Chapter 3
Constructive generation methods for dungeons and levels (DRAFT)

Noor Shaker, Antonios Liapis and Julian Togelius

3.1 Dungeons and levels

A dungeon, in real life, is a cold, dark and dreadful place where prisoners are kept. A dungeon, in a computer game, is a labyrinthine environment where adventurers enter at one point, collect treasures, evade or slay monsters, rescue noblepeople, fall into traps and ultimately perhaps exit at another point. This conception of dungeons probably originated with the tabletop roleplaying game *Dungeons and Dragons*, and has been a key feature of almost every computer role playing game (RPG), including genre-defining games such as the *Legend of Zelda* series and the *Final Fantasy* series and recent megahits such as *The Elder Scrolls V: Skyrim*. Of particular note is the roguelike genre of games which, following the original *Rogue* from 1980, feature procedural runtime dungeon generation; the *Diablo* series are the most recent high-profile games in this tradition.

We can see a dungeon as an two-dimensional structure, consisting of a number of cells. The cells could be free (empty), or have different types of content, such as wall, entrance, exit, dragon, diamond, amoeba, banana etc., depending on the particularities of the game in which it is used. Similarly, what constitutes a good and enjoyable dungeon (as opposed to a boring, trivial or unplayable one) depends on the particularities of the game. To solve the content generation problem of creating good dungeons therefore requires that we know some aspects of the game, or make some assumptions.

In the exercise at the end of the previous chapter, you constructed a simple search-based dungeon generator. You made a number of key assumptions, such that the character can move up, down, left and right, can move through free space but not (normally) through walls, and that it was a key demand that the character can move from the entrance to the exit (the exit is reachable from the entrance). Additional assumptions could of course be made, such that the game becomes boring if the player has to backtrack too often, or never at all, which could translate to limits on

the number of paths that are dead ends, and how much paths branch in the dungeon in general.

The dungeon as a game artefact is related to another classic two-dimensional structure: the platform game level. We are here talking about levels for such classic games as *Super Mario Bros* and *Sonic the Hedgehog* and their countless clones and near-clones, as well as games that have drawn inspiration from them; a modern-day example of a game in this tradition that features procedural generation is *Spelunky*, discussed in the first chapter. Like dungeons, platform game levels typically feature free space, walls, treasures or other collectables, enemies and traps. However, the game mechanics differ from those of a roguelike or top-down RPG in that the player agent is typically constrained by gravity: the agent can move left or right and fall down, but can typically only jump a small distance upwards. This leads to the interplay of platforms and gaps being an important addition to the vocabulary of platform game levels.

In this chapter, we will study a number of methods for creating two-dimensional structures such as dungeons and platform game levels. These methods do not have any common methodological foundation. What they do have in common is that they are constructive, so that they only produce one artefact per run and only produce it once (as opposed to e.g. search-based methods). They also have in common that they are fast, offer limited control over the outcome of the process, and have been successfully used to create levels at runtime. All of these techniques will generalise to various types of content, but for simplicity each algorithm is presented in the context of a single content type, either dungeons or platform game levels.

The first family of algorithms to be discussed in this chapter are those based on space partitioning. This is a simple and very general principle that have been used in such diverse areas as 3D graphics and compression, but it is also very useful in PCG. Two different examples of how dungeons can be generated by space partitioning are given; the core idea is to recursively divide the dungeon in pieces and then connect them. This is followed by a discussion on agent-based methods for generating dungeons, with the core idea that agents dig paths into a primeval mass of walls. The next big family of algorithms to be introduced is cellular automata; these computational models, widely used in physics and complexity theory, turn out to be a simple and fast means of generating certain kinds of structures such as cave-like dungeons. This is followed be a discussion of agent-based level generation methods as they have been implemented in two platform games, the commercial game Spelunky and the open-source framework Infinite Mario Bros, and its recent offshoot InfiniTux. The lab exercise will have you implementing at least one method from the chapter using the InfiTux API.

3.2 Space Partitioning Algorithms

True to its name, a space partitioning algorithm divides a two-dimensional or three-dimensional space into two or more disjoint subsets (*space partitions*) so that any

point in the space can lie in exactly one of these subsets. Space partitioning systems are usually hierarchical: each space partition is further subdivided by applying the same algorithm recursively. This allows partitions to be arranged in a tree identified as the *space partitioning tree*. This neatly arranged data structure allows for fast geometric queries regarding any point within the space; this makes space partitioning trees particularly important for computer graphics as it enables efficient raycasting, frustum culling and collision detection.

The most popular method for space partitioning is *binary space partitioning*, which recursively divides a space into two partitions. Through binary space partitioning, the space can be represented as a binary tree (specifically identified as a *BSP tree*). Binary space partitioning is most often used to render polygons via the *painter's algorithm*; variations of this algorithm have been used in computer games dating as far back as *DOOM* (id Software, 1993). The painter's algorithm constructs a BSP tree from an unsorted list of all polygons in a scene, by assuming that each polygon has a 'front' and a 'back' face. Starting from a polygon (potentially chosen at random), the BSP algorithm arranges the remaining polygons of the scene (dividing polygons where necessary) into those positioned 'in front' or 'behind' this first polygon. The process is applied recursively on polygons or partitions of polygons on either side of the tree until all polygons have been arranged (see Fig. 3.1).

