PAPER

Graph Based Wave Function Collapse Algorithm for Procedural Content Generation in Games

Hwanhee KIM[†], **Teasung HAHN**[†], **Sookyun KIM**[††], ***Nonmembers***, ***and*** **Shinjin KANG**[†††a)], ***Member***

SUMMARY This paper describes graph-based Wave Function Collapse algorithm for procedural content generation. The goal of this system is to enable a game designer to procedurally create key content elements in the game level through simple association rule input. To do this, we propose a graph-based data structure that can be easily integrated with a navigation mesh data structure in a three-dimensional world. With our system, if the user inputs the minimum association rule, it is possible to effectively perform procedural content generation in the three-dimensional world. The experimental results show that the Wave Function Collapse algorithm, which is a texture synthesis algorithm, can be extended to non-grid shape content with high controllability and scalability.

key words: *wave function collapse (WFC), procedural content generation (PCG), Voronoi diagram, navigation mesh, game content*

1. Introduction

One of challenging issues in current commercial games is the volume of content they contain. Especially in Massive Multiple Online Games (MMOGs) on PC and mobile platforms, hundreds of different players may occupy the same area in the MMOG world. The game developer must construct incredibly vast landscapes that might contain thousands of quests or missions so that players will have days and even years of content to complete. Because both the quality and quantity of a game world are appealing to many users, it is likely that MMOGs will remain a popular source of entertainment. To satisfy users' demand for content, the cost of building massive worlds has increased dramatically.

To solve this problem, game development companies have been considering various productivity improvement techniques at the development stage. As a result, recent online games have adopted Procedural Content Generation (PCG) schemes to make the levels (Levels), Non-Playable Characters (NPCs), and item parameter generation [1]–[4] as middleware. The PCG-based generated content can generate a random parameter based on a specific rule, thereby repeatedly providing the player with various play experiences. This PCG scheme is one of several similar methodologies and has been continuously studied in the field of game artificial intelligence.

However, there are many difficulties in applying PCG techniques directly in practice. The resulting output is one outcome of optimizing the abstracted objective function, giving the game designer little control over it. This means that it is difficult for the game designer to make a game in the intended design direction. Hence, the PCG algorithm has not been actively used in game development.

For PCG to be used in real game development work, a new practical approach is needed. There is a strong need for a framework that allows the game designer to intuitively describe the core rules in PCG development. In this paper, we propose a new PCG scheme through tile based rule modularization. We propose a PCG generation method that is friendly to game designers by introducing a WFC technique that is widely used in texture synthesis. We applied our technique to the creation of levels, which form the core content of online games, and confirmed that a game designer is able to produce various results while meeting the intended core design rules.

In particular, we extended the existing grid-based WFC method of our research to a non-grid type, and it is integrated with navigation information in current 3D PC game world. Compared to existing 2D tile based studies [5], [6], our research has the advantage of being universally applicable to 3D game content or traditional 2D game content. Because our system can be applied at the developer level, game designers can evaluate the game content with various candidate PCG contents. This can help reduce the manual development costs of creating levels, NPCs, and object layouts of games for game designers.

2. Previous Work

PCG is the algorithmic creation of game content with limited or indirect user input. Content consists of the assets of a game, e.g., levels, items, characters, and quests [7]. Markov-based models and tile-based pattern recognition methods were the most popular creation techniques in early PCG research. Snodgrass et al. described a method of procedurally generating maps using hierarchical Markov chains. Their method learns statistical patterns from human-authored maps with Super Mario Bros. (SMB), based on tile-by-tile and abstract tile transitions. They presented a collection of strategies both for training the Markov chains and for generating maps from such Markov chains [8]. Snodgrass et al. also proposed Markov models to

Manuscript received November 7, 2019.
Manuscript revised March 24, 2020.
Manuscript publicized May 20, 2020.

[†]The authors are with NCSOFT, 12 Daewangpangyo 644, Bundang, Seongnam, Gyeonggi, Korea.
[††]The author is with Jeju Naional University, 102 Jejudaehakro, Jeju-si, Jeju Special Self-Governing Province, Korea.
[†††]The author is with Hongik University, 2639 Sejong-ro, Jochiwon, Sejong, Korea.
a) E-mail: shinjin.kang@gmail.com

DOI: 10.1587/transinf.2019EDP7295

Mario Game Level Difficulty - Solution

generate content for multiple games. They applied the Markov models to three game genres to determine how well their models perform in terms of the playability of the generated content [9]. Dahlskog used evolutionary computation approaches that search for instances of game design patterns at different abstraction levels that make up SMB. Levels and a learning algorithm were implemented based on n-grams of patterns from the original SMB-game [10]. Summerville et al. presented a method for guiding Markov chain generation using Markov Chain Monte Carlo Tree Search (MCMCTS). These approaches require substantial human authoring, such as authored patterns of level content or the categorization of stylistically similar level elements [11].

Recent advances in deep learning technology have had a great impact on the PCG field. Reed et al. proposed a model using an auto-encoder, a rotated game sprite, and 3D car model images, and applied animation to other images [12]. Summerville et al. also suggested a machine-learning technique to train generators on SMB videos, generating levels based on latent play styles learned from a YouTube video. They demonstrated the process for extracting the path from the video and how the information feeds into a long short-term memory network or recurrent neural network [13].

Jain et al. learned the pattern of a game map using an auto-encoder, and automatically generated a new game map [14]. Xue et al. proposed a new convolutional neural network model, and combined new behaviors with objects in images [15]. Horsley et al. suggested that convolutional neural networks can be used for game sprite generation, even with few input datasets. They utilized a Deep Convolutional Generative Adversarial Network (DCGAN) for learning about and generating sprites [16]. Kim proposed an automatic character portrait generation system using a variational auto-encoder [17]. Beckham et al. proposed terrain generation system by leveraging extremely high-resolution terrain and height map data provided by a NASA project. They used Generative Adversarial Networks (GANs) to create a two-stage pipeline in which height map can be randomly generated as well as a texture map that is inferred from the height map [18]. Giacomello et al. applied GANs to learn a model of DOOM game levels from human-designed content. They trained two GANs: one using plain level images, one using both the images and some of the features. Their results showed that GANs could generate the structure of DOOM levels in first-person shooter games [19]. PCG techniques based on machine learning have a disadvantage in that it is difficult for them to generate content that is intended by the game designer because of the black box characteristic inherent to artificial neural networks. In recent years, research has focused on industry to introduce a PCG technique that satisfies specific rules and patterns intended to create content intended by game designer. Hoover et al. generated video game levels through a representation called functional scaffolding for musical composition, which was originally designed to procedurally compose music. They presented a method for deconstructing game levels into multiple "instruments" or "voices", wherein each voice represents a tile type. Their proof-of-concept experiments showcase that music is a rich metaphor for representing naturalistic but unconventional and playable levels in the classic platform game SMB [20]. Khalifa et al. introduce the general video game rule generation problem and the eponymous software framework, which will be used in a new track of the General Video Game AI competition. The problem is as follows: given a game level as input, generate the rules of a game that fits that level. They describe the application programming interface and three different rule generators: random, constructive, and search based. Early results indicate that the constructive generator generates playable and somewhat interesting game rules [21]. Sharif et al. suggested rhythmic analysis of the General Video Game Level Generation (GVG-LG) framework and have discerned 23 common design patterns. In addition, they have segregated the identified patterns into four unique classes. The categorization is based on the usage of identified patterns in game levels [22].

