# **Discrete Event Modeling and Simulation**

# **SYSC 5104**

# 

# **(Fall 2017)**

**Assignment 2**

**DUNGEON GENERATION**

**CARLETON UNIVERSITY, OTTAWA**

**PARAMJEET KAUR**

**101086553**

**Dungeon Generation: Conceptual Model**

I propose to use a Cellular-DEVS model to implement a dungeon generation using cellular automata.

I read a thesis paper “Constructive Generation Methods for Dungeons” by “Macro Niemann”. I was intrigued by his/her reliable and efficient approach to procedurally generate a system of dungeons in cellular automata. The very basic element the Cellular Automata will operate on, is a 2D grid of cells with each of the cells being in a finite amount of states(something like{1,0}).

Another characteristic of this method is the concept of so called neighborhoods, which refers to

the cells surrounding a chosen cell. Two common patterns are the Moore and the von Neumann neighborhood:

• **the Moore neighborhood** (Figure 1a) spans all cells (marked in light brown) surrounding a selected cell (marked in blue) - it can even have multiple levels (second level indicated in dark brown )

• **the von Neumann neighborhood** (Figure 1b) only includes the cells in the north, south, west

and east of the selected cell.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | c |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | c |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

a) Moore Neighborhood b) von Neumann Neighborhood

**Figure 1:** Grid and Neighborhoods of Cellular Automata

In dungeon generation process, I made use of Moore Neighborhood of level 1.

As Cellular Automata are time-discrete systems, they change their state in time-steps and not

continuously (so a sequence like *t*0 *! t*1 *! ::: ! tn* exists). The type of the state change is determined by a fixed set of rules and the given neighborhood and usually all cells change their state after a time step. So , there could be the simple rule in a dungeon level, that whenever there are three cells of wall in the neighborhood of a cell, it will be transformed to a wall as well (or to a floor if there is less than three wall cells and the cell is a wall). The concept is simple: a dungeon is defined as an array of cells, with 30 rows and 30 columns and two states which are a wall cell or a floor/empty cell. State“1” represents that the cell is a wall, and state“0” represents that cell is a floor cell (a free cell).

There are two states of cell that are wall and the floor. The entire process of dungeon generation took place as described below:

1 initialize empty grid with a x b cells // e.g. 30 x 30

2 initialize floor/empty cells

3 convert randomly with probability r

4 for n iterations

5 go through each cell

6 calculate neighborhood values

7 if past threshold T

8 convert

9 create adjacent grids // for square cells that will be 4 or 8

10 for each repeat steps 1-8

11 if two adjacent grids are not connected

12 create a connection // between two accessible areas

13 run n additional Cellular Automata iterations // removing inconsistencies

14 create wall cells // special rock cells

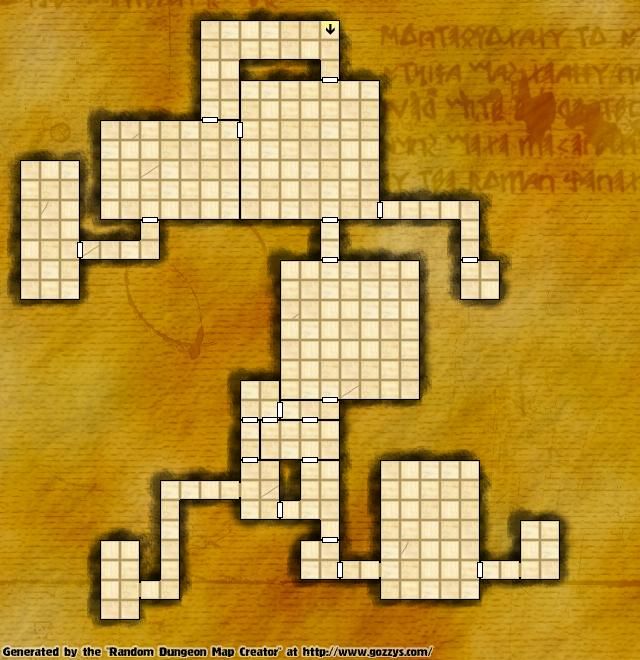


Fig 1. depicts some random dungeon which can be generated using the above algorithm.

To start of an initial map is laid down, for this we start with a rectangular map of cell with a non-diggable/permanent border representing the wall cells. Then randomly empty cells are placed in the map. So, a clean map is generated to start off with. The second step includes the Cellular Automata rules which are run for dungeon generation. The basic rule used for the dungeon generation is the 4-5 rule which states:

1. For any given square, if it has 3 or fewer adjacent wall squares (counting all 8 cardinal compass points), the square 'starves' and becomes a floor.
2. If it has greater than 5 adjacent wall squares, the square becomes a wall.
3. Otherwise, leave it as is.