Unlike the painter's algorithm, other variants of BSP do not use existing polygons to divide the space; instead, they create their own splitting hyperplanes based on some implementation rules. Such algorithms include quadtrees and octrees: a quadtree partitions a two-dimensional space into four quadrants and an octtree partitions a three-dimensional space into eight octants. We will be using quadtrees on two-dimensional images as the simplest example, although the same principles apply for octtrees and for other types of stored data. While a quadtree's quadrants can have any rectangular shape, they are usually equal-sized squares. A quadtree with a depth of n can represent any binary image of 2^n by 2^n pixels, although the number of the tree's nodes (and its depth) depend on the structure of the image. The root node represents the entire image, and its four children represent the top left, top right, bottom left and bottom right quadrants of the image. If the pixels in any quadrant are not the same value (either 0 or 1), that quadrant is subdivided; the process is applied recursively until each leaf quadrant (regardless of size) contains pixels of the same value (see Fig. 3.2).

3.2.1 Space Partitioning for Dungeon Generation

As space partitioning algorithms are traditionally used in 2D or 3D graphics, their purpose is to store existing elements such as polygons or pixels rather than create new ones. However, the principle that space partitioning results in disjoint subsets with no overlapping areas is particularly desirable for creating rooms in a dungeon or distinct areas in a game. Dungeon generation via BSP follows a *macro* approach, where the algorithm acts as an all-seeing dungeon architect rather than a 'blind'

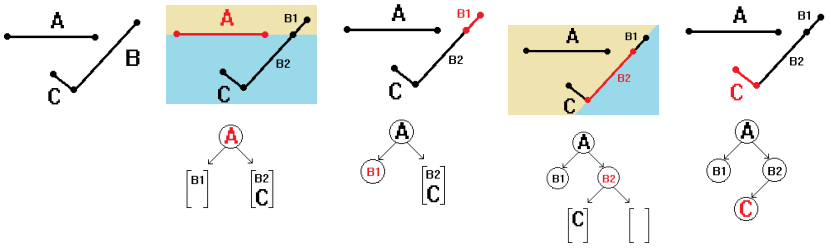
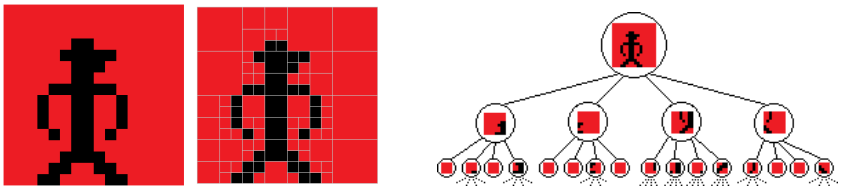


Fig. 3.1: A simple example of binary space partitioning using the painter's algorithm (Fig. 3.1a). We choose to begin with line A as our root node (Fig. 3.1b). The space between the front of A (yellow) and the back of A (cyan): this space division requires line B to be split into B1 and B2. At this point, the BSP tree contains a list of lines on the front of A on the left side of the tree (B1) and a list of lines on the back of A on the right side (B2, C). Moving to the left side of the BSP tree (Fig. 3.1c), since line B1 is the only member of this list it is added as a child node of A. Moving to the right side of the tree (Fig. 3.1d), we begin from line B2 and add it as a child node of A; then the tree is split between the front of B2 and the back of B2. Since C is in the front of B2, the list of lines on the front of B2 on the left side of the tree contains C and the list of lines on the back of B2 is empty. Moving to C (Fig. 3.1e), since it is the only member of this list it is added as a child node of B2, and the BSP tree is complete. It should be noted that if we started with B as our root node, we would not need to split any lines and thus our tree would be smaller; for more complex problems, choosing the right root node can have a large impact on the size and depth of the BSP tree.

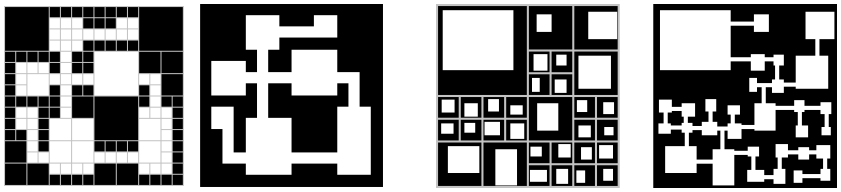
digger as is often the case with agent-based approaches presented in Section 3.2.2. The entire dungeon area is stored in the root node of the BSP tree and is partitioned recursively until a terminating condition is met (such as a minimum or maximum size for rooms). The BSP approach secures that no two rooms will be overlapping and allows for a very 'structured' appearance of the dungeon.

How closely the generative algorithms follow the principles of "traditional" partitioning algorithms affects the appearance of the dungeon created. For instance, a dungeon can be created from a quadtree by selecting quadrants at random and splitting them; once complete, each quadrant can be assigned a value of 0 (empty) or 1 (room), taking care that all rooms are connected. This creates very symmetric, 'square' dungeons such as those seen in Fig. 3.3a. The principle that a leaf quadrant must be composed of a single element (or the same pixel value in the case of images) can be relaxed for the purposes of dungeon generation; if each leaf quadrant contains a single room but can also have empty areas allows rooms of different sizes, as long as their dimensions are smaller than the quadrant's bounds. These rooms can then be connected with each other, using random or rule-based processes, without



(a) (b)

Fig. 3.2: An example quadtree partition of a binary image (0 shown as red, 1 as black). Large areas of a single color, such as those on the right edge of the image, are not further partitioned. The image is 16 by 16 pixels, so the quadtree has a depth of 4. While a fully expanded quadtree (with leaf nodes containing information about a single pixel) would have 256 leaf nodes, the large areas of a single color result in a quadtree with 94 leaf nodes. Fig. 3.2b shows the first layers of the tree: the root node contains the entire image, with the four children ordered as: top left quadrant, top right quadrant, bottom left quadrant, bottom right quadrant (although other orderings are possible).



