PROCEDURAL MAP GENERATION FOR GAMES

TNM084, PROCEDURAL IMAGES

Viktor Tholén
vikth187@student.liu.se

Friday 6th May, 2022

Abstract

This report covers the theory behind procedural methods for generating game maps. Several common methods for generating tile-based closed environments are presented and explained. Finally a project in Unreal Engine 4 was implemented inspired by L-Systems primarily. The result was a playable map with multiple floors and passages. The performance was lacking in really large map sizes but small to medium sizes gave a playable result with decent variety.

1 Introduction

Procedural methods are today quite common within the game industry and provide a lot of value for players. Such methods tend to sacrifice control due to their random nature, but can ultimately create very interesting and appealing results. Typically, procedural methods are often used within rogue-like games or dungeon crawlers. However, large scale terrain that can be seen in open-world and various strategy games can also be achieved with procedural approaches, but this report focus on closed space environments such as a building or a dungeon. The main advantage of using procedural maps is the variety it provides for the player, especially for the case of replayablity. This report covers common procedural methods for creating game maps and explores how one can implement a variant of such a method in Unreal Engine 4.

2 Background

Although detailed procedural methods for generating maps are not commonly documented since they are a core part of the game's development, the basic theory can still be found. There are two distinct different categories that each method derive from, tile-based and noise-based methods.

Tile-based methods are based on a grid structure where each coordinate can be occupied by a tile. They are typically used for 2D games but can in theory be useful for 3D grids as well. The first procedurally generated game was called *Rogue* (1980) which featured randomly placed rooms with corridors connecting them all together. This game introduced a new subgenre called *roguelikes* which set the foundation of procedurally generated dungeon crawlers. One seed of the game's map can be seen in Figure 1.

Three different methods for generating tile-based maps in 2D are BSP trees, tunneling algorithms and cellular automata [1]. Each method produce a different layout and can also be combined together to produce interesting results.

2.1 BSP trees

The most basic method is the Binary Space Partitioning (BSP) trees approach. By recursively

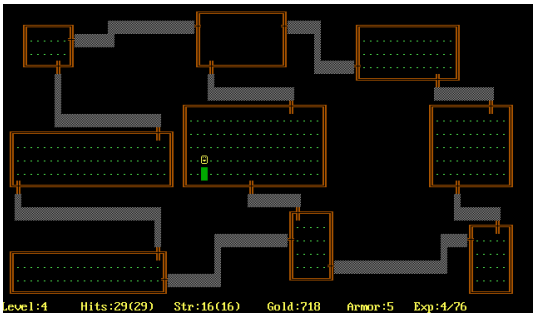

Figure 1: Map from the Rogue game from 1980.

subdividing the area into two, we eventually get a data structure which can be used for generating the map. From the leafs of the tree, the rooms can be grown within the container [2]. Then the corridors can be generated from the centers of the containers with the same parent. The result is the structure seen in Figure 1.

2.2 Tunneling algorithms

Tunneling algorithms can generate similar patterns to BSP trees but use a different technique. Instead of using a tree structure, this method is based on digging out paths from solid terrain, therefore the name. By using multiple tunnelers to dig out paths and spawn new *children* from them with different properties, such as different size, the resulting map can provide a lot of variety and appealing structure. An example of a map generated with a tunneling algorithm can be seen in Figure 2.

2.3 Cellular automata

Cellular automation differ quite a lot from the other two methods mentioned here. This method does not result in the classic room layout, instead it produces a more natural-looking cave system. This can be done by using neighboring tiles to decide whether each tile should be open or closed. The map is generated by picking random points in the grid and, depending on certain adjacency rules, decide the traversable paths of the map. This method can however cause some disconnectivity issues where all paths are not connected to each other, which means that some parts are inaccessible to the player. Thus, this issue have to be resolved after the generation. An example of a map generated with this technique can be seen in Figure 3. In the figure, we can see that this method creates a chaotic pattern that resembles noise, but can still provide a connected path.

2.4 L-Systems

An L-System is a recursive process that is commonly used in computer graphics to simulate organic objects such as plants and trees, but also fractals. They are based on a set of rules that describe how to procedurally generate the desired object. L-Systems commonly uses a string of symbols to easily change the outcome of the generation, where each symbol corresponds to different rules to apply.

3 Method

A procedural generation of a map was implemented inside Unreal Engine 4. This tile-based implementation took inspiration from L-Systems since it is done recursively and formed by a number of rules. Unlike the examples from Section 2, this project aims to create a 3D implementation which means that walls, floors and ceilings need to be considered for the algorithm.

3.1 Traversal

We use one starting position which determine the position of the map. From that position, the tile can traverse in four directions randomly. When traversing, several rules need to be formed in order to not overlap tiles. Overlapping is managed with a data structure that stores the id and its tile, then it is possible to check if a certain grid position contains a tile already. Tiles store references to each actor contained in that tile, such as walls and floors. This way we can access these if we need to delete or adjust them later. Walls are only added if no other tiles are adjacent to

Figure 2: Tunneling algorithm from the Dungeon Maker software. [3]

