FROM: http://web.archive.org/web/20080203123815/www.aarg.net/~minam/dungeon_design.html

Random Dungeon Design

The Secret Workings of Jamis Buck's Dungeon Generator

So you've tried my Dungeon Generator once or twice, and it's got you thinking. Perhaps you're a programmer and would like to incorporate similar features in a program of your own. Or maybe you're not a programmer, but would be interested in an overview of how this program works.

Either way, I've been asked how this random dungeon generator works many, many times, and I finally decided that, to save myself time, I'd just put up the description on a web page.

If you find this explanation useful, please let me know. Likewise, if you feel that I was too technical, or not techinical enough, or too ambiguous, let me know that, too, and I can try and improve it.

Please send all comments, questions, suggestions, and flames to:

jgb3@email.byu.edu

I. A Dungeon is a Maze

First of all, it is helpful to think of any dungeon as simply a maze—a collection of corridors that turn every which way. The first part of generating any dungeon, then, is to create a random maze.

Now, there are *lots* of different ways to generate mazes (for some idea of how many different types of mazes and algorithms there are, check out the Maze Algorithms page at Think Labyrinth). For the dungeon generator, I just picked a straightforward algorithm that I'm pretty familiar with—it's a variation on the "Hunt-and-Kill" algorithm. The algorithm creates a *2D*, *normal*, *orthoganol*,*perfect* maze, which simply means that the maze is rectangular, with all passages intersecting at right angles, and that there are no loops or inaccessible areas in the maze.

Here's how the algorithm I picked works. Feel free to substitute this one with any other algorithm.

1. Start with a rectangular grid, *x* units wide and *y* units tall. Mark each cell in the grid *unvisited*.
2. Pick a random cell in the grid and mark it *visited*. This is the *current cell*.
3. From the current cell, pick a random direction (north, south, east, or west). If (1) there is no cell adjacent to the current cell in that direction, or (2) if the adjacent cell in that direction has been visited, then that direction is *invalid*, and you must pick a different random direction. If all directions are invalid, pick a different random *visited* cell in the grid and start this step over again.
4. Let's call the cell in the chosen direction *C*. Create a corridor between the current cell and *C*, and then make *C* the current cell. Mark *C* visited.
5. Repeat steps 3 and 4 until all cells in the grid have been visited.

Once that process finishes, you'll have your maze! There are a few variations you can do to make the maze more interesting; for example, my dungeon generator has a parameter called "randomness". This is a percentage value (0–100) that determines how often the direction of a corridor changes. If the value of *randomness* is 0, the corridors go straight until they run into a wall or another corridor—you wind up with a maze with lots of long, straight halls. If the *randomness* is 100, you get the algorithm given above—corridors that twist and turn unpredictably from cell to cell.

II. Mazes vs. Dungeons

It is important to note that the algorithm given above results in no loops in the maze. It is also important to note that the algorithm results in a *dense* maze—that is, every cell contains a corridor.

This "pure" maze is probably not what you had in mind when you asked for a dungeon. For example, sometimes a dungeon passage intersects with another passage, or with itself, forming a loop. Also, an underground dungeon may cover a lot of territory, but not fill every square meter of rock—it is probably *sparse*, as opposed to *dense*.

There are two steps I used to convert the maze into something more like a dungeon (though still lacking rooms).

The first step involves a parameter I called *sparseness*. It is an integer value; you may want to experiment with it to arrive at a value (or set of values) that work best for you. It is used as follows:

1. Look at every cell in the maze grid. If the given cell contains a corridor that exits the cell in only one direction (in otherwords, if the cell is the end of a dead-end hallway), "erase" that cell by removing the corridor.
2. Repeat step #1 *sparseness* times (ie, if *sparseness* is 5, repeat step #1 five times).

After *sparsifying* the maze, you should wind up with large "blank" gaps, where no passages go. The maze, however, is still *perfect*, meaning that it has no loops, and that no corridor is inaccessible from any other corridor.