(a) (b)

Fig. 3.3: Fig 3.3a shows a dungeon created using a quadtree, with each partition consisting entirely of empty space (black) or rooms (white). Fig 3.3b shows a dungeon created using a quadtree, but with each partition containing a single room (placed stochastically) as well as empty space; corridors are added after the partitioning process is complete.

taking the quadtree into account at all. Even with this added stochasticity, dungeons are still likely to be very neatly ordered (see Fig. 3.3b).

The following paragraph will describe an even more stochastic approach based loosely on binary space partitioning techniques.

We consider an area for our dungeon, of width w and height h, stored in the root node of a BSP tree. Space can be partitioned along vertical or horizontal lines, and the resulting partitions do not need to be of equal size. While generating the tree, in every iteration a leaf node is chosen at random and split along a randomly chosen vertical or horizontal line. A leaf node is not split any further if it is below a minimum size (we will consider a minimal width of $w/4$ and minimal height of

$h/4$ for this example). Each partition contains a single room; the corners of each room are chosen stochastically so that the room lies within the partition and has an acceptable size (i.e. is not too small). Once the tree is generated, corridors are generated by connecting children of the same parent with each other. Below is the high-level pseudo-code of the generative algorithm, and Fig. 3.4 and 3.5 shows the process of generating a sample dungeon.

```
1: start with the entire dungeon area (root node of the BSP tree)
2: divide the area along a horizontal or vertical line
3: select one of the two new partitions
4: if this partition is above than the minimal acceptable size:
5:   go to step 2 (using this partition as the area to be divided)
6: select the other partition, and go to step 4
7: for every partition:
8:   create a room within the partition by randomly
     choosing two points (top left and bottom right)
     within the partition's boundaries
9: starting from the lowest layers, draw corridors to connect
   rooms in the nodes of the BSP tree with children of the same
   parent
10:repeat 9 until the children of the root node are connected
```

While binary space partitioning is primarily used to create non-overlapping rooms, it should be noted that the hierarchy of the BSP tree can be used for other aspects of dungeon generation as well. The previous example of Fig. 3.5 has demonstrated how room connectivity can be determined by the BSP tree: using corridors to connect rooms belonging to the same parent minimizes the chances of overlapping or intersecting corridors. Moreover, the higher-level partitions can be used to create groups of rooms following the same theme; for instance, a section of the dungeon may contain higher-level monsters, or monsters that are more vulnerable to magic. Coupled with corridor connectivity based on BSP tree hierarchy, these groups of rooms may have a single entrance from the rest of the dungeon; this allows such a partition to be decorated as a prison or as an area with dimmer light, favoring players who excel at stealthy gameplay. Some examples of themed dungeon partitions are shown in Fig. 3.6.

3.2.2 Agent-based dungeon growing

Agent-based approaches for dungeon generation usually amount to using a single agent to dig tunnels and create rooms in a sequence. Contrary to the space partitioning approaches of Section 3.2.1, an agent-based approach such as this follows a *micro* approach and is more likely to create an organic and perhaps chaotic dungeon instead of the neatly organized dungeons of Section 3.2.1. The appearance of the dungeon largely depends on the behaviour of the agent: an agent with a high degree of stochasticity will result in very chaotic dungeons while an agent with some "look-ahead" may avoid intersecting corridors or rooms. It should be noted that the

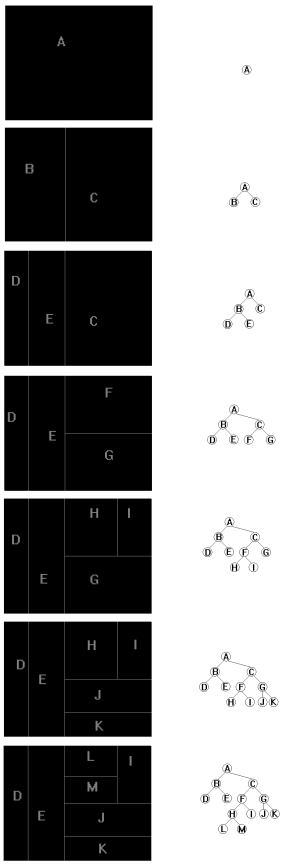


Fig. 3.4: Stochastically partitioning the dungeon area A, which is contained in the root node of the BSP tree. Initially the space is split into B and C via a vertical line (its x-coordinate is determined randomly). The smaller area B is split further with a vertical line into D and E; both D and E are too small to be split (in terms of width) so they remain leaf nodes. The larger area C is split along a horizontal line into F and G, and areas F and G (which have sufficient size to be split) are split along a vertical and a horizontal line respectively. At this point, the partitions of G (J and K) are too small to be split further, and so is partition I of F. Partition H is still large enough to be split, and is split along a horizontal line into L and M. At this point all partitions are too small to be split further and dungeon partitioning is terminated with 7 leaf nodes on the BSP tree. Fig. 3.5 demonstrates room and corridor placement in this dungeon.

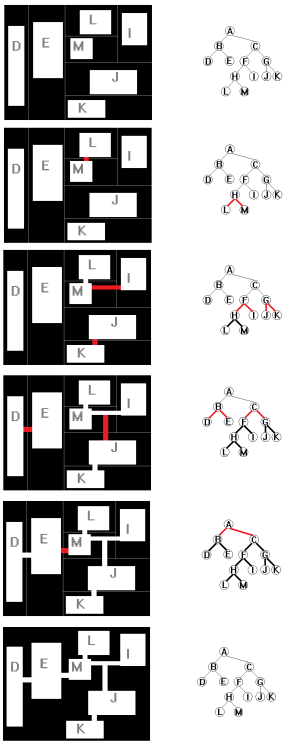


Fig. 3.5: Room and corridor placement in the partitioned dungeon of Fig. 3.4. For each leaf node in the BSP tree, a room is placed by randomly choosing coordinates for top left and bottom right corners, within the boundaries of the partition (1st image). Then a corridor is added to connect the leaf nodes of the lowest layer of the tree (L and M); for all intents and purposes, the algorithm will now consider rooms L and M as joined, grouping them together as their parent H. Moving up on the tree, H (the grouping of rooms L and M) is joined via a corridor with room I and rooms J and K ar joined via a corridor into their parent G. Further up, rooms D and E of the same parent are joined together via a corridor, and the grouping of rooms L, M and I are joined with the grouping of rooms J and K. Finally, the two subtrees of the root node are joined together and the dungeon is fully connected.

