ANALYSIS OF THE PROCEDURAL GENERATION ALGORITHM OF EDGAR PRO TOOL WITH A FASTER VARIETY CREATION IN DUNGEON LAYOUT

Cristina Souza de Araujo[1], Ana Carolina Brandão Salgado[1], Gabriel Barroso da Silva Lima[2], Jucimar Maia da Silva Junior[2], Clairon Lima Pinheiro[2] and Luis Cuevas Rodriguez[2]
[1]CESAR SCHOOL
Recife, Brazil
[2]Escola Superior de Tecnologia - EST
Universidade do Estado do Amazonas - UEA, Manaus, Brazil

ABSTRACT

This article describes the process of analysis the procedural generation package Edgar for Unity game engine. This package focuses on generating dungeon layouts for top-down 2D games. The generation works with the developer creating room models and a non-directional graph for the dungeon, with the package generating a random layout from these inputs. The purpose of this article is to expand the variety of layouts that the package generates, creating a greater variety of results for the player without losing the randomness factor that comes from procedural generation.

KEYWORDS

Procedural Generation, Games, Unity, Game Development

1. INTRODUCTION

During the game design process, many elements are taken into account regarding game development: mechanics, UX, UI, platforms, progression, accessibility [Motta, Junior 2013]. One of these elements refers to the replayability factor of the game, that is, how much new content the game can offer for each new playthrough. Many games offer unlockable difficulties, new game modes, or achievements to be obtained. However, other games go beyond that.

Roguelike games have two characteristics that differentiate them from other genres: one is permanent death, which refers to the loss of progress (or part of it) when the player loses; and the other is procedural generation of new maps for each playthrough, allowing new experiences to be enjoyed by the player in every new game [Cerny, Dechterenko 2015]. The type of procedural generation can vary greatly depending on what the game design of the project requires and the game engine used can do.

The Edgar procedural generation package is a tool for creating dungeon layouts focused on top-down 2D games [Nepožitek 2018]. With it, it is possible to create and edit rooms and assemble the layout of the dungeon to be generated, allowing the developer to procedurally generate maps without losing control of the game's level design. This tool was based on the work of Chongyang Ma, where many of the concepts and techniques used were presented [Ma, Vining, Lefebvre, Sheffer 2014, May]. This article describes the process of replicating the artificial intelligence used by the Edgar package in its procedural generation of 2D dungeons, but applying techniques that avoid creating similar layouts in near generations, making the player to feel a greater variety of results that the algorithm can provide.

1.1 Justification

The Edgar procedural generation package is a great tool for 2D dungeon procedural generation, but in Ondřej Nepožitek's article, there is no mention of something that tries to avoid creating layouts similar to those created

in past generations, which can be a problem, especially in games that have procedural generation as a fundamental part of their design (such as roguelike games).

To solve this problem, this article presents an approach to the procedural generation technique of the Edgar package, but applying validation functions that verify and ensure that new layouts are visually different from dungeons generated in past generations. This allows the variety of maps to be better explored and, therefore, increases the replayability of the game, giving more value to the final product. The algorithm works based on tilesets. This working method was chosen because it makes replication of the Edgar package less complex, in addition to aligning with the results of the Edgar package.

2. DUNGEON ELEMENTS

This algorithm works in three parts: the first part involves obtaining configuration spaces between rooms, which are the positions where a room connects correctly to another; the second part involves breaking the planar graph of the dungeon layout into subgraphs, determining the generation priority order between the obtained subgraphs; and the third part involves generating the rooms from the subgraphs, using the configuration spaces to determine the position of a new room and whether a generated layout is valid or not.

To compare with previously generated dungeons, points such as the angle of the new room with the room it connects to, similarity between the arranged rooms, the number of connections the room has, and how old the old layout being compared is are checked. All these factors contribute to assigning a value that determines whether a new room is different enough to be arranged or whether a new generation needs to be made. These steps contribute to generating a dungeon efficiently, with the minimum processing and time required and effectively.

The dungeon layout graph is the logical representation of the desired dungeon progression, containing information about which rooms (or set of rooms) connect with each other. This information is loaded into the generation algorithm, which translates these values into a procedural dungeon.