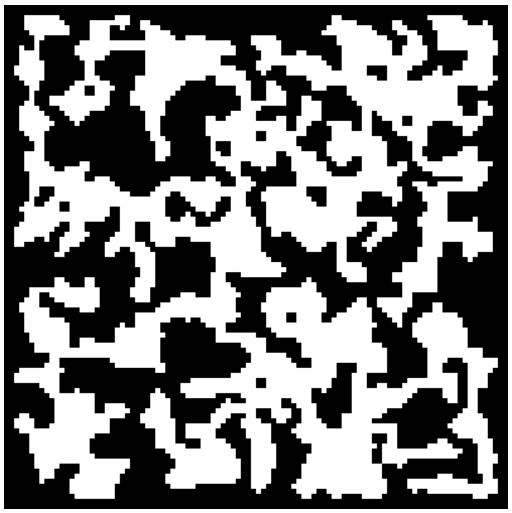

Figure 3: Cellular automata.

the current tile. This avoids overlapped walls which can cause unwanted clipping. Furthermore, the walls are not spawned in the opposite of the traversing direction, which will consequently form corridors. This algorithm spawns a randomly sized patch of tiles which resemble the room layout from BSP trees and handles the overlap accordingly. If the patch size is greater than one, the traversal will be continued in the opposite side of the patch. Figure 4 shows the resulting map from only these few rules affecting walls and floors.

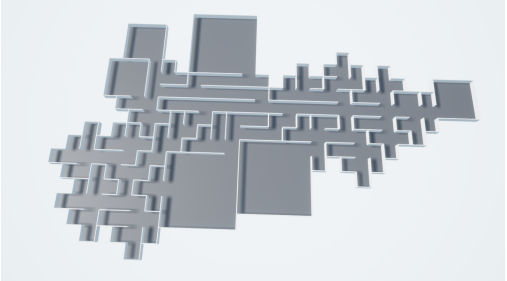

Figure 4: Map generated with only floors and walls.

3.2 Room separation

Separating the the tiles into rooms distinguishes it from looking like a maze. The rooms are separated with doors where Room objects are created. Each room contains a list of the tiles included for that room. This data structure can then be utilized in order to color the rooms' tiles in one color for example. The doors are spawned at random in the opposite direction of traversal.

3.3 Vertical traversal

The grid structure is expanded into three dimensions by adding random stairs to the map. This requires some new rules for checking where a stair can be built, especially to prevent stairs from intersecting paths. The tile after the stairs have to be traversing in the stairs direction which means that two tiles needs to be checked for this. Adding multiple storeys to the map also makes it necessary to add ceil-

Figure 5: Room separation with doors colored in blue. All rooms are colored in random colors to display which tiles belongs to what room.

ings to the each tile. The ceiling can add lights to the map which can become very dark since it is now a closed environment.

3.4 Secret doors

The paths of this algorithm can be quite predictable since they do not interact with other branches. Thus, a solution to this is to add secret doors which provide shortcuts to other rooms that are not directly connected. To do this, the rooms need to store all rooms connected via a door. That data can then be used to add the secret doors by accessing the tiles of the rooms with walls attached, followed by deleting the adjacent wall and replacing it with a door.

4 Results and discussion

The project resulted in a tile-based map generation algorithm that is able to generate interesting game maps. Although the performance was not focused on, it still performs decently inside Unreal Engine 4. The overall performance was tested with a maximum number of secret doors set to 50. The result can be seen in Table 1.

Finally, the generated map is playable in a decent frame-rate across the board. A screenshot from inside the game can be seen in Figure 6. There is a noticeable worse performance

Performance			
Depth	Tiles	Rooms	Time (ms)
10	460	74	323
20	1011	190	693
50	14655	3129	12786
100	47211	9927	56124

Table 1: Performance table.

on higher recursive depths which could possibly be optimized further, but can also be expected as more geometry and lights are present. Another performance issue with this algorithm is that for over 50 recursive depth limits, it becomes apparent that the time it takes to generate is not acceptable for a game. However, maps that are above 50 in depth are very large, containing over 14000 tiles. The most apparent improvement that could be made would arguably be to improve the control of the resulting map. Currently, there is no way to determine where the player should go since a goal does not exist. This could technically be achieved by creating a guided path to the furthest away tile from the starting position, as other methods have done before [4]. Overall, the method used in this project is usable for games and can be considered a viable option for generating these types of maps. There is a lot of potential for what is possible with these types of procedural methods, and this project only scratches the surface.

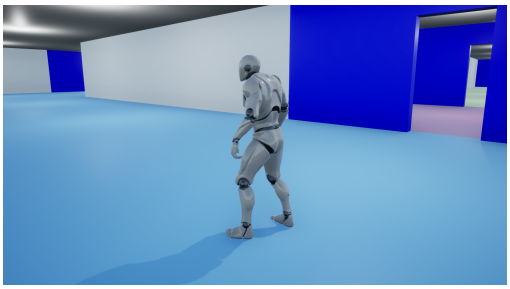

Figure 6: Character playable inside the generated map.

References

[1] Kyzrati. Procedural Map Generation. https://www.gridsagegames.com/blog/2014/06/procedural-map-generation/, 2014. GridSage games development blog, Accessed: 2022-01-14.

[2] Eskerda. Dungeon generation using BSP trees. https://eskerda.com/bsp-dungeon-generation/, 2013. Accessed: 2022-01-14.

[3] Dungeon Maker. Dungeon Maker Manual. http://dungeonmaker.sourceforge.net/DM2_Manual/manual3.html, 2002. Accessed: 2022-01-14.

[4] Roguelike Celebration. Herbert Wolverson - Procedural Map Generation Techniques. https://www.youtube.com/watch?v=TlLIOgWYVpI, 2020. Accessed: 2022-01-14.