impact of the AI behaviour's parameters on the generated dungeons' appearance is difficult to guess without extensive trial-and-error; as such, agent-based approaches are much more unpredictable than space partitioning methods. Moreover, there is no guarantee that an agent-based approach will create a dungeon without rooms overlapping with each other or a dungeon which spans only a corner of the dungeon area rather than its entirety. The following paragraphs will demonstrate two agent-based approaches for generating dungeons.

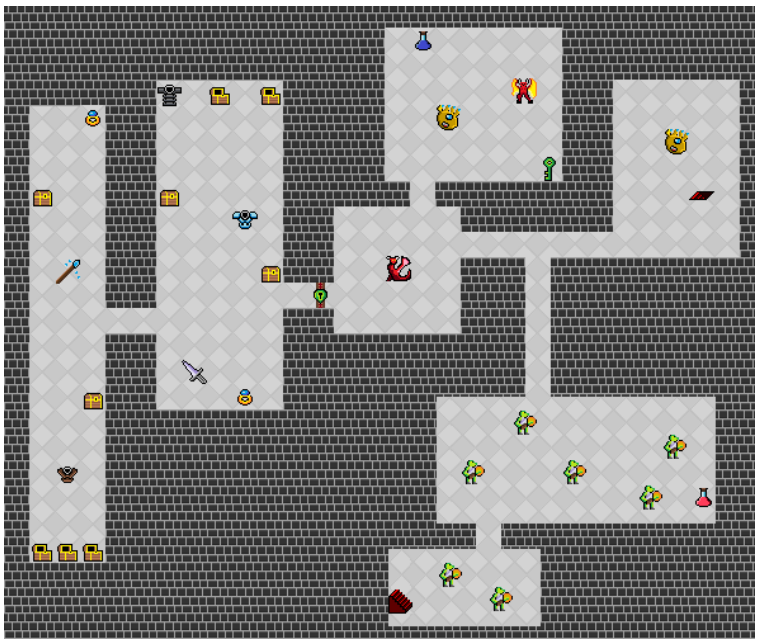


Fig. 3.6: The example dungeon from Fig. 3.5, using the partitions to 'theme' the room contents. Partitions B and C are only connected by a single corridor; this allows the rooms of partition B to be locked away (green lock), requiring a key from partition C in order to be accessed (room L). Similarly, rooms of partition B contain only treasures and rewards, while rooms of partition C contain predominantly monsters. Moreover, the challenge rating of monsters in partition C is split between its child nodes: partition G contains weak goblins while partition F contains challenging monsters with magical powers. Further enhancements could increase the challenge of partition G by making it darker (placing fewer light sources), using different textures for the floor and walls of partition B, or changing the shape of rooms in partition C to circular.

There is an infinite amount of AI behaviours for digger agents when creating dungeons, and they can result in vastly different results. As an example, we will first consider a highly stochastic, 'blind' method. The agent is considered to start at some point of the dungeon, and a random direction is chosen (up, down, left or right). The agent starts digging in that direction, and every dungeon tile dug is replaced with a 'corridor' tile. After making the first 'dig', there is a 5% chance that the agent will change direction (choosing a new, random direction) and another 5% chance that the agent will place a room of random size (in this example, between 3 and 7 tiles wide and long). For every tile that the agent moves in the same direction as the previous one, the chance of changing direction increases by 5%. For every tile that

Fig. 3.7: A short run of the stochastic, 'blind' digger. The digger starts at a random tile on the map (1st image), and starts digging downwards. After digging 5 tiles (3rd image), the chance of adding a room is 25%, and it is rolled, resulting in the 4th image. The agent continues moving downwards (4th image) with the chance of adding a room at 5% and the chance of changing direction at 30%: it is rolled, and the new direction is right (6th image). After moving another 5 tiles (7th image), the chance of adding a room is at 30% and the chance of changing direction is at 25%. A change of direction is rolled, and the agent starts moving left (8th image). After another tile dug (9th image), the chance of adding a room is 40% and it is rolled, causing a new room to be added (10th image). Already from this very short run, the agent has created a dead-end corridor and two overlapping rooms.

the agent moves without a room being added, the chance of adding a room increases by 5%. When the agent changes direction, the chance of changing direction again is reduced to 0%. When the agent adds a room, the chance of adding a room again is reduced to 0%. Fig. 3.7 shows an example run of the algorithm, and below is the pseudo-code for running this algorithm.

```
1: initialize chance of changing direction Pc=5
2: initialize chance of adding room Pr=5
3: place the digger at a dungeon tile and randomize its direction
4: dig along that direction
5: roll a random number Nc between 0 and 100
6: if Nc below Pc:
7:   randomize the agent's direction
8:   set Pc=0
9: else:
10:  set Pc=Pc+5
11:roll a random number Nr between 0 and 100
12:if Nr below Pr:
13:  randomize room width and room height between 3 and 7
14:  place room around current agent position
14:  set Pr=0
15:else:
16:  set Pr=Pr+5
17:if the dungeon is not large enough:
18:  go to step 4
```

In order to avoid the lack of control of the previous stochastic approach, which can result in overlapping rooms and dead-end corridors, the agent can be a bit more informed about the overall appearance of the dungeon and 'look ahead' whether the addition of a room would result in room-room or room-corridor intersections. Moreover, the change of direction does not need to be rolled in every step, to avoid winding pathways.