Since rooms cannot overlap, it is required that the dungeon layout can be represented in a planar graph. In a planar graph, its edges can be rearranged so that there are no edge crossings [Bondy, Murty 1976]. In addition to being planar, the layout must be simple enough for the generation algorithm to create a dungeon with great variety and without major issues. A complex graph or one with more restrictions will eventually cause greater processing time, and graphs that are impossible (with constraints that prevent any possibility of dungeon creation) should be avoided at all costs.

The dungeon is a connected set of rooms, and each room, like the dungeon itself, also has its properties. Rooms can have distinct characteristics such as enemies, bosses, shops, customizable events, but they all have two fundamental parameters: the list of corners and the list of openings. Since rooms are composed of tiles, both corners and openings must be aligned to the game grid.

The list of corners informs the position of the corners and delimits the area of the room. The wall of a room is determined by the line that starts from a corner to the next corner on the list (in the case of the last corner, it connects to the first corner, closing the area of the room). The list of openings informs the allowed areas for connection, which are attached to the walls and inform which walls are successive to be replaced by a connection through a corridor. These two properties are crucial for the elaboration of configuration spaces.

Corridors are special types of rooms whose purpose is to connect rooms in a dungeon. They are characterized by being rectangular and having their openings at their respective ends: horizontal corridors have openings only on the left and right walls, and vertical hallways have openings on the top and bottom walls.

Before starting the generation process, a calculation is made based on the corridors inputted by the developer, determining the size of the openings, the width, and orientation of the hallways. This is important information that allows for the creation of a greater variety of possibilities for room connection.

3. CONFIGURATION SPACES

Configuration spaces are positions where two rooms can connect correctly without one overlapping the other [Nepožitek 2018] [Ma, Vining, Lefebvre, Sheffer 2014, May]. This greatly speeds up procedural generation as

it is not necessary to randomly place the room and check if the connection is valid, just place the room in one of the available configuration spaces.

To obtain the configuration spaces, one room must remain static while the other moves around it. During the movement, the positions where the moving room is correctly connected to the static room (intersection greater than or equal to minimum size and no overlap between the rooms) are recorded. At the end of the process, all the found positions are saved to be accessed later during the dungeon generation.

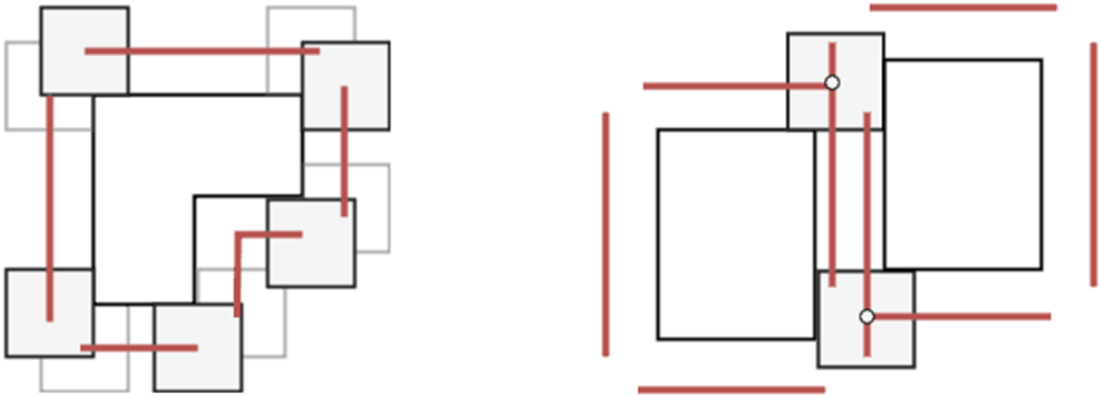

Figure 1. Configuration spaces: room move around fixed room, getting all valid connection positions. A configuration space intersection means a room can connect to more than one placed room

3.1 Configuration Space Creation

The process begins with the selection of two rooms, one static and one mobile. Initially, both are placed on top of each other, then the mobile room is moved one unit upwards until there is no more overlap, and starts to move around the static room. The rotation of the mobile room around the fixed room is straight until it encounters an overlap or the mobile room is no longer in contact with the fixed room. This loop continues until the room returns to its initial position, at which point the process is finished with all configuration spaces saved.

