024.

[2] D. Revido, I. Kumara, I. N. Farida, and P. Kasih, "Pengembangan Game 2D Platformer 'Skimo' Sebagai Media Relaksasi Dengan Menggunakan Metode Finite State Machine," in *PROSIDING SEMINAR NASIONAL TEKNOLOGI DAN SAINS*, 2025.

[3] M. Didik Wahyudi and H. Z. Fahmi2, "Perancangan Game Android 2D Platformer Runner Math Untuk Meningkatkan Kemampuan Operasi Aritmatika Siswa Sekolah Dasar (Studi Kasus: SD Negeri Kletek)," *Jurnal Manajemen Informatika*, vol. 12, no. 2, Jul. 2023.

[4] H. A. Rosyid and A. A. Prasetyo, "Implementasi Algoritma Binary Space Partitioning Untuk Procedural Map Generation Dalam Gim," *Jurnal Informatika: Jurnal Pengembangan IT*, vol. 10, no. 4, pp. 882–894, Sep. 2025, doi: 10.30591/jpit.v10i4.8629.

[5] R. J. D. Sorongan, P. T. D. Rompas, and M. H. Tinambunan, "GAME STRATEGI 'MON WAR' DENGAN PROCEDURAL WORLD GENERATION MENGGUNAKAN METODE PENGEMBANGAN AGILE PROGRAMMING," *Jurnal Pendidikan Teknologi Informasi dan Komunikasi*, vol. 6, Jun. 2026.

[6] F. Febryan and L. Abdurrahman, "Analisis Algoritma Wave Function Collapse sebagai Metode Prosedural Generator untuk Pembuatan Medan Permainan Digital," *Blend Sains Jurnal Teknik*, vol. 4, no. 4, pp. 678–684, Apr. 2026, doi: 10.56211/blendsains.v4i4.1405.

[7] Muhammad Ijai, A. Arbansyah, and S. H. Suryawan, "Penerapan Algoritma Random Walk Untuk Procedural Map Pada Game 'The Last Hope' 2D," *Jurnal CoSciTech (Computer Science and Information Technology)*, vol. 4, no. 3, pp. 754–762, Jan. 2024, doi: 10.37859/coscitech.v4i3.6420.

[8] R. Rahman, P. Sasmito, and D. Rudhistiar, "PERANCANGAN GAME 2D TOP-DOWN SHOOTER 'OUTBREAK25' MENGGUNAKAN METODE FINITE STATE MACHINE (FSM) DAN PROCEDURAL GENERATION," *Jurnal Mahasiswa Teknik Informatika)*, vol. 9, no. 6, 2025.

[9] N. A. Ranggianto, A. P. Segara, D. Wijonarko, A. Andrianto, and M. Habibullah Arief, "Procedural Content Generation pada Level Gim Sokoban Menggunakan Model Hybrid GPT2 dan Algoritma Genetika," *Remik: Riset dan E-Jurnal Manajemen Informatika Komputer*, vol. 9, no. 3, 2025, doi: 10.33395/remik.v9i3.15188.

[10] R. Selviana and D. Bastha Fiastat Lugata, "Implementasi Generate Map dan Pemunculan Objek secara Acak pada Game 3D menggunakan Bahasa C# dan Metode Perlin Noise di Unity: Studi Kasus pada Game Development," *Jurnal SPIRIT*, vol. 15, no. 1, pp. 18–22, May 2023.

[11] T. Asrori and M. Imron, "Model Konseptual Value-Driven Procedural Content Generation (PCG) untuk Pengembangan Game Edukasi Karakter," *Jurnal Aplikasi Teknologi Informasi dan Manajemen (JATIM)*, vol. 6, no. 2, 2025.

[12] F. Maleki and R. Zhao, "Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration," in *Proceedings of the Twentieth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2024. [Online]. Available: www.aaai.org

[13] V. Hedi Satria, F. Tri Anggraeny, and P. Wirya Atmaja, "PEMBANGKITAN PETA PROSEDURAL MENGGUNAKAN GRAF MISI (MISSION GRAPH) DALAM GAME BERTEMA PETUALANGAN MENGGUNAKAN GMS 2," *Jurnal Informatika dan Sistem Informasi (JIFoSI)*, vol. 01, no. 2, 2020.

[14] S. Cooper and M. Bazzaz, "A Constraint-Based Graph Grammar Approach Unifying Level and Playthrough Generation," in *Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies*, Dec. 2025.

[15] J. Vijayakumar and L. Mathew, "NC-ENCE Graph Grammars: Properties and Application to Game Design," *International Journal of Foundations of Computer Science*, pp. 1–26, Dec. 2025, doi: 10.1142/S0129054125500467.