Fig. 3.8: A short run of the informed, "look ahead" digger. The digger starts at a random tile on the map (1st image), and places a room (2nd image) and a corridor (3rd image) since there can't be any overlaps in the empty dungeon. After placing the first corridor, there is no space for a room (provided rooms must be at least 3 by 3 tiles) which doesn't overlap with the previous room, so the digger makes another corridor going down (4th image). At this point, there is space for a small room which doesn't overlap (5th image) and the digger carries on placing corridors (6th image and 8th image) and rooms (7th image and 9th image) in succession. After the 9th image, the digger can't add a room or a corridor that doesn't intersect with existing rooms and corridors, so generation is halted despite a large part of the dungeon area being empty.

We will consider a less stochastic agent with 'look ahead' as a second example. Like above, the agent starts at a random point in the dungeon. The agent checks whether adding a room in the current position will cause it to intersect existing rooms. If all possible rooms result in intersections, the agent picks a direction and a digging distance that will not result in the potential corridor intersecting with existing rooms or corridors. The algorithm stops if the agent stops at a location where no room and no corridor can be added without causing intersections. Fig. 3.8 shows an example run of the algorithm, and below is the pseudo-code for running this algorithm.

```
1: place the digger at a dungeon tile
2: set helper variables Fr=0 and Fc=0
3: for all possible room sizes:
3:   if a potential room will not intersect existing rooms:
4:     place the room
5:     Fr=1
6:     break from for loop
7: for all possible corridors of any direction and length 3 to 7:
8:   if a potential corridor will not intersect existing rooms:
9:     place the corridor
10:    Fc=1
11:    break from for loop
12:if Fr=1 or Fc=1:
13:  go to 2
```

It should be noted that the examples provided with the 'blind' and 'look ahead' digger agents show naive, simple approaches; Figures 3.7 and 3.8 show to a large degree 'worst-case scenarios' of the algorithm being run, with resulting dungeons being either overlapping or prematurely terminated. While simpler or more complex code additions to the provided digger behavior can avert many of these problems, the fact still remains that it is difficult to anticipate such problems without running the agent's algorithm on extensive trials. This may be a desirable attribute, as the

uncontrollability of the algorithm may result in organic, realistic caves (simulating human miners trying to tunnel their way towards a gold vein) and reduce the dungeon's predictability to a player, but may also result in maps that are unplayable or unentertaining. Moreso than most approaches presented in this chapter, the digger agent's parameters can have a very strong impact on the playability and entertainment value of the generated artifact and tweaking such parameters to best effect is not a straightforward or easy task.

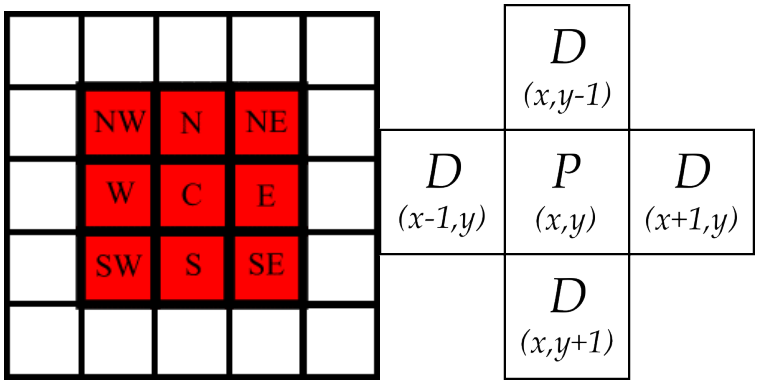
3.3 Cellular automata

A cellular automaton (plural: cellular automata) is a discrete computational model. Cellular automata are widely studied in computer science, physics and even some branches of biology, as models of computation, growth, development, physical phenomena etc. While cellular automata have been the subject of a prohibitive number of investigations leading to untold pages of intimidating mathematics, the basic concepts are actually very simple and can be explained in a few paragraphs and a picture or two.

A cellular automaton consists of an n-dimensional grid, a set of states and a set of transition rule. Most cellular automata are either 1-dimensional (vectors) or 2-dimensional (matrices). Each cell can be in one of several states; in the simplest case, cells can be *on* or *off*. The distribution of cell states at the beginning of an experiment (at time t_0) is the initial state of the cellular automaton. From then on, the automaton evolves (not in the Darwinian sense) in discrete steps based on the rules of that particular automaton. At each time t, each cell decides its new state based on the state of itself and all of the cells in its *neighbourhood* at time $t-1$.

The neighbourhood defines which cells around a particular cell c affects c's future state. For one-dimensional cellular automata, the neighbourhood is defined by its size, i.e. how many cells to the left or right the neighbourhood stretches. For two-dimensional automata, the two most common types of neighbourhoods are *Moore neighbourhoods* and *von Neumann neighbourhoods*. Both neighbourhoods can have a size of any whole number, one or greater. A Moore neighbourhood is a square: a Moore neighbourhood of size 1 consists of the eight cells immediately surrounding c, including those surrounding it diagonally. A von Neumann neighbourhood is like a cross centred on c: a von Neumann neighbourhood of size 1 consists of the four cells surrounding c above, below, to the left and to the right (see Figure 3.9).

The number of possible configurations of the neighbourhood equals the number of states for a cell to the power of the number of cells in the neighbourhood. These numbers can quickly become huge, for example a two-state automaton with a Moore neighbourhood of size two has $2^{25} = 33554432$ configurations. For small neighbourhoods, it is common to define the transition rules as a table, where each possible configuration of the neighbourhood is associated with one future state, but for large neighbourhoods the transition rules are usually based on the proportion of cells that are in each state.