Cellular Automata systems usually run through many cycles and interesting structures can emerge from this. For dungeon generation purposes here, though, one cycle was enough to generate an interesting set of dungeons. Other Cellular Automata rules, of course, may produce different dungeon patterns. The first two steps of the dungeon generation algorithm yield a nice-looking dungeon with a few distinct scattered dungeons. The third step involves joining these dungeons into a big, fully connected dungeon. So, for this, think of each dungeon being a set of floor squares. There are 6-7 disjoint sets one map after step 2. What was needed to do is draw lines of floor squares between the dungeons. For each square on that line, he union-end it with one of the sets, stopping when the two sets are union-end into one (the dungeons will be joined. At the end he got a nice, clean result showing a 30x30 dungeons which took 5 seconds to generate.

**Dungeon Generation: Formal Specification**

**A. Cell-DEVS Atomic Model Specification**

The following is the formal specification for the Cell-DEVS dungeon-generation model:

CD = < X, Y, I, S, θ, N, d, δint, δext, τ, λ, D >

X = Ø

Y = Ø

S = { 0, 1 }

N = neighborhood = {(-1,1),(0,1),(1,1),(0,-1),(0,0),(0,1),(-1,-1),

(0,-1)(1,-1)} // Moore Neighborhood of level 1

d = 100 ms // Inertial delay

τ: N🡪S is defined by the rules described in the previous section, i.e.:

S = 1 if cell(0,0) = 0 and truecount >5

S = 0 if cell(0,0) = 1 and truecount <4

S= otherwise as it is .

**B. Cell-DEVS Coupled Model Specification**

The model specification from part (A) above gives rise to the following model definition:

[top]

components: dungeon\_generation

[dungeon\_generation]

type : cell

width : 30

height : 30

delay : inertial

defaultDelayTime : 100

border : wrapped

neighbors : dungeon\_generation(-1,1) dungeon\_generation(1,0) dungeon\_generation(1,1)

neighbors : dungeon\_generation(0,-1) dungeon\_generation(0,0) dungeon\_generation(0,1)

neighbors : dungeon\_generation(-1,-1) dungeon\_generation(-1,0) dungeon\_generation(1,-1)

initialvalue : 0

localtransition : dungeon\_generation-rule

[dungeon\_generation-rule]

rule : 1 100 { (0,0) = 0 and trueCount > 5 }

rule : 0 100 { (0,0) = 1 and trueCount < 4 }

rule : {(0,0)} 100 { t }

The model definition also defines the initial random dungeon by specifying the states of the cells as 1’s or 0’s.Initially,the following initial values are assigned to each cell so as to show that 42% of the area is covered by wall. For example, the following is a definition of the random dungeon which can be solved using the Cell-DEVS model:

initialrowvalue : 0 111111111111111111111111111111

initialrowvalue : 1 100000001110111010110110101001

initialrowvalue : 2 111101101001000010111111000001

initialrowvalue : 3 110110111001101110010010001001

initialrowvalue : 4 111100010001101100110110011111

initialrowvalue : 5 110111101110101111000011010111

initialrowvalue : 6 100111110110011011100101000101

initialrowvalue : 7 111110010110010011110101110101

initialrowvalue : 8 101111100101100011101001110011

initialrowvalue : 9 110100111100111110011001000111

initialrowvalue : 10 100111100001011001001011101001

initialrowvalue : 11 100000100111011001010000100111

initialrowvalue : 12 101000001011011110010001001001

initialrowvalue : 13 100000100001010001011101110111

initialrowvalue : 14 111110111001111100001101100001

initialrowvalue : 15 100111111100101011010100010001

initialrowvalue : 16 111101000011010101100101110001

initialrowvalue : 17 101101111010001100101011101101

initialrowvalue : 18 110101010001101000001100110011

initialrowvalue : 19 110101010011010110011110011011

initialrowvalue : 20 111011110010001001101100110101

initialrowvalue : 21 110110010010011111000010101111

initialrowvalue : 22 110111001001010011001110100011

initialrowvalue : 23 100000110000110111011010001111

initialrowvalue : 24 110001100001000100011110101001

initialrowvalue : 25 111101010110001101101100110001

initialrowvalue : 26 110110010010010010111011001011

initialrowvalue : 27 111111011100111000000100111111

initialrowvalue : 28 111110001101011101110010011101

initialrowvalue : 29 111111111111111111111111111111

**C. Implementation and Testing**

The model definitions given above was implemented in one test case with two separate steps, in which initially, a random map was created with 42% of the map being flat/floor. Each step was implemented as a separate model file (called dungeon1.ma, dungeon2.ma).