At each position of the mobile room, two checks are made: the first is to analyze whether the mobile room overlaps with the fixed room or there is no contact between the two rooms. If neither of these cases occurs, then the openings between the two rooms are checked. Each saved configuration space has information about the opening, which is important during the placement of corridors after the dungeon layout is finalized. configuration spaces are saved in the fixed room.

For each valid connection, a search is made for available corridors, taking into account the orientation of the contact. This is done so that during the arrangement of the rooms, it is not necessary to correct the position to accommodate the corridors, as the positions already allow this naturally. For example, if a valid position is found and this valid position is on an upper wall in the fixed room, it is understood that the mobile room should move upwards, as it is the direction in which it moves away from the fixed room. It is also understood that this connection occurs in the vertical orientation, as it is the orientation in which the room moves.

All saved configuration space has information about the opening, which is important during the placement of corridors after the dungeon layout is finalized. This information includes: connection orientation, room orientation, distance from the room, opening size, socket values of the openings in contact, and the position where the corridor should be placed for a perfect connection.

3.2 Wall Intersection Check

The wall intersection check between the rooms is extremely important for obtaining the configuration spaces, since in order for a position to be valid for connection, it is necessary to meet some prerequisites, which are: not overlapping the other room and the number of connecting openings is at least the minimum required. Each of these prerequisites has its own functions that perform this check.

For each point, three checks are performed: if the point is far enough to discard any comparison; if the point position is on the wall of the static room (position passed if it falls between the two corner values) and if the point is inside the static room (even-odd rule [Foley, Van, Van Dam, Feiner, Hughes 1996]). These checks are performed in this order and, if one of them is true, it is not necessary to perform subsequent checks.

These functions use techniques to speed up the checking process as much as possible, avoiding unnecessary repetitions and comparisons. Basically, they involve traversing the points of the walls of the mobile room and comparing them to the points of the walls of the static room. Because the rooms are located within a grid, it is not necessary to analyze decimal values since each room can only be located within the grid, with its position being an integer value. The points of a wall are the integer values that the wall occupies: if a room has a wall that goes from (0, 0) to (0, 6), the points of the wall are all the integer values that are on that line, in this case: (0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), and (0, 6).

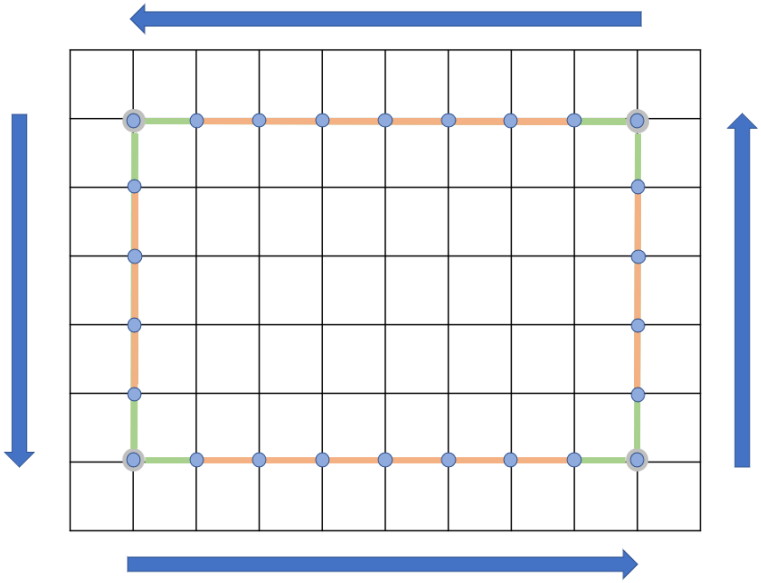

Figure 2. Starting from the first corner, all wall points are traversed until all points of the grid that the walls of the room occupy have been checked

The second and third checks help to decide the direction that the mobile room will take in the next iteration: if no contact is identified between the rooms or there is overlap, the room returns to the previous position and follows the next path. If there is contact without overlap, then the algorithm is able to check the intersection of the openings of the rooms.

3.3 Opening Intersection Check

If the wall intersection check verifies the current position of the room is a valid position, the opening intersection check verifies if the current position of the room is valid for capturing configuration spaces from the fixed room to the mobile room. This is validated from the intersections between the openings of the rooms where the largest consecutive intersection determines if the current position is a configuration space or not.