In the computer graphics community, texture synthesis is the problem of generating a large output image with texture resembling that of a smaller input image. In many texture synthesis approaches, the input and output images are characterized in terms of the local patterns they contain, where these patterns are typically sub-images of just a few pixels in width. Independent game developer and generative artist Maxim Gumin proposed the wave function collapse (WFC) [23] algorithm in 2016, which is one of the texture synthesis techniques. It builds on the approach of model synthesis by dividing the model into smaller pieces and propagating the constraints to construct a coherent model. It is able to produce larger texture from a smaller input while respecting local similarity and consistency. There have been many attempts to apply WFC to game content creation.

Karth et al. [5] introduced the fact that WFC is a universally usable constraint solver, and presented a re-implementation of 2D Bitmap WFC with answer set programming. They examined WFC as an instance of a constraint solving method. They traced WFC's explosive influence on the technical artist community and explained its operation in terms of ideas from the constraint solving literature. In addition, they defined the adjacency relation in the existing 2D bitmap WFC as a "positive" adjacency relation and introduced a "negative" adjacency relation for prohibited patterns. They demonstrate how an artist might craft a focused set of additional positive and negative examples by critique of the generator's previous outputs [6]. Scurti et al. [24] describe a tool based on the WFC algorithm that performs example-based path generation on fixed maps. Their design aims at a practical system usable by non-programmers, and includes both easy input control and multiple post-processing steps. The design is implemented in Unity and enables users to visualize the results of experimenting with different path descriptions and game levels.

These WFC studies have had a direct impact on the game industry and have been used in a variety of indie

NLP+DL can solve this problem. I guess ;)

games. Various commercial games have generated a game level demonstration using the WFC algorithm [25], [26]. These results are mainly limited to level creation and 2D and 3D worlds with grid-type shapes.

The existing WFC studies have the advantage that the user can intuitively reflect the intention of the game designer in the PCG part by designing the tile and defining the linkage rule. This methodology is very useful for creating 2D or 3D games in grid form. However, recent 3D games are based on a navigation mesh based terrain data structure based on convex hulls. Therefore, to introduce the traditional grid-based WFC algorithm into 3D world-level content generation based on a 3D navigation mesh, it is necessary to additionally support a graph-based type combining rule. In this paper, we propose a graph-based WFC algorithm to extend the WFC algorithm, which was limited to 2D grid-based game content creation, to the 3D world. Our research aims to partially automate 3D game level generation, which was previously done by hand, by using the WFC algorithm.

3. Wave Function Collapse Algorithm

According to Karth et al. work [5], the WFC algorithm is defined as follows:

"The WFC algorithm is an example-driven image generation algorithm that emerged from the craft practice of PCG. In WFC, new images are generated in the style of given examples by ensuring every local window of the output occurs somewhere in the input. Operationally, WFC implements a non-backtracking greedy search method."

WFC is also related to the general search methods in CSP because it solves problems by spreading constraints. AC3 is a typical method [34]. The main difference between the two is that CSP can quickly find the solution and effectively solve even the most difficult problems [36]; however, it is not easy to control the quality of the solution. WFC can adjust the quality of the solution to the desired level through statistical sampling. By adjusting the probability that each tile will be selected, a user can also adjust the probability that multiple tiles will appear as a combined pattern. Figure 1 shows the results of generating two WFC algorithms that exhibit a significant difference when the probability of each tile is changed.

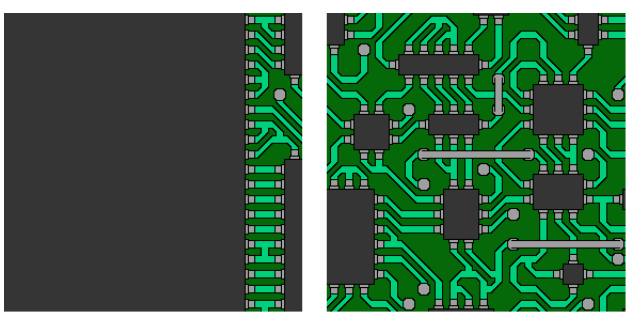

Fig. 1 Two generated results with different probability set in the same WFC algorithm (left: probability with few tiles with biased distribution, right: probability with many tiles with uniform distribution).

The idea is to divide the input texture into N×N unique patterns, where N is the number of pixels per dimension. Each unique pattern is assigned a unique label. The user provides the output texture size and the algorithm starts with all output pixels in their unobserved states—that is, each pixel value is a superposition of all colors from the texture and has the potentiality of collapsing into any of the assigned unique patterns. Then, the algorithm enters the following observation–propagation cycle:

1. Observation step: an unobserved N×N pattern with the lowest entropy is collapsed into a fixed state.
2. Propagation step: the label constraints that the observation step has imposed on the neighboring unobserved patterns are propagated throughout the entire output.

The algorithm is completed when the output is in a completely observed state (all the pixels in the output have exactly one label assigned to each of them).

By local similarity, the user implies that

1. Each N × N pattern that appears in the output should occur at least once in the input.
2. Over a sufficiently large number of outputs, the distribution of the N × N patterns in the output should approximate the distribution of the patterns in the input texture. That is, the probability density of each N × N pattern in the output should approximate the probability of each pattern in the input.

Similar to discrete model synthesis, the algorithm may end up in a contradictory state if a pixel cannot be assigned a label after the constraint propagation. Likewise, determining if a certain texture allows a non-failing solution is NP-hard. It is thus impossible to have a solution that never reaches contradiction, unless P = NP. Two different WFC models are proposed.