(a) Moore neighbourhood (b) von Neumann neighbourhood

Fig. 3.9: Two types of neighbourhoods for cellular automata.

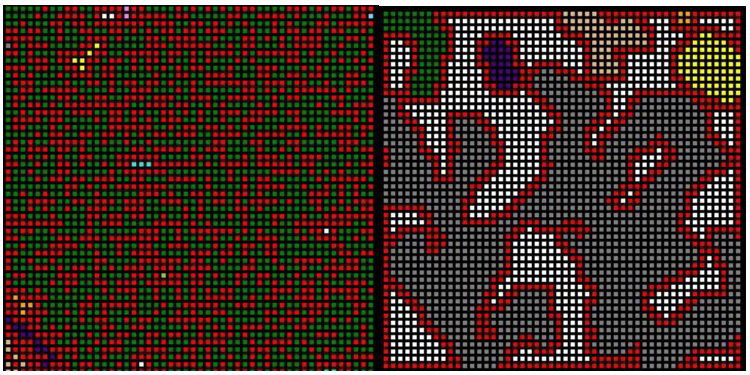
Cellular automata are very versatile, and several types have been shown to be Turing complete. It has even been argued that they could form the basis for a new way of understanding nature through bottom-up modelling [20]. However, in this chapter we will mostly concern ourselves with how they can be used for procedural content generation.

3.3.1 Example: generating infinite caves

In a paper from 2010, Johnson et al. describe a system for generating infinite cave-like dungeons using cellular automata [3]. The motivation was to create an infinite cave crawling game, with environments stretching out endlessly and seamlessly in every direction. An additional design constraint is that the caves are supposed to look organic or eroded, rather than having straight edges and angles. No storage medium is large enough to store a truly endless cave, so the content must be generated at runtime, as players choose to explore new areas. The game does not scroll but instead presents the environment one screen at a time, which offers a time window of a few hundred milliseconds to create a new room every time the player exits a room.

Each room is a 50x50 grid, where each cell can be in one of two states: *empty* or *rock*. Initially, the grid is empty. The generation of a single room works as follows.

- The grid is "sprinkled" with rock: for each cell, there is probability r (e.g. 0.5) that it is turned into rock. This results in a relatively uniform distribution of rock cells.



(a) Random map (b) CA map

Fig. 3.10: Cave generation: Comparison between a CA and a randomly generated map ($r = 0.5$ in both maps); CA parameters: $n = 4$, $M = 1$, $T = 5$. Rock and wall cells are represented by red and white colour respectively. Coloured areas represent different tunnels (floor clusters).

- A cellular automaton is applied to the grid for n (e.g. 2) steps. The single rule of this cellular automaton is that a cell turns into rock in the next time step if at least T (e.g. 5) of its neighbours are rock, otherwise it will turn into free space.
- For aesthetic reasons the rock cells that border on empty space are designated as "wall" cells, which are functionally rock cells but look different.

This simple procedure generates a surprisingly lifelike cave-room. Figure 3.10 shows a comparison between a random map (sprinkled with rock) and the results of a few iterations of the cellular automaton.

But while this generates a single room, the game requires a number of connected rooms. A generated room might not have any openings in the confining rock at all, and there is certainly no guarantee that any exits align with entrances to the adjacent rooms at all. Therefore, whenever a room is generated, its immediate neighbours are also generated. If there is no connection between the largest empty spaces in the two rooms, a tunnel is drilled between those areas at the point where they are least separated. Two more iterations of the cellular automaton are then run on all nine neighbouring rooms together, to smooth out any sharp edges. Figure 3.11 shows the result of this process, in the form of nine rooms that seamlessly connect. This generation process is extremely fast, and can generate all nine rooms in less than a millisecond on a modern computer.

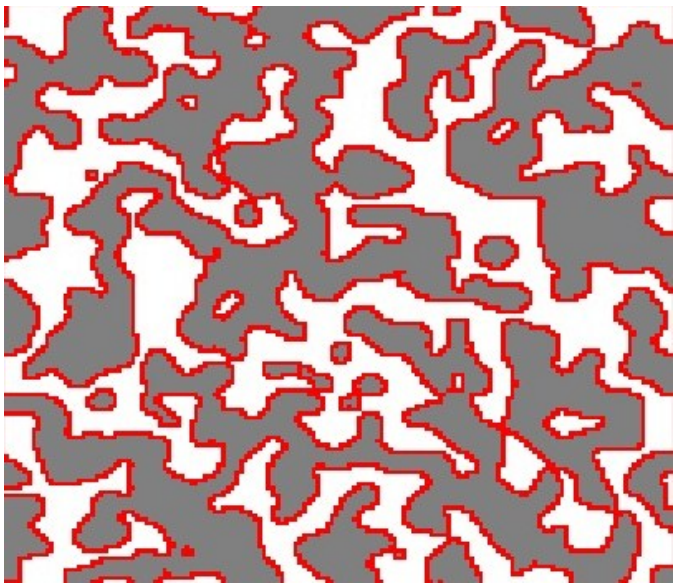


Fig. 3.11: Cave generation: a 3×3 base grid map generated with CA. Rock and wall cells are represented by red and white colour respectively. Grey areas represent floor. ($M = 2; T = 13; n = 4; r = 50\%$)

3.4 Spelunky

Spelunky is a 2D rough-like platform indie game originally created by Derek Yu in 2008. The PC version of the game is available for free. An update version of the game was later released for the Xbox Live Arcade in 2012 with better graphics and more content. An enhanced edition was also released on PC in 2013. The gameplay in Spelunky consists of traversing the 2D levels, collecting items, killing enemies and finding your way to the end of the level. To win the game, the player needs to have good skills in managing different types of resources such as ropes, bumps and money.