The next step is to remove dead-ends by adding loops to the maze. The "deadends removed" parameter of my generator is a percentage value that represents the chance a given dead-end in the maze has of being removed. It is used as follows:

1. Look at every cell in the maze grid. If the given cell is a dead-end cell (meaning that a corridor enters but does not exit the cell), it is a candidate for "dead-end removal."
2. Roll d% (ie, pick a number between 1 and 100, inclusive). If the result is less than or equal to the "deadends removed" parameter, this deadend should be removed. Otherwise, proceed to the next candidate cell.
3. Remove the dead-end by performing step #3 of the maze generation algorithm, above, except that a cell is not considered invalid if it has been visited. Stop when you intersect an existing corridor.

So, now you have something looking more like a dungeon. All it lacks, now, are rooms…

III. Room Generation and Placement

This was perhaps the trickiest step. Looking at my generator, you'll see three parameters: "room count" (R_n), "room width", (R_w), and "room height" (R_h).

Generating rooms is actually easy: R_w is just a random number between the minimum and maximum widths. R_h is generated similarly.

Placing the rooms was trickier. The idea is to find a place in the maze where the given room overlaps a minimum of corridors and other rooms, but where the room touches a corridor in at least on place. To this end, I implemented a *weighting system*.

The program tries to put the room at every possible place in the dungeon. The algorithm works as follows:

1. Set the "best" score to *infinity* (or some arbitrarily huge number).
2. Generate a room such that $W_{min} <= R_w <= W_{max}$ and $H_{min} <= R_h <= H_{max}$.
3. For each cell *C* in the dungeon, do the following:
    a. Put the upper-left corner of the room at *C*. Set the "current" score to 0.
    b. For each cell of the room that is adjacent to a corridor, add 1 to the current score.
    c. For each cell of the room that overlaps a corridor, add 3 to the current score.
    d. For each cell of the room that overlaps a room, add 100 to the current score.
    e. If the current score is less than the best score, set the best score to the current score and note *C* as the best position seen yet.
4. Place the room at the best position (where the best score was found).
5. For every place where the room is adjacent to a corridor or a room, add a door. (If you don't want doors everywhere, add another parameter that determines when a door should be placed, and when an empty doorway [ie, archway, etc.] should be placed).
6. Repeat steps 2 through 6 until all rooms have been placed.

IV. Populating the Dungeon

I won't go into any great detail here, since I took the algorithms for this part straight from the *Dungeon Master's Guide*. The idea is simply to put *something* in each room of the dungeon: hidden treasure, a monster, some description, etc. At this stage, you also determine whether any given door is secret or concealed, and also what the door's properties are (wooden, locked, trapped, etc, etc.). Random tables work quite well to determine all of this.

V. *Finis*

And that, as they say, is the proverbial that. All that remains is to display the dungeon, and that has nothing to do with dungeon generation algorithms. :)

Feel free to download the source code for my dungeon generator—that's where you'll find the *real* technical explanation. The sources are a bastardized mix of C and C++, but fairly readable for all of that. The "jbmaze.cpp" file contains the maze generation algorithms, and the "jbdungeon.cpp" file contains the dungeon generation stuff. The "jbdungeondata.cpp" file populates the dungeon.

The sources can be obtained here:

- Core Code and Web Interface: this file contains the "core" dungeon generator code, as well as the CGI (online) version of the generator. You'll also need the GD and qDecoder libraries (see below) if you plan to actually compile this.
- Windows Interface: this file (.tar.gz, which may cause some grief for windows users) contains the code for the Windows version of the generator. However, it does *not* contain the core code—if you wish to compile the windows version, you *must*download the "Core Code and Web Interface" file, above.
- qDecoder: qDecoder is a CGI library. If you wish to compile the *web* (CGI) version of the generator, you'll need this library.*(The Windows version does* not *need this library.)*
- GD: The GD library is a bunch of graphics routines that are used for creating images. Both the CGI and the Windows versions of the generator use this library to display and/or save the dungeon maps.

Enjoy!