Chapter 3
Constructive generation methods for dungeons and levels

Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra

Abstract This chapter addresses a specific type of game content, the dungeon, and a number of commonly used methods for generating such content. These methods are all "constructive", meaning that they run in fixed (usually short) time, and do not evaluate their output in order to re-generate it. Most of these methods are also relatively simple to implement. And while dungeons, or dungeon-like environments, occur in a very large number of games, these methods can often be made to work for other types of content as well. We finish the chapter by talking about some constructive generation methods for *Super Mario Bros.* levels.

3.1 Dungeons and levels

A dungeon, in the real world, 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 noble people, fall into traps and ultimately exit at another point. This conception of dungeons probably originated with the role-playing board 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, features procedural runtime dungeon generation; the *Diablo* series is a high-profile series of games in this tradition. Because of this close relationship with such successful games, and also due to the unique control challenges in their design, dungeons are a particularly active and attractive PCG subject.

For the purposes of this chapter, we define adventure and RPG dungeon levels as labyrinthic environments, consisting mostly of interrelated challenges, rewards and puzzles, tightly paced in time and space to offer highly structured gameplay progressions [9]. An aspect which sets dungeons apart from other types of levels

is a sophisticated notion of gameplay pacing and progression. Although dungeon levels are open for free player exploration (more than, say, platform levels), this exploration has a tight bond with the progression of challenges, rewards, and puzzles, as designed by the game's designer. And in contrast to platform levels or race tracks, dungeon levels encourage free exploration while keeping strict control over gameplay experience, progression and pacing (unlike open worlds, where the player is more independent). For example, players may freely choose their own dungeon path among different possible ones, but never encounter challenges that are impossible for their current skill level (since the space to backtrack is not as open as, for example, a sandbox city). Designing dungeons is thus a sophisticated exercise of emerging a complex game space from predetermined desired gameplay, rather than the other way around.

In most adventure games and RPGs, dungeons structurally consist of several rooms connected by hallways. While originally the term 'dungeon' refers to a labyrinth of prison cells, in games it may also refer to caves, caverns, or human-made structures. Beyond geometry and topology, dungeons include non-player characters (e.g. monsters to slay, princesses to save), decorations (typically fantasy-based) and objects (e.g. treasures to loot).

Procedural generation of dungeons refers to the generation of the topology, geometry and gameplay-related objects of this type of level. A typical dungeon generation method consists of three elements:

1. A representational model: an abstract, simplified representation of a dungeon, providing a simple overview of the final dungeon structure.
2. A method for constructing that representational model.
3. A method for creating the actual geometry of a dungeon from its representational model.

Above, we distinguished dungeons from platform levels. However, there are also clear similarities between these two types of game level. Canonical examples of platform game levels include those in *Super Mario Bros.* and *Sonic the Hedgehog*; a modern-day example of a game in this tradition that features procedural level 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, in the game mechanics of platformers, 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. As a result, the interplay of platforms and gaps is an essential element in the vocabulary of platform game levels.

In this chapter, we will study a variety of methods for procedurally creating dungeons and platform game levels. Although these methods may be very disparate, they have one feature in common: they are all constructive, producing only one output instance per run, in contrast with e.g. search-based methods. They also have in common that they are fast; some are even successful in creating levels at runtime. In general, these methods provide (rather) limited control over the output and its properties. The degree of control provided is nowadays a very important characteristic of any procedural method. By "control" we mean the set of options that a designer

(or programmer) has in order to purposefully steer the level-generation process, as well as the amount of effort that steering takes. Control also determines whether editing those options and parameters causes sensible output changes, i.e. the intuitive responsiveness of a generator. Proper control assures that a generator creates consistent results (e.g. playable levels), while maintaining both the set of desired properties and variability.

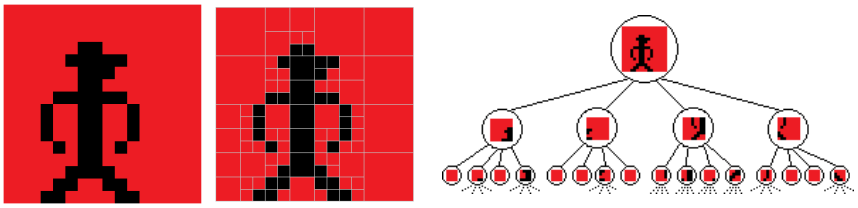
We will discuss several families of procedural techniques. For simplicity, each of these techniques will be presented in the context of a single content type, either dungeons or platform game levels. The first family of algorithms to be discussed in this chapter is space partitioning. Two different examples of how dungeons can be generated by space partitioning are given; the core idea is to recursively divide the available space into pieces and then connect these pieces to form the dungeon. This is followed by a discussion of agent-based methods for generating dungeons, with the core idea that agents dig paths into a primeval mass of matter. The next family of algorithms to be introduced is cellular automata, which turn out to be a simple and fast means of generating structures such as cave-like dungeons. Generative grammars, yet another family of procedural methods, are discussed next, as they can naturally capture higher-level dungeon design aspects. We then turn our attention to several methods that were developed for generating platform levels, some of which are applicable to dungeons as well. The chapter ends with a discussion of the platform level generation methods implemented in the commercial game *Spelunky* and the open-source framework *Infinite Mario Bros.*, and its recent offshoot *InfiniTux*. The lab exercise will have you implement at least one method from the chapter using the *InfiniTux* API.

