Procedural Dungeon Generation Analysis and Adaptation

Jessica R. Baron
Clemson University School of Computing
Clemson, SC 29634
jrbaron@clemson.edu

ABSTRACT

The objective of this article is to analyze several approaches that produce a particular form of procedural content in games - dungeon maps - and adapt the work to both 2D and 3D game projects. Procedural video game content provides unique levels for users to experience, reduced allocated memory, and more time for developers to focus on other aspects of a game. The work presented consists of two parts: the implementation and analysis of five different algorithmic approaches that generate a 2D dungeon map, consisting of traversable rooms and corridors; and the application of the original code to a 3D game environment in Epic Game's Unreal Engine 4.

ACM Reference format:
Jessica R. Baron. 2017. Procedural Dungeon Generation Analysis and Adaptation. In *Proceedings of ACM SE '17, Kennesaw, GA, USA, April 13-15, 2017,* 5 pages.
DOI: http://dx.doi.org/10.1145/3077286.3077566

1 INTRODUCTION

1.1 Context and Related Work

Procedural content is created algorithmically versus manually and is commonly used in computer graphics and video games. Rogue (1980), developed by Michael Toy and Glenn Wichman, is one of the first games to use run-time procedural dungeon generation, and games following it have been considered "Rogue-like" in their procedural generation [6] [5]. With modern resources, procedurally generated maps allow game developers to devote more time for creating other aspects of games, such as developing more complex story and gameplay [4]. The first games with procedurally generated levels used ASCII characters to represent the dungeons, similar to the implementation of the analysis part of this paper. However, increased computer processing power and memory accessible by modern games allow procedural generation using high-resolution original artwork, allowing for more appealing and organic maps such as in Minecraft (2009) and Diablo (1996) [11].

1.2 Project Overview

The algorithms presented cover a brief survey of approaches to this problem: a brute force implementation, the binary space partitioning data structure [10] [7], and an application of cellular automata [8]. Each approach is subject to a standard set of rules. The 2D graph can be set to dimensions between 30x30 vertices and 500x500 vertices, but for testing purposes all algorithm pairs use a 60x60 graph and create connected graphs. Testing each pair produces standard deviation percentages from a set ratio between "on" and "off" vertices, and these deviations across the different algorithm pairs differ slightly from one another, since each pair has its own unique pattern in creating rooms and corridors. The final appeal of the composition of a graph depends on the map's purpose and the user's goals; multiple approaches to generating dungeons exist, and the author narrowed focus on generating a particular kind of map for the analysis part of this project. The second part of the project focuses on adapting one of the algorithm pairs to a 3D UE4 project that has the atmosphere of an adventure game.

1.3 Formal Problem Statement

Procedurally generate a 2D map composed of rectangular "rooms" connected by narrow "corridors" using a width of W vertices and a height of H vertices. Each vertex is assigned a state: "on" (traversable rooms and corridors) or "off" (untraversable boundaries, or, "walls"). The "on" vertices of the graph are connected, meaning each "on" vertex is adjacent to at least one other "on" vertex, guaranteeing that all traversable vertices ultimately lead to one another. The graph should reflect a desired sparsity ratio between "on"- and "off"-vertices, which is defined as approximately 0.4 of the graph vertices as "on" and 0.6 as "off," targeting a 0.1 deviation from that ratio. A "room" is a subset of the set of graph vertices that consist of "on"-vertices adjacent to one another and that form a sub-graph; the width and height of a room are constrained by minimum and maximum values that are fractions of the graph's width and height. A "corridor" is also a sub-graph where either its width or its height must be equal to one. Corridors are generated after rooms have been created and connect two rooms at a time. A room cannot be connected to another room, but it must be connected to at least one corridor. A corridor must be connected to at least one other corridor or room, and if it is connected to a corridor, the linked corridor must be connected to a room.

2 ALGORITHMS AND ANALYSIS

2.1 The Algorithms

Five algorithms employ a base graph class in this project, each being either room-generating or corridor-generating, and they are combined to create five unique, map-generating algorithm pairs. The room-generating algorithms are named Random Room Placement and BSP Room Placement, and the corridor-generating ones are Random Point Connect, Drunkard's Walk, and BSP Corridors.

