Video Game Map Randomization Algorithms

Abstract:

This paper explores randomization algorithms, how they solve the problem of re-playability in video games, how some different types of algorithms work, how they are actually applied in video games and where they can be researched in the future. Within the paper, the algorithms from the games Rogue, DungeonMaker and Tyrant are discussed in detail, along with brief descriptions of Minesweeper, random maze generation and Minecraft.

Introduction:

Re-playability is one of the most important issues that must be dealt with while planning a video game. Many people often will not buy a game unless they believe they will enjoy it for more than one play through. The "Rogue-like" game subgenre of role playing games has addressed this issue very well. Rogue-like games use a randomized map to give the player a unique experience during every play. However, how can instructions be written to make a map that is different every time? It cannot be purely random, there must be some order. That's where map randomization algorithms come in. These algorithms allow for maps to be different, but not so disorganized that the game is unplayable (i.e. there is no way to get to the end because the passageways are assembled randomly).
So how do these algorithms work?

In many cases, the map is drawn as multiple separate rooms that will be randomly connected. One such way is used in the game Rogue, the basis for the genre. The algorithm is summarized below:

1. "Divide the map into a grid (Rogue uses 3x3, but any size will work).
2. Give each grid a flag indicating if it's "connected" or not, and an array of which grid numbers it's connected to.
3. Pick a random room to start with, and mark it "connected".
4. While there are unconnected neighbor rooms, connect to one of them, make that the current room, mark it "connected", and repeat.
5. While there are unconnected rooms, try to connect them to a random connected neighbor (if a room has no connected neighbors yet, just keep cycling, you'll fill out to it eventually).
6. All rooms are now connected at least once.
7. Make 0 or more random connections to taste.
8. Draw the rooms onto the map, and draw a corridor from the center of each room to the center of each connected room, changing wall blocks into corridors. If your rooms fill most or all of the space of the grid, your corridors will very short - just holes in the wall.
9. Scan the map for corridor squares with 2 bordering walls, 1-2 bordering rooms, and 0-1 bordering corridor, and change those to doors.
10. Place your stairs up in the first room you chose, and your stairs down in the last room chosen in step 5. This will almost always be a LONG way away."(Hughes)
This sort of algorithm is fairly straightforward. Rooms are created and connected, and then items are placed (in this case, stairs). Other algorithms place items first, and then create walls around them. The algorithm from the program DungeonMaker is a good example of this.

1. Place special items at random locations throughout the map
2. Create a room of random dimensions (according to parameters set previously) around each item
3. Draw walls to define rooms
4. Use objects (pens, WallCrawler, etc.) which move in random directions to draw other walls that connect and divide rooms (Henningsen)

This algorithm is good for creating more random-looking maps; however it is not necessarily the most efficient. Yet another type of algorithm draws the map from the player's perspective, creating a start area, then drawing room after room until the end area is reached. This is used in the Rogue-like game "Tyrant."

1. "Fill the whole map with solid earth
2. Dig out a single room in the center of the map
3. Pick a wall of any room (For the first loop, there is only one room to choose from)
4. Decide upon a new feature (room, hall, etc.) to build
5. See if there is room to add the new feature through the chosen wall
6. If yes, continue. If no, go back to step 3
7. Add the feature through the chosen wall
8. Go back to step 3, until the dungeon is complete
9. Finally, sprinkle some monsters and items liberally over dungeon" (Zapata).
This algorithm is not necessarily the fastest, however it can be used to create 3d maps (add the floor and ceiling as walls).

How can randomization algorithms be used in games?

The simplest form would be the Rogue-like, discussed above. These games use a 2d map made up of ASCII characters. The randomization algorithm determines where these characters will go in order to create the illusion of walls and floors (Figure 1).

![Figure 1: Screenshot from the game “Rogue”, which uses ANSII characters to represent the environment (Leonard).](image)

While Rogue-like games have the greatest demand for randomization, other games also use map randomization algorithms. One of the simplest of these games is Minesweeper, which uses an algorithm to randomly place the mines. This algorithm is much simpler than those used in Rogue-like games because the mines are placed randomly and the only other features, the numbers that appear telling you how many mines are surrounding each number, are calculated based on the mines (Figure 2).
Another sort of game is a random maze, which can be generated via dozens of algorithms (See Pullen), but still uses the idea of restricted randomness used in Rogue-likes. Finally, a game which takes the algorithm to the next level is Minecraft. This is a far more complicated game currently in beta testing that uses a 3 dimensional map randomization algorithm that simulates real scenery (Figure 4). In order to do this without using too much memory, the game creates smaller, pseudo-randomized "chunks", saves the seeds and reloads them whenever necessary, creating the illusion of an infinite world.

Figure 2: Screenshot from the game “Minesweeper”. The flags are “mines” and the numbers tell you how many mines border the square with the number (Leonard).

Figure 3: Maze generated with the recursive backtracker perfect maze generation algorithm (Pullen).
So how can these algorithms be researched the future?

These algorithms can be used to create anything that appears in an organized yet random order. One possible, yet very complex, application that has not yet been done because of its complexity is the random Sudoku. A Sudoku is a number puzzle played on a 9x9 grid with numbers filled in. The goal is to fill in the empty spaces so that each 3x3 box, row and column has the digits 1-9 in it (Figure 5). To date, there is no computer algorithm for generating a Sudoku because it is difficult to design a puzzle that has exactly one solution. The algorithm for designing a Sudoku is normally carried out by humans.

1. Divide the grid into exposed (the ones already filled in) and empty spaces (the ones the player must fill in) randomly
2. Fill in one empty space randomly

Figure 4: Screenshot from the game “Minecraft”. The algorithm for this game generates massive, realistic, 3 dimensional worlds (Sommer).
3. Fill in 4 exposed spaces with the same number such that they are in every row and column intersecting the box that the first number was in.

4. Repeat for the rest of the board, making sure there are no contradictions.

5. Check for multiple solutions. If there are, make edits so that the extra solutions are invalid (Metzler).

Obviously, the process of backtracking in step 5 is more difficult for a computer to do, considering it involves creating multiple scenarios involving different methods of solving the Sudoku, all of which would use memory.

Figure 5: An unsolved Sudoku (Metzler).
Works Cited