3.1 Overlapping Model

The overlapping model is a model in which the user gives a small example bitmap as input and divides it in N × N overlapping patterns. It then constructs an adjacency matrix by trying to overlap each possible pair of unique patterns and deciding which pairs can form compatible neighbors. Because the adjacency rules are inferred from the structure of the texture, the user does not need to provide any input aside from the example texture. The creation of the new texture is thus completely automated at the cost of a loss of

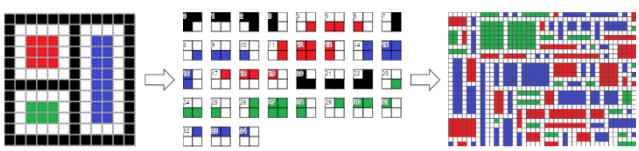

Fig. 2 WFC overlapping model example.

What about let designer to design the level based on simple painting????
Fantastic idea!

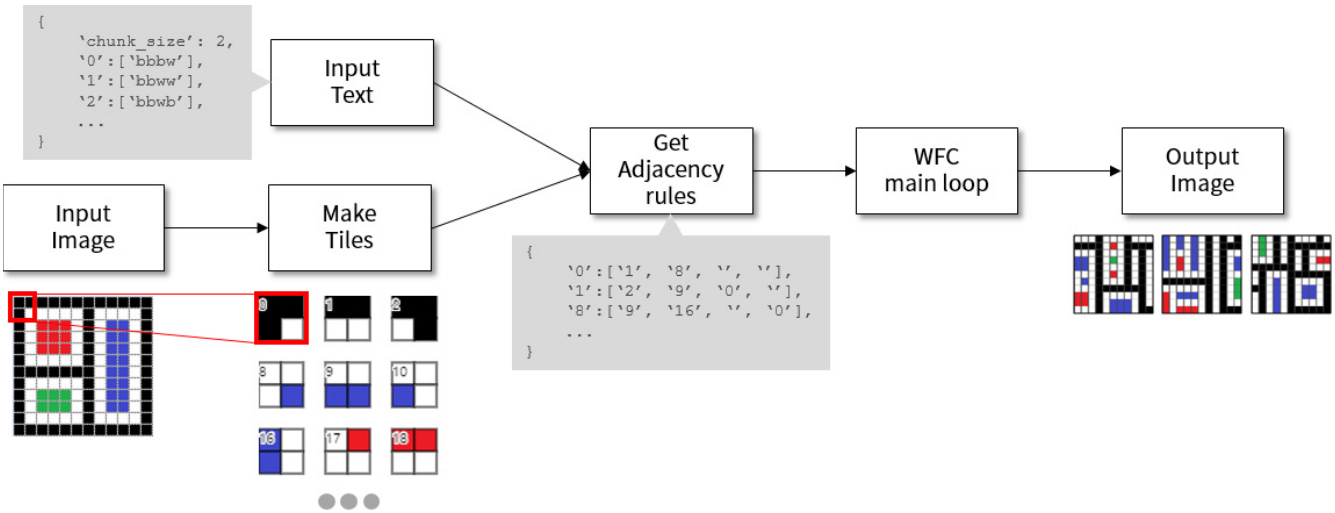

Fig. 4 Overview of the WFC algorithm.

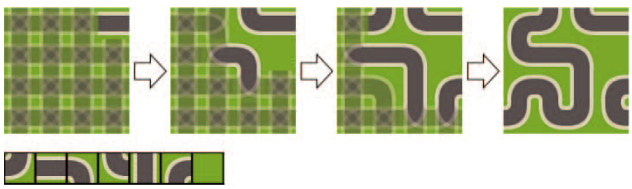

Fig. 3 WFC simple tiled model example.

precise control over the pattern placement by the user. Figure 2 shows the image generation process of the overlapping model.

3.2 Simple Tiled Model

The simple tiled model takes in an array of tiles of any size and a set of adjacency rules that define the set of allowed neighbors in the four possible orthogonal directions on a 2D plane. The only constraint is that all tiles in the array must be of the same size. This approach enables the user to create large and consistent tile map textures from smaller tiles and can be especially useful for video game development, such as the popular Pokemon game, which uses tile maps for its game development. In this study, we implemented a user interface incorporating a special symmetry system to significantly reduce the number of manual rule assignments needed for each tile. Figure 3 shows the image generation process of a simple tiled model.

4. Graph-Based WFC Algorithm

In this paper, we aim to extend the grid-based WFC algorithm to a graph-based type for application to un-structured data such as a navigation mesh. Figure 4 shows an overview of the proposed WFC process and Fig. 5 show the main loop of the WFC algorithm. The state space of the previous WFC is a regular grid, and each node constituting the grid has an equal number of neighbors. In contrast, the state space of a

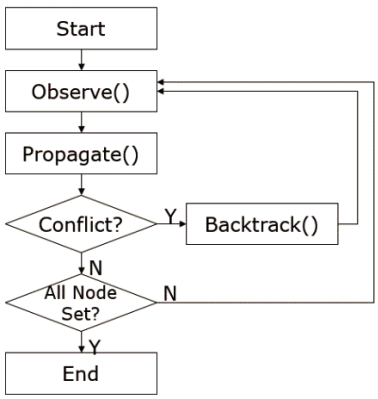

Fig. 5 Main loop of the WFC algorithm.

graph-based WFC has an unlimited and variable number of neighbors. Hence, we modified the propagation and compatibility in the original WFC algorithm.

4.1 Handling Input Data

Yeah we call it "tile"

Both WFC and graph-based WFC receive input data and produce output. The input can be in the form of an image or text. It is assumed that the chunks attached to each other in the image are connected. The non-redundant set of chunks is called the tile, and it becomes the default unit of placement for the WFC. When the input is in text form, we need to define tile information in the same format as a JSON file. Graph-based WFC uses only text-type input. This is because a typical graph structure has a link that cannot specify a direction, unlike an image.