ACM SE '17, Kennesaw, GA, USA

DOI: http://dx.doi.org/10.1145/3077286.3077566

2.2 The Base Graph

The base graph class owns a 2D array of "vertex" objects that each hold a Boolean state value of "on" (true) or "off" (false), the default state being "off", and a flag depicting which type of graph component the vertex belongs to: a room, wall, or corridor. The graph is initialized as a blank dungeon map, represented by "off" vertices.

2.3 Room-Generating Algorithms

Rooms are generated before corridors to assure the whole map is connected by corridor start and ending coordinates being dependent upon the locations of rooms. This subsection describes the two approaches taken to populate the graph with rectangular rooms.

2.3.1 Random Room Placement. Random Room Placement is a brute force room-generating algorithm. It is implemented by randomly generating a room width and height (within limits), then randomly picking a vertex (x, y) on the graph that will be the top-left corner of the room to place. This potential room is then checked for overlapping with other "on"-vertices of the graph. If it will not overlap, the room is "drawn" in the graph, starting at the top-left corner vertex. Otherwise, attempt to place the same room again but at different vertex locations until succeeding or reaching a timeout for placing that particular room in the graph. An additional method checks for overlapping other rooms or the borders of the graph plus a small space of extra vertices horizontally and vertically around graph borders and other rooms. Once a room is created, it is added to an array. This algorithm continues in a loop until the percentage of the graph composed of "on" (specifically, "room") vertices reaches a particular value that is passed as a parameter. This value is different according to the corridor-generating algorithm with which this room-generating one is paired. In general the algorithmic efficiency of the Random Room Placement function is O(1), but with certain key variables having restricted maximum values, a more specific analysis results in O(w*h) (also constant), where w and h are the maximum room dimensions.

2.3.2 BSP Rooms. The BSP Room Placement algorithm is an implementation of binary space partitioning. It partitions the graph into "leaf" objects, each of which is stored in a tree data structure, starting with the entire graph as the "root" of the tree. The leaves are split until a split leaf has children of a defined minimum size, which is directly related to the graph's dimensions and designed to prevent room overlapping. A bottom leaf has no children as it does not get split. The bottom childless leaves are assigned rooms (the maximum size of which is smaller than the minimum leaf size), and these rooms are added to an array-list in the order of which the leaf objects are created. The efficiency of the BSP Room Placement algorithm in general is O(1) (constant) but specifically in the author's implementation, it is O(n+c+(w*h)), a more exact constant value, where n is number of splits, c is number of children (superset of leaves), and w and h are the maximum room dimensions.

2.4 Corridor-Generating Algorithms

Once the graph consists of rooms, corridors can be generated by connecting those rooms to each other. Random Point Connect and BSP Corridors produce relatively straight paths of corridors, and Drunkard's Walk results in twisting lines with many turns.

2.4.1 Random Point Connect. This corridor-generating algorithm accesses the list created by one of the room-generating algorithms and connects two rooms at a time, which are consecutive items in the room list. The algorithm connects rooms in a brute force manner, selecting a random vertex each bordering the two rooms and linking them the entire horizontal distance and then the entire vertical distance (or vice versa; the starting direction is randomly decided). Overlapping other corridors and rooms with these new corridors is ignored. Random Point Connect produces different results according to how the array of rooms is ordered. For example, a room list created by the Random Room Placement algorithm has no relation between consecutive elements in the list as randomly generated room are added as they are successfully placed, therefore connecting them with Random Point Connect will often have more overlapping of corridors. A room list created by BSP Room Placement is created per bottommost leaf objects, whose order is reflected by how the BSP tree splits. Therefore, when these rooms are linked, the corridors produced are often shorter and do not overlap as much as with a room list from Random Room Placement. The efficiency of the Random Point Connect algorithm is O(1), and the specific case is O(n*(W+H)), a constant value, where the main iterations (n) are capped at a constant due to the number of rooms, and W and H are the entire graph dimensions (being the maximum values a corridor can expand to).