**Step** 1**: A random dungeon generation.**

The first step was to generate a random dungeon from 30x30 map of randomly arranged walls and floors. The 30x30 map definition is shown in part (B) above, in the “initialrowvalue” lines of the .ma file. When the model was executed, the rules changed the wall cells with 3 or lesser walls in its neighbor, to a path and a path cell with more than 5 wall cells as its neighbors to a wall. This set of rules changed the map to a 30x30 structure with random distinct dungeons. Therefore, a **step 2** is required with different cellular automata rules and the result of step 1 as the initial values of the cell spaces, to generate a fully connected dungeon structure. The following diagram illustrates the initial cell state and the random distinct dungeons, as visualized in the Web viewer tool:

|  |  |
| --- | --- |
|  |  |

**Figure 2:** State change in Cellular Automata with a Moore neighborhood of level 1

**Step 2: Connecting the random distinct dungeons.**

Next, the random distinct dungeons were joined to generate a 30x30 fully connected dungeon. The model attributes like size and timing were not changed, but new cellular automata rules were used with the initial values of the cell space same as the result of the 1st step. Those are as follow:

[top]

components : dungeon\_generation

[dungeon\_generation]

type : cell

width : 30

height : 30

delay : inertial

defaultDelayTime : 100

border : wrapped

neighbors : dungeon\_generation(-1,1) dungeon\_generation(0,1) dungeon\_generation(1,1)

neighbors : dungeon\_generation(0,-1) dungeon\_generation(0,0) dungeon\_generation(0,1)

neighbors : dungeon\_generation(-1,-1) dungeon\_generation(0,-1) dungeon\_generation(1,-1)

initialvalue : 0

localtransition : dungeon\_generation-rule

[dungeon\_generation-rule]

rule : 0 100 { (0,0) = 1 and trueCount < 4 }

rule : 0 100 { (0,1) = 1 and trueCount = 3 }

rule : 1 100 { (0,0) = 1 and trueCount < 3 }

rule : 0 100 { (0,1) = 1 and trueCount < 4 }

rule : {(0,0)} 100 { t }

The initial values of the cell space same as the result of the 1st step. Those are used as follows:

initialrowvalue : 0 111111111111111111111111111111

initialrowvalue : 1 111111111111111111111111101001

initialrowvalue : 2 111111111001111111111111000001

initialrowvalue : 3 111111111001111110011110001001

initialrowvalue : 4 111111111001111110000110011111

initialrowvalue : 5 111111111110111111000011001111

initialrowvalue : 6 111111111110011111100011100111

initialrowvalue : 7 111111111110010011110001110011

initialrowvalue : 8 111111111101100011111001110011

initialrowvalue : 9 110111111000111110011001100111

initialrowvalue : 10 100011100001111000000001100011

initialrowvalue : 11 100000000011111000000000100111

initialrowvalue : 12 100000000011111100000001000011

initialrowvalue : 13 110000000001111100001101100011

initialrowvalue : 14 111111111001111100001101100001

initialrowvalue : 15 111111111000111110000100110001

initialrowvalue : 16 111111110000011100000101110001

initialrowvalue : 17 111111110000001100001011111001

initialrowvalue : 18 111111110000001000001100111111

initialrowvalue : 19 111111110000000100011110011111

initialrowvalue : 20 111111110000001111101100011111

initialrowvalue : 21 111111100000011111000110001111

initialrowvalue : 22 110111000000011111001110000111

initialrowvalue : 23 100000100000000111111110000111

initialrowvalue : 24 110001100000000111111110000001

initialrowvalue : 25 111101000000001101111111110001

initialrowvalue : 26 111110011000010000111111111011

initialrowvalue : 27 111111111100111000111111111111

initialrowvalue : 28 111111111111111101111111111111

initialrowvalue : 29 111111111111111111111111111111

When this model was executed, the result was that all the distinct dungeons were joined to each other and a fully connected 30x30 fully connected dungeon was generated in few seconds. The following diagram illustrates the initial cell state and the final state, as visualized in the Webviewer tool.

|  |  |
| --- | --- |
|  |  |

**Figure 3:** State change in Cellular Automata with a Moore neighborhood of level 1

**Dungeon Generation: Conclusions**

The Cellular-DEVS models included with this report and described here in correctly simulate the behavior of the dungeon system generation algorithm by Macro Niemann in his thesis paper. The dungeons, modeled as cell spaces, can be generated by destroying the old or creating new floor and wall cells, in order to fill bigger spaces with floor cell. This algorithm has an advantage over other dungeon generating algorithms as it can fully connected dungeon system instead of just leaving it with random distinct dungeons.

The downside to this algorithm is that it sometimes doesn’t join all the distinct dungeons, but can be made possible by increasing the no. of runs. Even it won’t work as the same for the dungeon in squares bigger than 50x50.

The author of this paper mentioned several approaches to generate dungeons. He explained in his research paper that the use of Cellular Automaton can be useful to generate more ’natural’/organic looking caverns. In terms of *diversity* and *believability* this is an advantage of using Cellular Automata over other dungeon generation approaches. Cellular Automata has an ability to generate infinite dungeon levels in very low average generation time approx. in milliseconds. The logic behind the algorithm is simple but a bit tricky. It would be interesting, in further studies, to determine if this algorithm would scale to 3-dimensional spaces.

This assignment was also successful in demonstrating the use of the CD++ tool in simulating a cellular automaton model, and using the associated tools (CD++ Builder, Webviewer) to visualize the outputs.