4.2 Graph and Grid Structures

A grid is a graph in which the number of neighbors of all nodes are the same. In other words, a graph is superset of a grid. In the regular rectangular grid, Sudoku game grid,

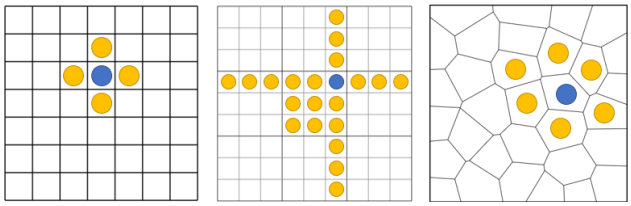

Fig. 6 Neighbors for arbitrary cells from a regular rectangular grid, Sudoku game grid, and Voronoi non-grid. The neighbors of the Sudoku game grid are determined by the game rules, and the neighbors of the regular rectangular grid and the Voronoi non-grid are determined by the adjacency of the edges. Yellow = neighbor cell.

and Voronoi non-grid[†] shown in Fig. 6, the neighbors for an arbitrary cell are visualized. The number of neighbors of a regular rectangular grid is constant at four. The number of the Sudoku game grid's neighbors is fixed to 20, which includes 3×3 small grids within each cell, horizontal line, and vertical line. In contrast, the number of Voronoi non-grid neighbors is unlimited and variable. All of the above structures are applicable to graph-based WFC.

4.3 Adjacency Node Sampling from a 3D World

Because the nodes of the navigation mesh focus on the edges of the region and their purpose is to enable the polygons that make up the mesh to be efficiently drawn, it is necessary to automatically place separate content placement nodes at the appropriate points above the navigation mesh that must be specified. These nodes must be extracted automatically from data structures representing existing worlds. The extracted data structure must be able to effectively represent the structure of the world so that it can utilize the correct adjacency information.

Poisson disc sampling [27] is an algorithm that can obtain a set of relatively uniformly distributed points in the state space with $O(n)$ time complexity. We can use this algorithm to obtain an arbitrary set of nodes on the navigation mesh.

It is necessary to determine the edge to be placed according to the connection relation of each node. After extracting edges less than the threshold value that can be connected to each other by pathfinding on the navigation mesh, overlapping edges, when viewed from the positive direction of the Z axis, are deleted in order of length. Then, a graph-based WFC algorithm can be executed on the obtained graph structure to obtain an arbitrary arrangement. Experiment 5.3 explains this further.

To apply WFC to 3D world data other than navigation-based terrain (e.g., atmospheric data), we need to discretize the world into an irregular shape and then extract the node values from it. To do this, we use a Voronoi diagram to discretize the 3D world and then use the methodology of constructing the 3D vector field by assigning vector values to the representative nodes in the discretized convex hull. This vector field can extract the characteristics of the world with lower computational cost than the existing grid type data structure, and it can easily apply the constraint or stationary value according to the WFC application intention. The vector consists of size and direction. A total of three floating point numbers (one size and two directions) are required to represent the size of the vector in 3D space. These numbers are combined to produce the desired number of vectors. The adjacency relation of the vectors is such that the size can be adjacent when the difference in absolute value is less than a certain value, and when the unit vectors of the two vectors are internally inclined, (same direction: 1, opposite directions: -1). Figure 7 shows the vector graph used for this. Experiment 5.4 explains this further.

[†]A non-grid is the opposite of a grid. The number of neighbors of all nodes does not need to be equal.

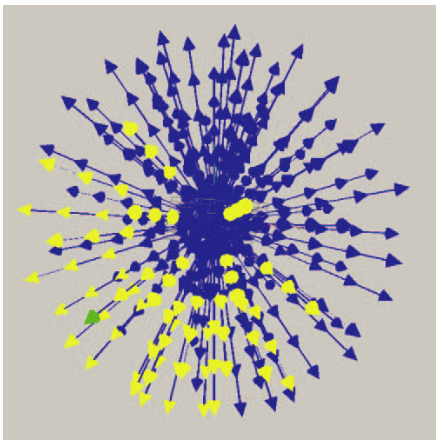

Fig. 7 The green arrow has yellow arrows for neighbors in the WFC with a maximum 60 degree radius and an absolute value difference of less than 2.

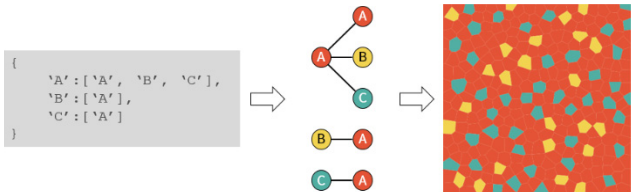

Fig. 8 Adjacency rules of the graph-based WFC algorithm according to arbitrary connection relationships and the corresponding output results.

4.4 Adjacency Rule Assignment

Figure 8 shows the graph-based adjacency rules used here and the corresponding WFC application results. The target map uses an unstructured lattice based on a Voronoi diagram instead of existing grid-type tiles. As described above, the algorithm is executed with the neighbor relation rule referring only to the connection relationships instead of the direction. As shown in Fig. 8, node A can be connected to nodes A, B, and C; nodes B and C can only be connected to node A. The output of the graph-based WFC can be obtained such that nodes B and C are arranged in an area surrounded by node A.

4.5 The Propagator and Compatible Variables

To implement the graph-based WFC, we modified the Propagator and Compatible variables of the original WFC. "Propagator" is a variable that summarizes the adjacency of two tiles. It stores the tile information that can be linked by the direction to each tile. Figure 9 shows the result of visualizing the Propagator. At the top, seven tiles that make up the Road Tileset are displayed. Beneath it are the tiles to the right, top, left, and bottom of each of the seven tiles, respectively.

"Compatible" refers the number of tiles that can be connected to a tile per direction in a particular grid location. With Propagator, one can calculate the number of tiles that can be attached to each tile in each direction. Figure 10 visualizes the Compatible variable.

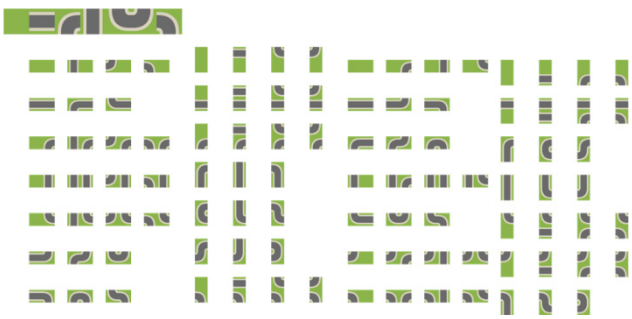

Fig. 9 Visualization of propagator variable in original WFC.

Fig. 10 Visualization of compatible variable in original WFC.

Figure 11 explains how Compatible works. First, when one tile is selected in the observe() function, the unselected tile is forbidden. It then propagates the forbidden tile's information to adjacent tiles. Only the left tile is described here. Based on the information that the green tile (without roads) is forbidden on the left tile, reduce the Compatible assigned to the four tiles by 1. If this Compatible becomes 0, it means that the tile cannot be placed on this node.