2.4.2 Drunkard's Walk. The Drunkard's Walk algorithm is an application of cellular automata, inspired by an original Drunkard's Walk (also called "Random Walk Cave Generation") algorithm used in creating cave structures rather than rooms and corridors [1]. The author's version of the Walk is used as a winding room-connecting, corridor-generating algorithm. As with Random Point Connect, this algorithm accesses the populated room list and connects pairs of rooms by randomly selecting a bordering vertex each of the two rooms. This algorithm varies from the previous corridor-generating one as it starts at the first room's selected vertex and proceeds in the appropriate x and y directions (whether positive or negative) towards the second chosen vertex, incrementing a randomly chosen amount of vertices (between 2 and 6) horizontally or vertically with each iteration. Direction is inverted if path reaches the graph's borders. Drunkard's Walk will continue to iterate until it reaches the other point, or when too many steps have been taken. (The latter may happen if the Walk is trapped at a corner of the graph and may reach an infinite loop if unchecked. If the number of steps does exceed the certain defined limit, the Walk will continue from the current vertex to the destination vertex in a brute force manner to ensure connectedness.) This algorithm produces the most varying results, ranging from sparse maps where corridors found their destinations quickly, to cave-like maps where the algorithm carved winding corridors in taking many iterations to find the destination vertex. The general efficiency of the Drunkard's Walk is O(1), and in the specific implementation for this project, the efficiency is O(n*m(W+H)), where n is the number of main loop iterations, which is based on the maximum number of rooms generated, m is the iterations of the while loop that iterates until the walk reaches another room or hits limit (which has variability but is also constant due to the timeout), and W and H are the graph dimensions (when the walk might travel those entire distances in one step).

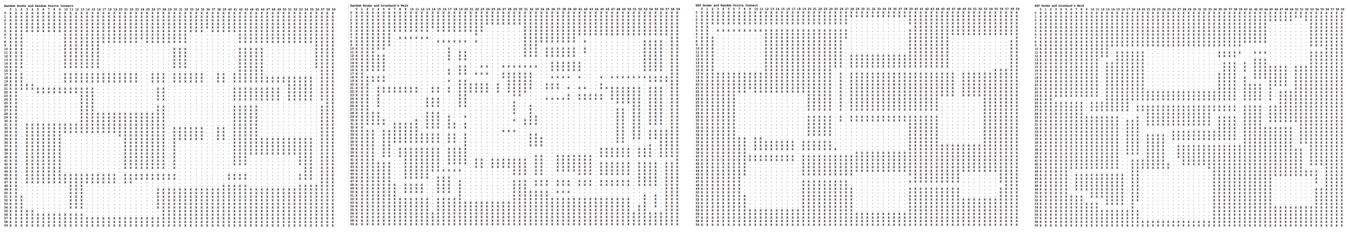

Figure 1: Random Room Placement and Random Point Connect, Random Room Placement and Drunkard's Walk (Average), BSP Room Placement and Random Point Connect, and BSP Room Placement and Drunkard's Walk (Sparse).

2.4.3 BSP Corridors. BSP Corridors must be paired with the BSP Room Placement algorithm as its functionality is within the classes that handle the binary space partitioning and relies on information gathered in BSP Room Placement. As with the previous corridor-generating algorithms, BSP Corridors connects room pairs from the list of rooms and, with each pair of rooms to connect, selects random border vertices each from the two rooms. BSP Corridors has a very similar implementation to Random Point Connect but varies slightly as it implements a possibility of one or two connections between a single pair of rooms. However, the largest difference between those two corridor-generating algorithms is how the pairs are determined. BSP Corridors connects rooms of bottommost leaf objects that share the same parent leaf. Each pair of sibling leaves is also connected to one other pair of siblings. Consequently, the graph result often has very orderly and linear paths bridging together the rooms. The algorithmic efficiency the of BSP Corridors function is O(1) (constant), and, more specifically in the context of this project, it is O(3c*(W+H)), where the constant 3c represents calculations per child, per corridor part (maximum of 4), per pairings (as the number of parents, c/2), and W and H are the graph dimensions.

2.5 Pairing the Algorithms

Each of the five algorithm pairs are implemented by combining the previously described functions. Considering efficiencies, for all algorithms, the maximum distances of rooms and corridors are directly related to the graph's dimensions. Therefore, if those dimensions are capped at maximum values set by the developer, the distances used in the efficiency analysis, w and h, are capped, and drawing along those distances would be constant: O(1).

2.5.1 Random Room Placement and Random Point Connect Analysis. When paired with Random Point Connect, the maximum percentage parameter passed to Random Room Placement is 0.4. The rooms produced should compose no more than 0.4 of the graph's vertices, since the amount of vertices that compose the corridors of the graph when generated by Random Point Connect will not be too large. The leftmost screenshot in Fig. 1 is of the basic console output of this algorithm pair. The efficiency of this pairing is the summation of the efficiencies evaluated for Random Room Placement and Random Point Connect. Therefore, in general, the overall algorithmic efficiency is O(1) but is specifically O(w*h) + O(n*(W+H)) = O(w*h + n*(W+H)).