3.2 Space partitioning for dungeon generation

True to its name, a space-partitioning algorithm yields a *space partition*, i.e. a subdivision of a 2D or 3D space into disjoint subsets, so that any point in the space lies in exactly one of these subsets (also called *cells*). Space-partitioning algorithms often operate hierarchically: each cell in a space partition is further subdivided by applying the same algorithm recursively. This allows space partitions to be arranged in a so-called *space-partitioning tree*. Furthermore, such a tree data structure allows for fast geometric queries regarding any point within the space; this makes space partitioning trees particularly important for computer graphics, enabling, for example, efficient raycasting, frustum culling and collision detection.

The most popular method for space partitioning is *binary space partitioning* (BSP), which recursively divides a space into two subsets. Through binary space partitioning, the space can be represented as a binary tree, called a *BSP tree*.

Different variants of BSP choose different splitting hyperplanes based on specific rules. Such algorithms include quadtrees and octrees: a quadtree partitions a two-dimensional space into four quadrants, and an octree partitions a three-dimensional space into eight octants. We will be using quadtrees on two-dimensional images as



(a) Image and partition

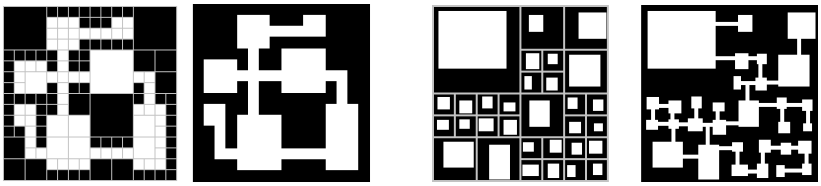
(b) Quadtree

Fig. 3.1: Example quadtree partition of a binary image (0 shown as red, 1 as black). Large areas of a single colour, 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 colour result in a quadtree with 94 leaf nodes. The first layers of the tree are shown in (b): 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)

the simplest example. 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 total number of tree nodes and depth depends 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 within any quadrant have different colours, that quadrant is subdivided; the process is applied recursively until each leaf quadrant (regardless of size) contains only pixels of the same colour (see Figure 3.1).

When space-partitioning algorithms are used in 2D or 3D graphics, their purpose is typically to represent existing elements such as polygons or pixels rather than to create new ones. However, the principle that space partitioning results in disjoint subsets with no overlapping areas is particularly suitable for creating rooms in a dungeon or, in general, distinct areas in a game level. Dungeon generation via BSP follows a *macro* approach, where the algorithm acts as an all-seeing dungeon architect rather than a "blind" digger as is often the case with the agent-based approaches presented in Section 3.3. The entire dungeon area is represented by the root node of the BSP tree and is partitioned recursively until a terminating condition is met (such as a minimum size for rooms). The BSP algorithm guarantees 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 Figure 3.2a. Furthermore, the principle that a leaf quadrant must consist of a uniform element (or of same-colour pixels,



(a) (b)

Fig. 3.2: (a) A dungeon created using a quadtree, with each cell consisting entirely of empty space (black) or rooms (white). (b) A dungeon created using a quadtree, but with each quadrant containing a single room (placed stochastically) as well as empty space; corridors are added after the partitioning process is complete

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, this permits for 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 taking the quadtree into account at all. Even with this added stochasticity, dungeons are still likely to be very neatly ordered (see Figure 3.2b).

We now describe an even more stochastic approach loosely based on BSP 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 partition cells 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). In the end, each partition cell 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 pseudocode of the generative algorithm, and Figures 3.3 and 3.4 show 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 partition cells
4: if this cell is bigger than the minimal acceptable size:
5:   go to step 2 (using this cell as the area to be divided)
6: select the other partition cell, and go to step 4
7: for every partition cell:
8:   create a room within the cell by randomly
     choosing two points (top left and bottom right)
     within its boundaries
9: starting from the lowest layers, draw corridors to connect
   rooms corresponding to children of the same parent
```