This two variables are a 3D array in the original WFC. Let T be number of all tiles, D be number of possible directions, and V be number of connectable tiles (this is a variable number). Then, the size of Propagator P is the product of the three terms: $P = TDV$. In contrast, in graph-based WFC, we cannot take into account the direction, so P can be described as $P = TV$. The Compatible C variable refers to the number of tiles that can be tied to a certain tile at a specific grid position, expressed as a 3D array of the number of all lattices N, T, and D in the existing WFC. Hence, the size of Compatible is the product of the three terms: $C = NTD$. In graph-based WFC, D is the number of neighbors that can be connected to one node, so it is limitless. So we used dictionary type than array for implementation. Therefore, we can calculate the size of Compatible as $C = NTD$.

5. Experiment

To compare the performance with the existing method, the results of Constraint Logic Programming (CLP) are presented as graph-based WFC. We used javascript on a graph-based WFC and CLP was programmed using ECLiPSe [35], an open source program. In Table 1, which summarizes the performance results, the execution time (seconds) and the number of times backtracking occurred are recorded. WFC searches nodes randomly, however the CLP has a better efficiency of searching in a predetermined order compared with random methods. The execution result of WFC presented an average of 100 runs and a 95% confidence interval, and the execution result of CLP presented

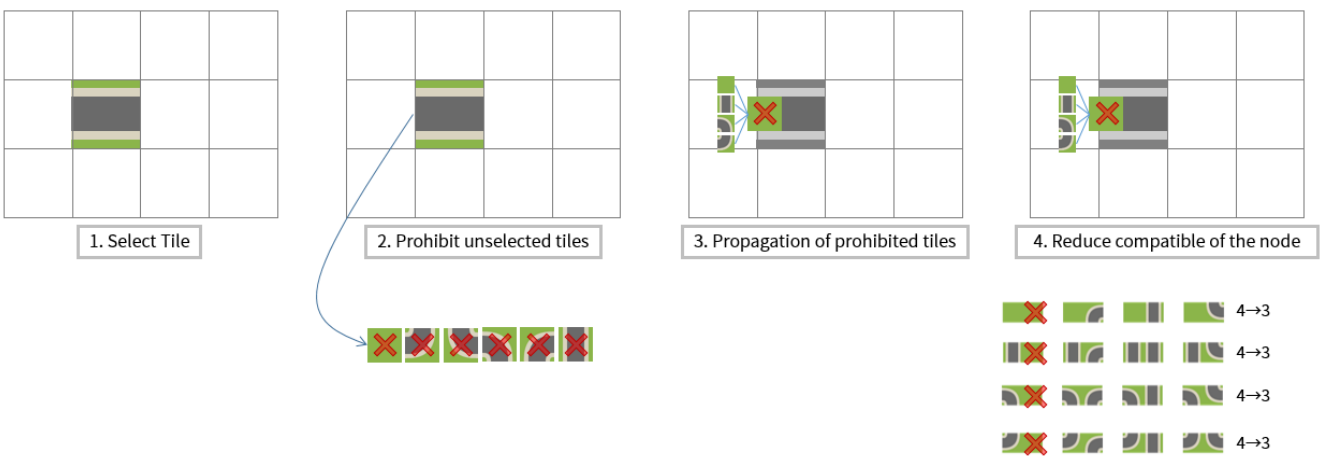

Fig. 11 Process of compatible operation in original WFC.

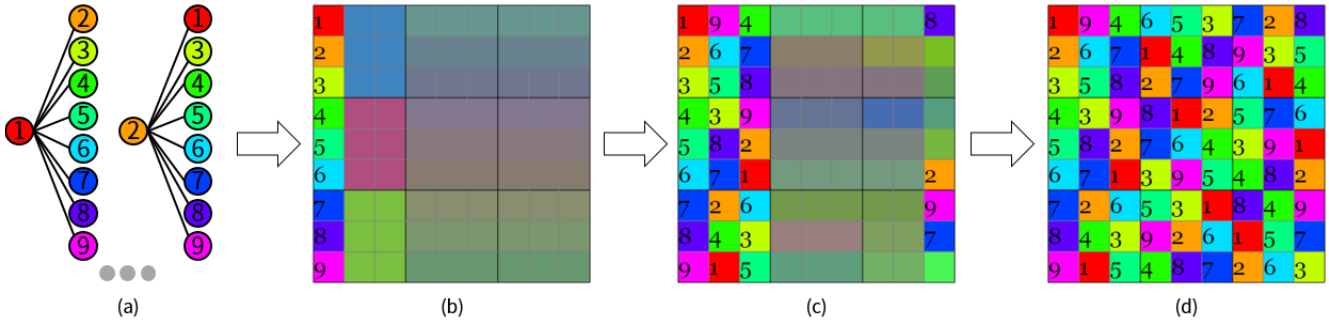

Fig. 12 Sudoku experiment. (a) Adjacency Relation. (b) Initial state (with manual constraints). (c) On progress. (d) Output.

Color Painting Question?

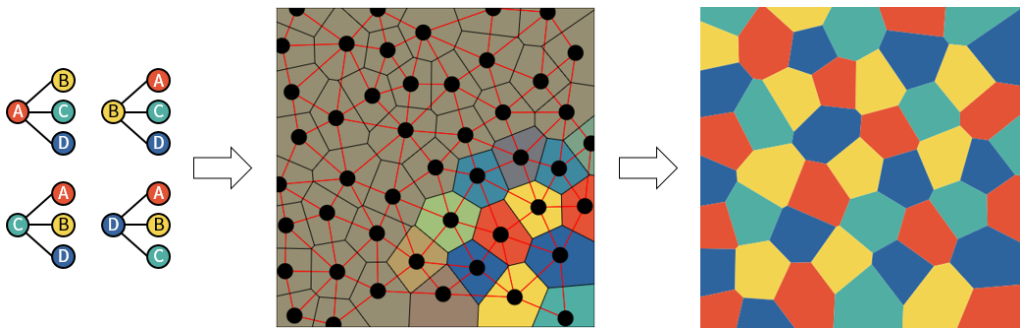

Fig. 13 4-coloring of planar graphs solving experiment. Node count = 47.

Table 1 Performance result of Sudoku experiment.

	N = 9	N = 16	N = 25
Graph-based WFC			
Time (sec)	0.21±0.01	1.43±0.13	186.05±40.91
Backtracking	0.62±0.18	6.09±1.19	182.46±35.30
CLP			
Time (sec)	0.00	0.09	0.75
Backtracking (#)	0	0	0

Table 2 Performance result of the four-color theorem experiment.