2.5.2 Random Room Placement and Drunkard's Walk Analysis. When paired with Drunkard's Walk, the maximum percentage parameter passed to Random Room Placement is 0.15 (the rooms should compose no more than 0.15 of the graph's vertices), anticipating a large amount of vertices to compose the corridors of the graph when generated by Drunkard's Walk. The second dungeon in Fig. 1 is an average, balanced result of this algorithm pair. The efficiency of this pair of algorithms is O(1) but specifically is O(w*h) + O(n*m*(W+H)) = O(w*h + n*m*(W+H)).

2.5.3 BSP Room Placement and Random Point Connect Analysis. When paired with Random Point Connect, the minimum leaf size modifier parameters passed to BSP Room Placement is 1/15 of the graph's width and 1/15 of the graph's height. More rooms can be placed in the graph due to being able to split more leaves, due to the amount of vertices that compose the corridors of the graph when generated by Random Point Connect being a smaller fraction of the total "on" vertices. A result of this algorithm pair is captured in the third image in Fig. 1. In general, the algorithmic efficiency of this pairing is O(1), but it is specifically, with ns = splits and nr = rooms, O(ns+c+(w*h)) + O(nr*(W+H)) = O(ns+c+(w*h) + nr*(W+H)).

2.5.4 BSP Room Placement and Drunkard's Walk Analysis. When paired with Drunkard's Walk, the minimum leaf size modifier parameters passed to BSP Room Placement is 1/8 of the graph's width and 1/8 of the graph's height (less rooms can be placed due to splitting less leaves with the greater minimum leaf size), anticipating a potentially large amount of vertices to compose the corridors of the graph when generated by Drunkard's Walk. The last dungeon captured in Fig. 1 is a sparse result, when the Walk found its destinations relatively quickly, which happened in about a third of the trials. The general efficiency of the BSP Room Placement and Drunkard's Walk algorithms together is O(1) and is specifically O(ns+c+(w*h)) + O(nr*m*(W+H)) = O(ns+c+(w*h) + nr*m*(W+H)).

2.5.5 BSP Room Placement and BSP Corridors Analysis. When paired with BSP Corridors, as with Random Point Connect, the minimum leaf size modifier parameters passed to BSP Room Placement is 1/15 of the graph's width and 1/15 of the graph's height (meaning that more rooms can be placed in the graph due to being able to split more leaves), since the amount of vertices that compose the corridors of the graph when generated by BSP Corridors will not be too large. The resulting dungeon appears similar to the Random Rooms and Random Point Connect dungeon. The efficiency for the BSP Rooms and BSP Corridors pairing is O(1) in general, and

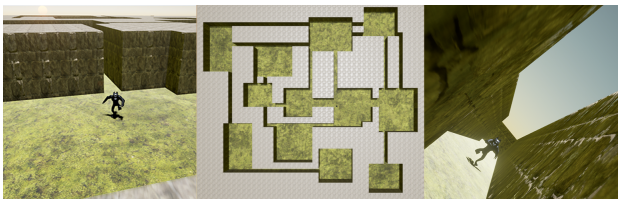

Figure 2: Bird's-Eye and Angled Views. ©Jessica Baron

it is specifically O(ns+c+(w*h)) + O(3c*(W+H)) = O(ns+c+(w*h) + 3c*(W+H)).

3 GAME ADAPTATIONS

The original dungeon generator had simply printed symbols to the output console as means of visualization of the dungeon graphs, as Fig. 1 shows. A basic GUI had been made to run the generator in a more game-like environment, and the dungeon generator has been adapted for usage in UE4.

3.1 Java Application

Using Java's GUI libraries, the developer had created an application allowing a user to run one of the five algorithm pairs via a button click, which produces and displays a procedurally generated graph. The user can control the player by keyboard input to move around the dungeon as three "enemies" meander towards the player's location. Due to the storage of the room and corridor instances, this application can be expanded with more objects to populate the dungeon, such as usable items and non-playable characters.