The verification is done similarly to the wall intersection check of the rooms with the difference that, instead of the function traversing point by point the walls, it traverses the openings of the mobile room and checks if they are within reach of the openings of the fixed room. Another difference is that, in this case, the function seeks to track the longest sequence of points that intersect the openings of the fixed room. A valid position for capturing configuration spaces is characterized when the longest sequence of intersections found is equal to or greater than the width of the opening defined by the room plus one.

4. GRAPH DECOMPOSITION

The ordering of the chains is done as follows: all cycles within the graph are identified using a technique similar to graph coloring [Jensen, Toft 2011], where all nodes are traversed, saving the addresses of the visited nodes, and when a repetition is detected, an array is saved in a list of found cycles with all addresses from the repetition to the last visited node. This list is then sorted from the array with the smallest size to the largest.

The ordering of the subgraphs has as priority the smallest cycle found, followed by the smallest cycles adjacent to the previously arranged cycles. When there are no more cycles connected to the subgraphs, a breadth-first search [Beamer, Asanovic, Patterson 2012] is performed where each node connected to the obtained subgraphs is arranged one by one into the set of subgraphs. If a new cycle is found, the process of inserting cycles into the list of chains is repeated. The process of breaking the graph into subgraphs is terminated when all elements of the received graph are within one of the chains in the created list.

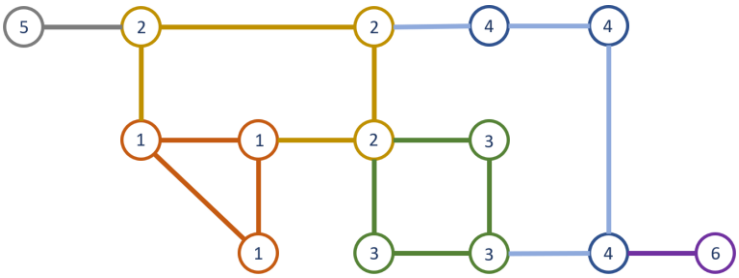

Figure 3. Order of breaking the graph into subgraphs, starting from the smallest cycle, then adding the smallest cycle connected to the previous ones, and finally the acyclic nodes

5. DUNGEON LAYOUT CREATION

The ordering of the chains is done as follows: all cycles within the graph are identified using a technique similar to graph coloring [Jensen, Toft 2011], where all nodes are traversed, saving the addresses of the visited nodes, and when a repetition is detected, an array is saved in a list of found cycles with all addresses from the repetition to the last visited node. This list is then sorted from the array with the smallest size to the largest.

5.1 Finding a Room Position

When receiving a chain, the algorithm first allocates the rooms in their proper positions. All rooms are based on the configuration spaces of the rooms they connect to, in order to define their position (the first room of the first current will have (0, 0) as position).

New rooms choose a random valid position among those listed in the configuration spaces (without overlapping with other rooms). However, for rooms that connect to only one already-disposed room, an evaluation is also made that checks how different each position is from layouts created in both this generation and past generations. This evaluation assigns a weight to the position. The higher the value, the more different the new room in that position is from other layouts. For a position to be considered different enough, it must have a minimum weight of 1.

The position selection function stores the list of configuration spaces to connect in a backup list. In case there are no intersections between configuration spaces (in cases where rooms connect to more than one room already placed) or if no valid positions are found during the position verification, the function returns a random position from the created backup, already indicating that the returned position did not meet the acceptance criteria.

5.2 Old Layouts Comparison

For each valid position available for the new room, a weight is assigned to determine if the position is different enough from other layouts created, both in the current generation and in past generations. This function takes into account three factors: the angle of the room in comparison to the room it connects to, if the comparison is between two instances of the same room, and the generation of the layout to which the room is being compared.

Initially, the weight and multiplier of rooms from past generations are calculated. The initial value of the weight is the angle of the new room's central point with the central point of the room it connects to. This angle is defined from the horizontal axis: with the center of the room exactly to the left of the center of the room it connects to, without being slightly up or down, its angle is 0, and as the room moves counterclockwise, the angle value increases: being below the room it connects to, the angle is 90, to the right, 180, up, 270, until returning to 0.