	N = 47	N = 307	N = 1,400
Graph-based WFC			
Time (sec)	0.13±0.00	0.82±0.04	10.90±1.85
Backtracking	0.04±0.05	2.49±0.45	47.75±8.64
CLP			
Time (sec)	0.00	0.02	0.11
Backtracking (#)	0	0	14

the result of one run.

5.1 Sudoku Experiment

Sudoku is a number puzzle that must be filled with numbers from 1 to 9 in a 9×9 blank space. The criteria for filling in the numbers are that only one of the numbers 1 through 9 may be filled in for the vertical, horizontal, and small 3×3 grids without overlapping. There are many ways to solve Sudoku, and the constraint satisfaction problem (CSP) solving is one popular method [28], [29]. Figure 6 shows that each Sudoku cell has 20 neighbors, so this is a grid. However, we do not have to specify direction for these 1 to 9 tiles, so a graph-based WFC can be used to solve this problem [30]. Figure 12 shows how a Sudoku puzzle is solved on graph-based WFC. The leftmost column is a predefined value constraint, and graph-based WFC can solve the problem even with constraints. The unresolved values in the middle cells show the average color value of their possible outputs. Table 1 shows the number of backtracking count and elapsed time according to the number of nodes in this experiment.

5.2 4-Coloring of Planar Graphs

To verify the implemented graph-based WFC, we checked whether it could solve the 4-coloring of planar graphs, which is used as a basic example in the CSP. The 4-coloring of planar graphs involves dividing the plane into finite parts, then painting the parts that are in contact with each other with a different color. Four colors are sufficient to do this. In this problem, when a plane is divided into finite parts, each part has a non-grid structure in which the number of neighbors in each case is not constant. Therefore, this problem is suitable for verifying the basic performance of graph-based WFC in this paper. We want to ensure that we can set up four adjacency conditions and create a batch satisfying them. Figure 13 shows the results of the creation. In this experiment, we show that the system proposed in this paper does not need any additional constraints to solve the 4-coloring of planar graphs and can solve the problem definition with the minimum definition, that is, with only the connections of each color tile. Table 2 shows the number of backtracking count and elapsed time according to the num-

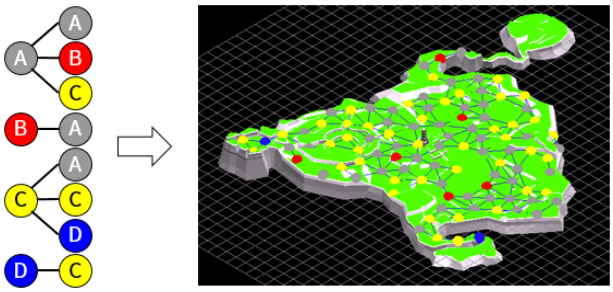

Fig. 14 PCG node placement on a 3D navigation mesh. Node count = 102.

ber of nodes in this experiment. The 4-coloring of planar graphs has relatively many adjacency rules, so we can see that the number of backtracking is large.

5.3 PCG Experiment with a Navigation Mesh

Experiments were conducted to verify whether the graph-based WFC works properly on a 3D prototype game level. The navigation mesh was created using Blender. We applied an open-source pathfinding algorithm [31] to the level for extracting reachable edges between nodes. Figure 14 shows the results generated by arranging the nodes with the graph-based WFC algorithm. Four nodes can function as an abstract symbol of the content of the game. Gray tile A represents empty space, red tile B represents weak monster, yellow tile C represents strong monster, and blue tile D represents reward. Players starting in the empty space must defeat strong monsters to reach the reward.

Variations of the level can be generated according to the intentions of the game design. As shown in Fig. 1, various results can be obtained by adjusting the probability of each tile appearing. Figure 15 shows the result of adjusting the probability of each tile appearing by adjusting the Stationary variable. The Stationary variable has a direct effect on the probability that each tile will be selected. For example, if the Stationary variable is $[2, 1, 1, 1]$, gray tiles are selected twice as often as red, yellow, and blue tiles under the same conditions. However, the probability of selecting a tile does not mean that the tile occupies the actual result. This is because the selected tile may be excluded if it is not suitable for the WFC connection.

Table 3 shows the number of backtracking count and elapsed time according to the number of nodes in this experiment. Since the experiment was performed while the stationary variable was fixed to $[20, 5, 1, 10]$, the CLP calculation result does not exist.

5.4 PCG Experiment with a 3D Vector Field

When placing the vector field in 3D space, as with the content placement above the navigation mesh, it is necessary to specify a node. We use Voronoi cells as nodes here. To obtain Voronoi cells in 3D space, we used the Voro library [32], which ports the Voro++ library written in C++ [33] to Javascript.

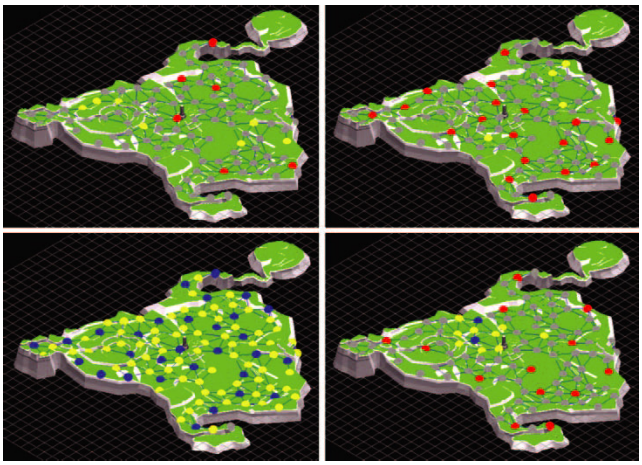

Fig. 15 Various results of changing the stationary variable. (Top left) If we increase the probability of the appearance of an empty space (gray tile) by 10 times, other nodes are rarely placed. (Top right) If we increase the probability of the appearance of weak monsters (red tile) by 10 times, the number of weak monsters increases, however the reward is almost invisible. (Bottom left) If we increase the probability of the appearance of a reward (blue tile) by 10 times, only the reward and strong monsters exist, and the empty space and weak monsters disappear. (Bottom right) Adjusting probability by adjusting the stationary variable to $[20, 5, 1, 10]$, a balanced map that fits the planning intention was obtained.

Table 3 Performance result of the navigation mesh experiment.