3.2 Usage in Unreal Engine 4

The dungeon generator code had been translated from Java to C++ for use in Epic's game engine, Unreal Engine 4 [2]. Its purpose is to procedurally generate a new dungeon every time the main game level of the project is loaded. 3D wall mesh actors, "actor" being the UE4 term for anything placeable in the game world [9], are placed in the 3D space of the game level based on the 2D array indices of the generated graph, where the untraversable walls or "off" vertices are located. The UE4 project utilizes the Random Room Placement and Random Point Connect algorithm pair. A DungeonGenerator class creates and manipulates instances of a graph class, which in turn maintains rooms and vertices. This class inherits from the generic UActor of UE4, allowing the generator to behave well with the engine, especially concerning class methods being "Blueprint callable." Blueprints are UE4's visual gameplay scripting system [3]. Upon the start of the project's level, an instance of the DungeonGenerator class is spawned by the Level Blueprint in UE4. Any class method set to "Blueprint callable" in the code can be accessed by the Blueprint in the UE4 Editor. The author used the Level Blueprint to handle the interactions between the individual classes (the main generator, graph, and the wall mesh actor).

The final three figures are screenshots of the game project, in which the player is a metallic werewolf character lost in a procedurally generated dungeon. The center image in Fig. 2 is a bird's-eye-view perspective, similar to viewing the purely 2D dungeon maps, bordered by two angled perspectives, of a dungeon in-game. Fig. 3 better illustrates the dungeon existing in a 3D environment while still having the above camera view.

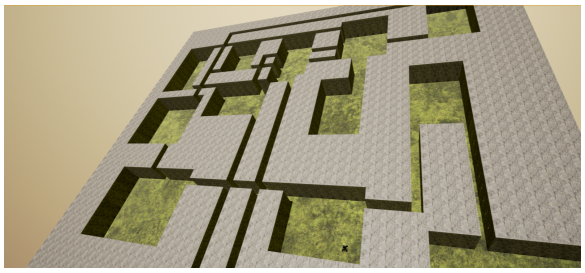

Figure 3: Tilted Bird's-Eye-View of an In-Game Dungeon ©Jessica Baron.

4 CONCLUSIONS AND FUTURE WORK

Procedural content is useful in games, in both computational and aesthetic aspects. The efficiency analysis of each algorithm is in constant time due to the limitations imposed by the algorithm structures on the dimensions of the graph, along with the number and size of rooms generated. Scaling the graph, and therefore the rooms within the graph, to be larger than the dimensions used in this project would not significantly affect this analysis; a constant multiplied by a constant would result in a constant. Each run of any of the algorithms implemented took an average of about 10 milliseconds, a time unnoticeable in each of the visual applications (console output, Java GUI, and UE4 project). Future expansion upon this project entails implementing more variation to the dungeons generated, including researching additional algorithms and applying and randomly selecting from more of them to use in the UE4 project. Likewise, other game elements such as items and enemies to populate the maps next can be procedurally generated.

REFERENCES

[1] Random walk cave generation. http://www.roguebasin.com/index.php?title=Random_Walk_Cave_Generation. Accessed: 2017-01-27.

[2] What is unreal engine 4. https://www.unrealengine.com/what-is-unreal-engine-4. Accessed: 2016-12-19.

[3] Blueprints visual scripting, 2014.

[4] Craveirinha, R., and Roque, L. Designing games with procedural content generation: An authorial approach. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2015), CHI EA '15, ACM, pp. 1199–1204.

[5] Forsyth, W. Globalized random procedural content for dungeon generation. *J. Comput. Sci. Coll. 32*, 2 (Dec. 2016), 192–201.

[6] Hendrikx, M., Meijer, S., Van Der Velden, J., and Iosup, A. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl. 9*, 1 (Feb. 2013), 1:1–1:22.

[7] Hershberger, J., Suri, S., and Toth, C. D. Binary space partitions of orthogonal subdivisions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (New York, NY, USA, 2004), SCG '04, ACM, pp. 230–238.

[8] Sarkar, P. A brief history of cellular automata. *ACM Comput. Surv. 32*, 1 (Mar. 2000), 80–107.

[9] Sherif, W. *Learning C++ by Creating Games with UE4.* Packt Publishing, Birmingham - Mumbai, 2015.

[10] T'oth, C. D. A note on binary plane partitions. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry* (New York, NY, USA, 2001), SCG '01, ACM, pp. 151–156.

[11] Valtchanov, V., and Brown, J. A. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering* (New York, NY, USA, 2012), C3S2E '12, ACM, pp. 27–35.