The multiplier of this weight defines how lenient or strict the comparison will be. Larger values make the weight assignment more lenient, while smaller values make it more restrictive. Older layouts or rooms not identical to the room being checked tend to the multiplier to a larger value, while newer layouts and identical rooms tend to a smaller value.

Having saved the angles of the rooms and their multipliers, the function stores, in a new list, the distances of all the rooms, in angles, from the saved layouts, and then removes all indices whose distances are greater than double the average distance, avoiding values that may negatively contribute to the desired distance calculation. After that, the calculation of the angle of the new room is made, using the same principle applied to the other rooms. The more distant the new room angle is from the other layouts angles, the higher the result of the division.

5.3 Biased Random Choice

To choose the position of a room based on its weight, a biased random choice is made where the higher the weight of the room, the greater its chances of being chosen. First, the sum of all weights is stored and a variable is created that receives a random value between 0 and the sum of the weights.

The function then creates a zeroed variable that will help to iterate through the positions' weights: if this variable plus the current weight is equal to or greater than the obtained random value, the function returns the current index. Otherwise, the variable receives itself plus the weight value and the same verification is done for the weight of the next index. This method ensures that larger weights have a greater chance of being chosen, and to further reinforce this aspect, all weights are squared before calling this function, so that larger weights become even more valuable compared to smaller weights.

5.4 Chain Energy

After the rooms in the chain are placed in their defined positions, the energy of the current chain is calculated. This energy is defined based on two factors: the total area of overlap between the rooms and the squared distance between connected rooms in the dungeon graph that are not connected in the generated layout. The energy of a chain refers to how close the set of placed rooms is to becoming a valid set, i.e., correctly connected. The calculation of the chain is given by the following function [Ma, Vining, Lefebvre, Sheffer 2014, May]:

$$E = e^{A/m} \cdot e^{D/m} - 1$$

In this equation, 'E' represents the total energy; 'A' represents the total overlap area; 'D' represents the sum of squared distances of non-connected rooms, and 'm' represents how much the function tends to accept higher energy chains. Is recommended a value of 'm' that allows for greater chances of creating a valid chain without compromising the optimization of the code too much. In the article "Game Level Layout from Design Specification" by Chongyang Ma, it is mentioned that empirically, a value of one hundred times the average area of the rooms was found to satisfy both proposed requirements.

For 'D' the process is simple: it is possible to create a triangulation with the center of the two unconnected rooms in order to form a right triangle, and from the Pythagorean theorem formula a^2 + b^2 = c^2 [Maor 2019].

For ‘m’, the algorithm counts all blocks occupied by the room, starting from the uppermost line to the lowers. The area of a room is the sum of all blocks occupied by each row. Then, after getting the area of all rooms, the result is divided by the room amount and then multiplied by one hundred to get the desired value.

For ‘A’, The process starts at the first corner of a room, traversing, point-by-point, the walls of the room. When it identifies an overlap with the other room, that is, the current point is inside the other room, the function starts storing all the corners that it traverses in a list. When it is identified that the current point is no longer inside the other room, the previous position is stored as a corner, and the function starts traversing the wall of the other room in the direction where the point is inside the first room. After getting all intersection corners, the area is calculated using the same method to get the ‘m’ value. ‘A’ is the sum of all overlap areas.

5.5 Simulated Annealing

Simulated annealing is an optimization metaheuristic that seeks to find a low-cost solution without excessive time cost. Its characteristic is the possibility of accepting higher cost results to avoid minimal locals, and this probability decreases with each iteration [Kirkpatrick, Gelatt Jr, Vecchi 1983].

After arranging the rooms in a chain, its energy is calculated, and the chain becomes the reference layout for future changes. If the chain has an energy higher than 0 (not a valid layout), a change is made in a random room of the chain. After that, the energy of the layouts is compared: if the energy of the altered layout is lower, it becomes the reference for creating new layouts; otherwise, it can still be chosen as the reference if the following condition is met [Ma, Vining, Lefebvre, Sheffer 2014, May]:

$$e^{-\Delta E/(a*t)} < Random(0\sim1)$$

In this condition, 'ΔE' represents the energy difference between the reference layout and the new layout, 'a' represents the average of energy differences during the creation of layouts based on other references. 't' represents the current temperature of the algorithm and *'Random (0~1)'* is a random number between 0 and 1 generated during the comparison. The 't' variable decreases with each cycle of comparisons, reducing the chance of layouts with higher energy costs being accepted as a reference.