	N = 106	N = 245	N = 545	N = 1,062
Time (sec)	0.32±0.01	0.95±0.02	2.67±0.04	7.96±0.08
Backtracking (#)	0.00±0.00	0.00±0.00	0.00±0.00	0.00±0.00

The number of a Voronoi cell near the surface of the main play space needs to be high, and the number of a Voronoi cell of the empty sky space, which the player rarely visits, can be low. Within a single Voronoi cell, a single vector acts. Particles or characters in a Voronoi cell receive the same vector. A Voronoi cell is closer to the size of the world than the size of the character, so it can be used to implement weather changes in a global unit rather than implement the elaborate movements of the character.

The vectors thus obtained are gathered to become a vector field. We ran a particle simulation to determine if the generated vector works properly. The vector of the closest vector field at the location of the particle is added to the acceleration of the particle to be moved. Particles initially placed at random locations converge with a constant flow as the simulation proceeds. Backtracking rarely occurs because the number of tiles that can be adjacent to one tile in a connection relationship is very large. Figure 16 shows the results of Voronoi generation and the vector fields generated from them. Figure 17 shows the particle simulation results over the generated vector field. Table 4 presents the number of backtracking count and elapsed time according to the number of nodes considered in this experiment. Because the probability of tiles appearing was adjusted using stationary variables, the result of CLP calculation does not exist.

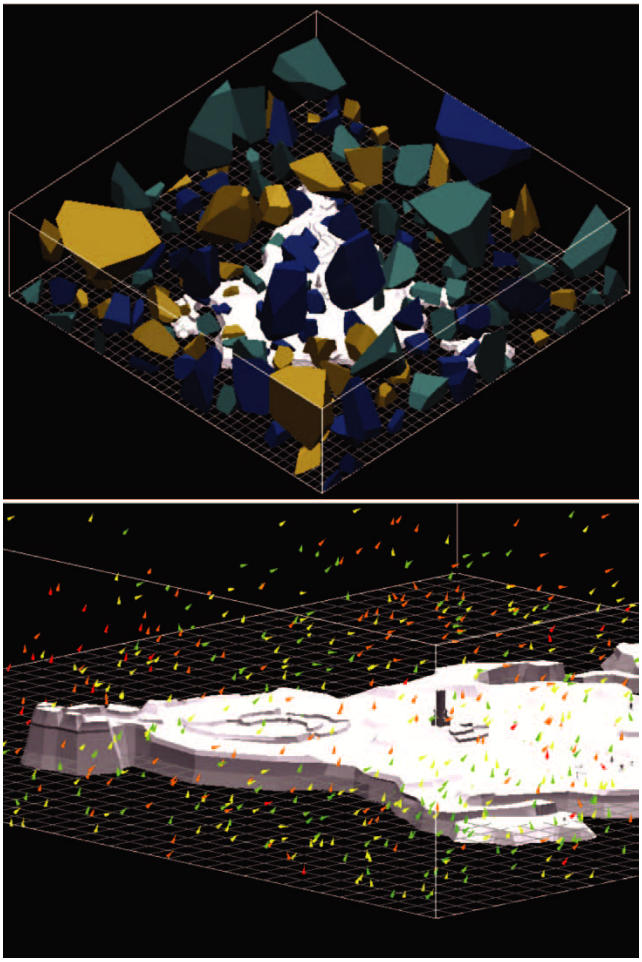

Fig. 16 (Top) Voronoi cell placed in a 3D space. The Voronoi cell of the empty sky space is large because of the low precision of the sky. Closer to the surface, higher precision is required and the cell size is smaller. Only 50% of the cells are displayed for visualization. (Bottom) The wind directions of each cell.

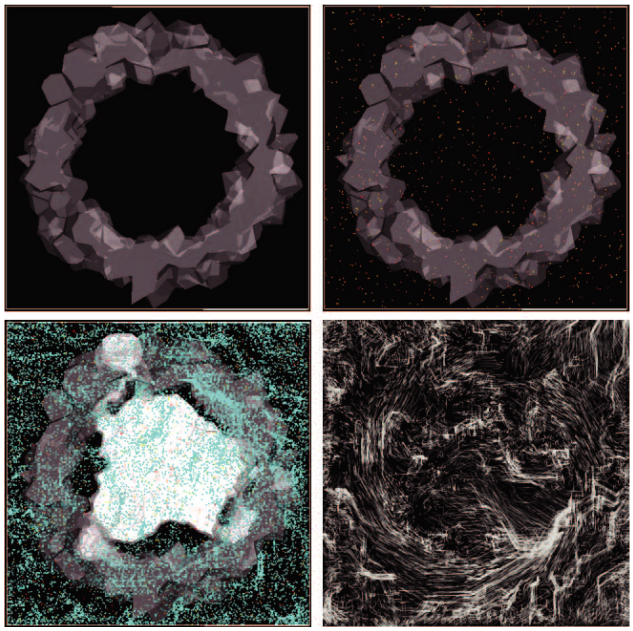

Fig. 17 Particle simulation. (Top left) The stationary variable is applied differently to only 253 indicated cells among all 1,000 Voronoi cells to create a circular vector field. (Top right) Wind vector is generated with graph-based WFC. (Bottom left) Simulation of 20,000 particles moving according to the wind vector. (Bottom) Trail of particles.

Table 4 Performance result of the 3D vector field experiment.

	N=300	N=1,000	N=3,000	N=6,000
Time (sec)	6.13±0.08	33.42±0.57	329.20±1.03	858.28±1.88
Backtracking (#)	0.00±0.00	0.00±0.00	0.00±0.00	0.00±0.00

The results of these four experiments show that the system proposed in this paper satisfies the user's input conditions and can arrange various irregularities in the 3D world. It took about 1 to 800 s to calculate the graph-based WFC for 100 to 10,000 nodes on an actual PC equipped with a Windows 10 OS, i7-7700K CPU, GTX 1080 Ti Graphics Card, and 32 GB RAM. We provided this interface in a web environment to enable fast production iterations. We can confirm that it is possible to generate the metadata required for planning quickly within 0.25 hours with 100~10,000 nodes in the map used in an actual game.

6. Conclusion

In this paper, we showed that the WFC algorithm, which is limited to an existing grid-based system, can be extended to a graph-based system. Our methodology can help to facilitate the use of PCG in CSP and graph space by taking advantage of the excellent content control offered by WFC. However, graph-based WFC in CSP has a drawback in that it can only solve problems irrespective of the order of the solution. To compensate for this, we must make more use of the search to solve the important order of the solution (cf. the river crossing puzzle). In addition, optimization is necessary because the calculation time is longer when the number of constraints increases in a general graph structure. In the future, we intend to apply this PCG generation technique to other content besides games, such as music and storytelling.