The game consists of four groups of level of increasing difficulty. Each set of levels has a distinguished layout and introduces new challenges and new types of enemies. An example level from the second set is presented in Fig. 3.12.

The standout feature of Spelunky is the procedural generation of game content. The use of PCG allowed the generation of endless variations of content that are unique in every playthrough. Losing the game at any level requires restarting the game from the beginning.

Each level in Spelunky is divided into a 4×4 grid of 16 rooms with two rooms marking the start and the end of the level (see Fig. 3.13) and corridors connecting adjacent rooms. Not all the rooms are necessarily connected. In Fig. 3.13 there are

Fig. 3.12: Snapshot from Spelunky.

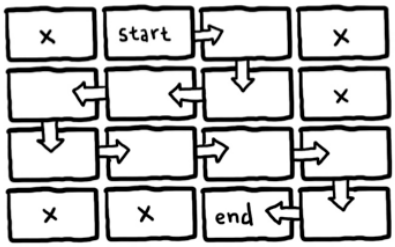


Fig. 3.13: Level generation in Spelunky.

some isolated rooms such as the ones at the top left and down left corners. In order to reach these rooms, the player needs to use bombs, which are a limited resource, to destroy the walls.

The layout of each room is selected from a set of predefined templates. An example template for one of the rooms presented in Fig. 3.13 can be seen in Fig. 3.14. In each template, a number of chunks are marked in which randomisation can occur. Whenever a level is being generated, these chunks are replaced by different types of obstacle according to a set of randomised number generators [6]. Following this method, a new variation of the level can be generated with each run of the algorithm.

Each room in Spelunky consists of 80 tiles arranged in a 8×10 metrics [2]. An example room template can be:

```
0000000011
0060000L11
0000000L11
0000000L11
0000000L11
0000000011
0000000011
1111111111
```

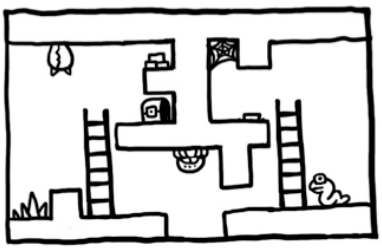


Fig. 3.14: An example design of a room in Spelunky.

Where 0 represents an empty cell, 1 stands for walls or bricks, L for ladders. The 6 in this example can be replaced by random obstacles making it possible to generate different variations. The obstacles, or traps, are usually of 5×3 blocks of tiles that overwrite the original tiles. Examples traps included in the game can be spikes, webs, monsters bots or arrow traps, to name some.

While the basic layout of the level is partially random, with the presence of opportunities for variations, the placement of monsters and traps is 100% random. After generating the physical layout, the level map is scanned for potential places where monsters can be generated. These include for example, a brick with empty tiles behind that offer enough space for generating a spider. There are another set of random numbers that control the generation of monsters. These numbers control the type and the frequency of the generation. For example, there is a 20% chance of creating a giant spider and once a spider is generated, this probability is set to 0 preventing the existence of more than one giant spider in a level.

In this sense, level generation in Spelunky can be seen as a composition of three main phases: in the first phase, the main layout of the level is generated by choosing the rooms from the templates available and defining the entrance and exit points. The second phase is obstacle generation which can be thought of as an agent going through the level and placing obstacles in predefined spaces according to a set of heuristics. The final phase is the monsters generation phase where another agent search the level and place a monster when proper space is found and a set of conditions is satisfied.

3.5 Infinite Mario Bros

Super Mario Bros is a very popular 2D platform game developed by Nintendo and released in the mid eighties [7]. A public domain clone of the game, named *Infinite Mario Bros.* (IMB) was later published by Markus Persson. IMB features the art assets and general game mechanics of Super Mario Bros. but differs in level construction. Infinite Mario Bros is playable on the web, where Java source code is also

available[1]. While implementing most features of Super Mario Bros, the standout feature of Infinite Mario Bros is the automatic generation of levels. Every time a new game is started, levels are randomly generated by traversing the level map and adding features according to certain heuristics.

The internal representation of the levels in Infinite Mario Bros is a two-dimensional array of game elements. In "small" state, Mario is one block wide and one block high. Each position in the array can be filled with a brick block, a coin, an enemy or nothing. The levels are generated by placing the game elements in the two-dimensional level map.

Different approaches can be followed to generate the levels [15, 13, 9, 17]. In the following we describe one possible approach.

3.5.1 Probabilistic Multi-pass

The Probabilistic Multi-pass Generator (PMPG) was created by Ben Weber [15] as an entry for the level generation track of the Mario AI Championship [10]. The generator is an agent based and works by first creating the base level and then performing a number of passes through it. Level generation consists of six passes from left to right and adding one of the different types of game elements. Each pass is associated with a number of events (14 in total) that may occur according to predefined uniform probability distributions. These distributions are manually weighted and by tweaking these weights one can gain control over the frequency of different elements such as gaps, hills and enemies.

The six passes considered are:

1. An initial pass that change the basic structure of the level by changing the hight of the ground and starting or ending a gap,
2. the second pass adds the hills in the background,
3. the third pass adds the static enemies such as pipes and cannons based on the basic platform generated,
4. moving enemies such as koopas and goombas are added in the fourth pass,
5. the fifth pass adds the unconnected horizontal blocks, and finally,
6. the sixth pass places coins throughout the level.

Playability, or the existence of a path from the starting to the ending point, is guaranteed by imposing constraints of the different items created or places. For example, the width of generated gaps is limited by the maximum number of blocks that the player can jump over and the hight of pipes is limited to ensure that the player can pass through.