5.6 Corridors Placement

With all rooms arranged in their respective positions, the position of the corridors that connect the rooms can be defined. In the configuration spaces of the rooms, each position returns a list of information on how the corridor should behave for the chosen position specifically.

During the creation of the configuration spaces, two directions are saved that are important to define the position of the room. The first is the direction in which the mobile room was going, and the second is the direction in which the room moves away from the fixed room. These two directions define which opening point the corridor will use as a reference when defining its position. All corridors have four opening points: the two that form the left opening and the two that form the right opening in the case of horizontal corridors, and the two at the bottom opening and the two at the top opening in vertical corridors.

6. RESULTS AND FUTURE WORKS

For the generated dungeon, the following characteristics were determined: the first room will always be the starting room, the following rooms will consist of a normal package of three rooms, and the last room will always be the final room. In the graph, the second and third rooms connect to the first, the fourth room connects to the second and third, the fifth and sixth rooms connect to the fourth, the seventh connects to the fifth and sixth, and the eighth room connects to the seventh.

Figure 4. Dungeons generated by passing the room packages, their connections, and corridors.

Having the algorithm already capable of creating dungeons from inputs defined by the developer, the next step is to apply it to a game in development. By being inserted into the logic of a game, it will be analyzed whether the algorithm can consistently, quickly and efficiently meet the requirements of the game designer.

Another step is to directly compare the results of the algorithm with the results produced by the procedural generation package Edgar, replacing the package in a game that uses it, comparing the dungeons and verifying if the algorithm presented in this article succeeded in generating greater varieties of dungeons in fewer interactions.

ACKNOWLEDGEMENT

The authors thanks to the Universidade do Estado Amazonas (UEA), Transire Eletrônicos, Tec Toy S.A. and Ludus Lab for their support. The results were published through the research and development activities of project ARKADE AD ASTRA, sponsored by Transire Eletrônicos and Tec Toy S.A, with the support of SUFRAMA under the terms of Federal Law No.8.387/1991.

REFERENCES

Baghdadi, W., Eddin, F. S., Al-Omari, R., Alhalawani, Z., Shaker, M., & Shaker, N. (2015). A procedural method for automatic generation of spelunky levels. In *Applications of Evolutionary Computation: 18th European Conference, EvoApplications 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings 18* (pp. 305-317). Springer International Publishing.

Beamer, S., Asanovic, K., & Patterson, D. (2012, November). Direction-optimizing breadth-first search. In SC'12: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (pp. 1-10). IEEE.

Bondy, J. A., & Murty, U. S. R. (1976). *Graph theory with applications* (Vol. 290). London: Macmillan. (pp. 135-139).

Cerny, V., & Dechterenko, F. (2015). Rogue-like games as a playground for artificial intelligence–evolutionary approach. In *Entertainment Computing-ICEC 2015: 14th International Conference, ICEC 2015, Trondheim, Norway, September 29-October 2, 2015, Proceedings 14* (pp. 261-271). Springer International Publishing.

Foley, J. D., Van, F. D., Van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). *Computer graphics: principles and practice* (Vol. 12110). Addison-Wesley Professional. (pp. 127)

Itch Corp (2023). *Most used Engines*. itch.io/game-development/engines/most-projects

Jensen, T. R., & Toft, B. (2011). *Graph coloring problems*. John Wiley & Sons.

Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, *220*(4598), 671-680.

Ma, C., Vining, N., Lefebvre, S., & Sheffer, A. (2014, May). Game level layout from design specification. In *Computer Graphics Forum* (Vol. 33, No. 2, pp. 95-104).

Maor, E. (2019). *The Pythagorean theorem: a 4,000-year history* (Vol. 65). Princeton University Press.

Motta, R. L., & Junior, J. T. (2013). Short game design document (SGDD). *Proceedings of SBGames*, 2013, 115-121.

Nepožitek, O. (2018). Procedural 2D Map Generation for Computer Games.