References

[1] World of Warcraft: https://worldofwarcraft.com/, accessed 17 July 2019.
[2] Blade and Soul: http://bns.plaync.com/, accessed 17 July 2019.
[3] No Man's Sky: https://www.nomanssky.com/, accessed 17 July 2019.
[4] Diablo3: https://us.diablo3.com/en/, accessed 17 July 2019.
[5] I. Karth and A.M. Smith, "WaveFunctionCollapse is constraint solving in the wild," Proc. 12th Int. Conf. Found. Digital Games, pp.1–10, ACM, 2017.
[6] I. Karth and A.M. Smith, "Addressing the fundamental tension of PCGML with discriminative learning," arXiv preprint arXiv:1809.04432, 2018.
[7] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," ACM Trans. Multimed. Comput. Commun. Appl. (TOMM), 9, 1, 2013.
[8] S. Snodgrass and S. Ontan, "Experiments in map generation using Markov chains," Found. Digital Games, 2014.

[9] S. Snodgrass and S. Ontan, "Learning to generate video game maps using Markov models," IEEE Trans. Emerg. Topics Comput. Intell., vol.9, pp.410–422, 2016.
[10] S. Dahlskog, "Patterns and procedural content generation in digital games: Automatic level generation for digital games using game design patterns," PhD dissertation, Faculty of Technology and Society, Malmö University, 2016.
[11] A.J. Summerville, S. Philip, and M. Mateas, "Mcmcts pcg4 smb: Monte Carlo tree search to guide platformer level generation," 11th Artif. Intell. Interact. Digital Entertain. Conf., 2015.
[12] S.E. Reed, Y. Zhang, Y. Zhang, and H. Lee, "Deep visual analogy making," Adv. Neural Inf. Process. Syst., pp.1252–1260, 2015.
[13] A.J. Summerville, M. Guzdial, M. Mateas, and M.O. Riedl, "Learning player tailored content from observation: Plat-former level generation from video traces using LSTMs," Artif. Intell. Interact. Digital Entertain. Conf., 2016.
[14] R. Jain, A. Isaksen, C. Holmgrd, and J. Togelius, "Autoencoders for level generation, repair, and recognition," Proc. ICCC Workshop Comput. Creativity Games, 2016.
[15] T. Xue, J. Wu, K. Bouman, and B. Freeman, "Visual dynamics: Probabilistic future frame synthesis via cross convolutional networks," Adv. Neural Inf. Process. Syst., pp.91–99, 2016.
[16] L. Horsley and D. Perez-Liebana, "Building an automatic sprite generator with deep convolutional generative adversarial networks," IEEE Conf. Comput. Intell. Games, pp.134–141, 2017.
[17] H.H. Kim, "Content generation using variational autoencoder," Nexon Dev. Conf., 2017.
[18] C. Beckham and C. Pal, "A step towards procedural terrain generation with GANs," arXiv preprint arXiv:1707.03383, 2017.
[19] E. Giacomello, P.L. Lanzi, and D. Loiacono, "DOOM level generation using generative adversarial networks," IEEE Games Entertain. Media Conf., pp.316–323, 2018.
[20] A.K. Hoover, J. Togelius, and G.N. Yannakis, "Composing video game levels with music metaphors through functional scaffolding," 1st Comput. Creativity Games Workshop, ACC, 2015.
[21] A. Khalifa, M.C. Green, D. Perez-Liebana, and J. Togelius, "General video game rule generation," IEEE Conf. Comput. Intell. Games, pp.170–177. IEEE, 2017.
[22] M. Sharif, A. Zafar, and U. Muhammad, "Design patterns and general video game level generation," Int. J. Adv. Comput. Sci. Appl., vol.8, no.9, pp.393–398, 2017.
[23] WaveFunctionCollapse: https://github.com/mxgmn/WaveFunction Collapse, accessed 17 July 2019.
[24] H. Scurti and C. Verbrugge, "Generating paths with WFC," 14th Artif. Intell. Interact. Digital Entertain. Conf., 2018.
[25] Bad North Steam: https://store.steampowered.com/app/688420, accessed 17 July 2019.
[26] City Generator: https://marian42.itch.io/wfc, accessed 17 July 2019.
[27] R. Bridson, "Fast Poisson disk sampling in arbitrary dimensions," Proc. ACM SIGGRAPH, 2007.
[28] Solving Every Sudoku Puzzle by Peter Norvig: http://norvig.com/sudoku.html, accessed 17 July 2019.
[29] Sudoku Solver Using Constraint Programming: https://gist.github.com/ksurya/3940679, accessed 17 July 2019.
[30] SolvingSudoku: https://twitter.com/greentecq/status/1037193248756957184, accessed 17 July 2019.
[31] Three-pathfinding: https://github.com/donmccurdy/three-pathfinding, accessed 17 July 2019.
[32] Voro, https://github.com/jimmyland/voro, accessed 17 July 2019.
[33] Voro++, http://math.lbl.gov/voro++/, accessed 17 July 2019.
[34] J.R. Stuart and P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2009.
[35] ECLiPSe, http://eclipseclp.org/, accessed 12 March 2020.
[36] S. Dymchenko and M. Mykhailova, "Declaratively solving tricky Google code jam problems with prolog-based ECLiPSe CLP system," 30th ACM/SIGAPP Symposium on Applied Computing, 2015.

Hwanhee Kim received a Bachelor's degree from the Department of Urban Planning & engineering from Yonsei University in 2007. From 2011, he joined Nexon Korea as a game designer. From 2017, he is working at NCsoft Korea as a senior game designer.

Teasung Hahn received a Bachelor's degree from the Department of Newspaper Broadcasting of from Yonsei University in 2007. He is working at NCsoft Korea as an lead game programmer.

Sookyun Kim received Ph.D. in Computer Science & Engineering Department of Korea University, Seoul, Korea, in 2006. He joined Telecommunication R&D center at Samsung Electronics Co., Ltd., from 2006 and 2008. He was a professor at Department of Game Engineering at Paichai University, Korea. He is now a professor at Department of Computer Science at Jeju National University, Korea.

Shinjin Kang received an MS degree from the Department of Computer Science & Engineering from Korea University in 2003. After graduation, he joined Sony Computer Entertainment Korea (SCEK) as a game developer. From 2006, He has worked at NCsoft Korea as a lead game designer. He received Ph.D. degree in Computer Science & Engineering at Korea University in 2011. And he is now a professor at the School of Games in Hongik University.