[1] http://www.mojang.com/notch/mario/

3.6 Lab session: Level generator for InfiTux (and Infinite Mario)

InfiTux, short for Infinite Tux, is a 2D platform game built by combining the underlying software used to generate the levels for Infinite Mario Bros (IMB) with the art and sound assets of *Super Tux* [19]. The game was created to replace IMB which is the platform extensively used in research [8, 16, 18, 5, 1] and the software for the Mario AI Championship [14, 12, 4]. Since the level generator for InfiTux is the same as the one used for IMB, the game features infinite variations of levels by the use of a random seed. The level of difficulty can also be varied using different difficulty values which control the number, frequency and types of the obstacles and monsters.

The purpose of this exercise is to use one or more of the methods presented in this chapter to implement your own generator that creates content for the game. The software you will be using is the one used for the Level Generation Track of the Platformer AI Competition [11]; a successor to the Mario AI Championship that is based on InfiTux. The software provides an interface that eases the interaction with the system and is a good starting point. You can either modify the original level generator, or use it as an inspiration. In order to help you to start with the software, we will in the following describe the main components of the interface provided and how it can be used.

As the software is developed for the Level Generation track of the competition, which invites participants to submit level generator that are fun for specific players, the interface incorporates information about player behaviour that you could use while building your generator. This information is collected while the player is playing a test level and stored in a gameplay metrics that contains statistical features extracted from a gameplay session. The features include, for example, the number of jumps, the time spent running, the number of items collected and the number of enemies killed.

For your generator to work properly, your level should implement the *LevelInterface* which specifies how the level is constructed and how different types of elements are scattered around the level:

```
public byte[][] getMap();
public SpriteTemplate[][] getSpriteTemplates()
```

The size of the level map is 320×15 and you should implement a method of your choice to fill in the map. Note that the basic structure of the level is saved in a different map than the one used to store the placement of enemies.

The level generator, which passes the gameplay metrics to your level and communicates with the simulator, should implement the *LevelGenerator* interface:

```
public LevelInterface generateLevel(GamePlay playerMet);
```

There are quite a few examples reported in the literature that use this software for content creation, some of them are part of the Mario AI Championship and their implementation is open source and freely available at the competition website [10].

References

1. Dahlskog, S., Togelius, J.: Patterns as objectives for level generation (2013)
2. Darius Kazemi: URL http://tinysubversions.com/2009/09/spelunkys-procedural-space/
3. Johnson, L., Yannakakis, G.N., Togelius, J.: Cellular Automata for Real-time Generation of Infinite Cave Levels. In: Proceedings of the ACM Foundations of Digital Games. ACM Press (2010)
4. Karakovskiy, S., Togelius, J.: The mario AI benchmark and competitions. Computational Intelligence and AI in Games, IEEE Transactions on **4**(1), 55–67 (2012)
5. Kerssemakers, M., Tuxen, J., Togelius, J., Yannakakis, G.: A procedural procedural level generator generator. In: IEEE Conference on Computational Intelligence and Games (CIG),, pp. 335–341 (2012)
6. Make Games: URL http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky
7. Nintendo Creative Department: (1985). Super Mario Bros, Nintendo
8. Ortega, J., Shaker, N., Togelius, J., Yannakakis, G.N.: Imitating human playing styles in Super Mario Bros. Entertainment Computing pp. 93–104 (2012)
9. Shaker, N., Nicolau, M., Yannakakis, G.N., Togelius, J., ONeill, M.: Evolving levels for super mario bros using grammatical evolution. pp. 304–311 (2012)
10. Shaker, N., Togelius, J., Karakovskiy, S., Yannakakis, G.: Mario AI Championship. URL http://marioai.org/
11. Shaker, N., Togelius, J., Yannakakis, G.: Platformer AI Competition. URL http://platformerai.com/
12. Shaker, N., Togelius, J., Yannakakis, G., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., Smith, G., Baumgarten, R.: The 2010 mario ai championship: Level generation track. IEEE Transactions on Computational Intelligence and AI in Games, **3**(4), 332–347 (2011)
13. Shaker, N., Togelius, J., Yannakakis, G.N.: Towards automatic personalized content generation for platform games. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE) (2010)
14. Shaker, N., Togelius, J., Yannakakis, G.N., Poovanna, L., Ethiraj, V.S., Johansson, S.J., Reynolds, R.G., Heether, L.K., Schumann, T., Gallagher, M.: The turing test track of the 2012 mario ai championship: Entries and evaluation. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG) (2013)
15. Shaker, N., Togelius, J., Yannakakis, G.N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., Smith, G., Baumgarten, R.: The 2010 Mario AI championship: Level generation track. IEEE Transactions on Computational Intelligence and Games pp. 332–347 (2011)
16. Shaker, N., Yannakakis, G.N., Togelius, J., Nicolau, M., ONeill, M.: Fusing visual and behavioral cues for modeling user experience in games. IEEE Transactions on System Man and Cybernetics; Part B: Special Issue on Modern Control for Computer Games
17. Sorenson, N., Pasquier, P.: Towards a generic framework for automated video game level creation. In: Proceedings of the European Conference on Applications of Evolutionary Computation (EvoApplications), pp. 130–139. Springer LNCS (2010)
18. Sorenson, N., Pasquier, P., DiPaola, S.: A generic approach to challenge modeling for the procedural creation of video game levels. IEEE Transactions on Computational Intelligence and AI in Games (3), 229–244 (2011)
19. SuperTux Development Team: Super Tux. URL http://supertux.lethargik.org/
20. Wolfram, S.: Cellular automata and complexity: collected papers, vol. 1. Addison-Wesley Reading (1994)