```
   in the BSP tree
10:repeat 9 until the children of the root node are connected
```

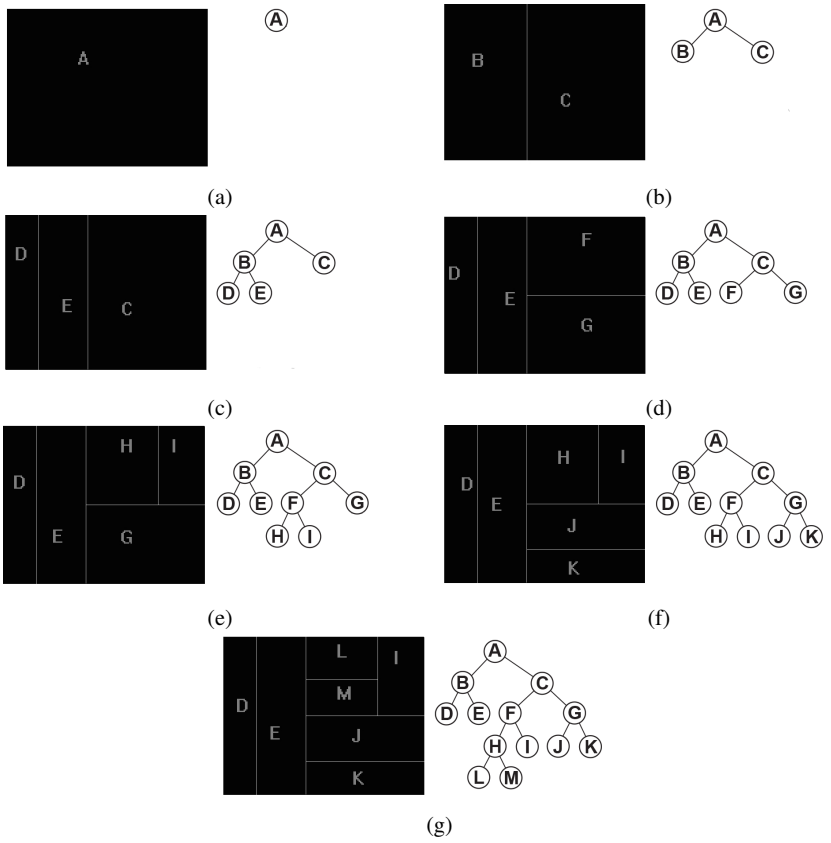


Fig. 3.3: Stochastically partitioning the dungeon area A, which is contained in the root node of the BSP tree. Initially the space is split into B and C via a vertical line (its x-coordinate is determined randomly). The smaller area B is split further with a vertical line into D and E; both D and E are 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 partition cells of G (J and K) are too small to be split further, and so is partition cell I of F. Cell 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. Figure 3.4 demonstrates room and corridor placement for this dungeon

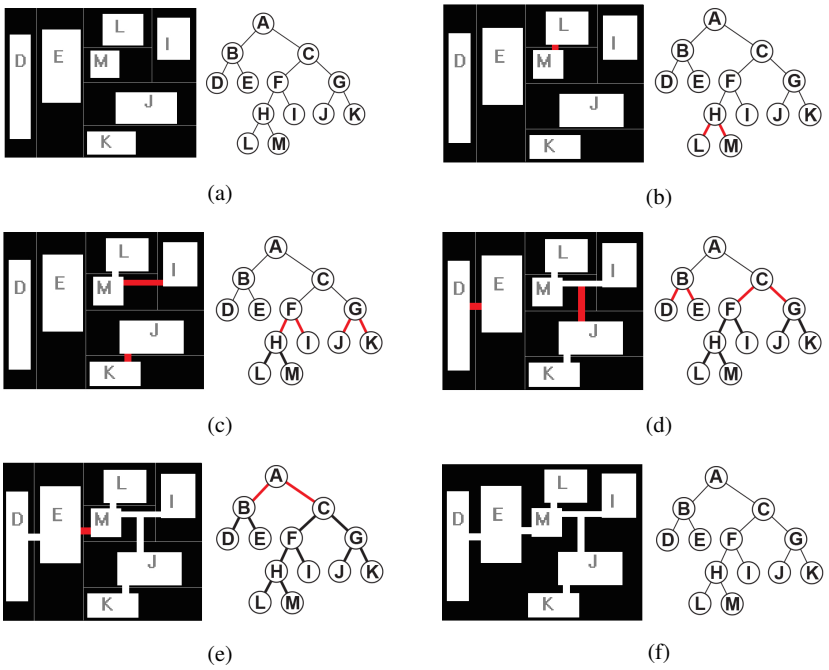


Fig. 3.4: Room and corridor placement in the partitioned dungeon of Figure 3.3. (a) For each leaf node in the BSP tree, a room is placed by randomly choosing coordinates for top left and bottom right corners, within the boundaries of the partition cell. (b) A corridor is added to connect the leaf nodes of the lowest layer of the tree (L and M); for all purposes, the algorithm will now consider rooms L and M as joined, grouping them together as their parent H. (c) Moving up the tree, H (the grouping of rooms L and M) is joined via a corridor with room I, and rooms J and K are joined via a corridor into their parent G. (d) 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. (e) Finally, the two subtrees of the root node are joined together and (f) the dungeon is fully connected

While binary space partitioning was primarily used here to create non-overlapping rooms, the hierarchy of the BSP tree can be used for other aspects of dungeon generation as well. The example of Figure 3.4 demonstrates how room connectivity can be determined by the BSP tree: using corridors to connect rooms corresponding to children of the same parent reduces the chances of overlapping or intersecting corridors. Moreover, non-leaf partition cells can be used to define 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 the BSP tree hierarchy, these groups of rooms may have a single entrance from the rest of the dungeon; this allows such a room

to be decorated as a prison or as an area with dimmer light, favouring players who excel at stealthy gameplay. Some examples of themed dungeon partitions are shown in Figure 3.5.

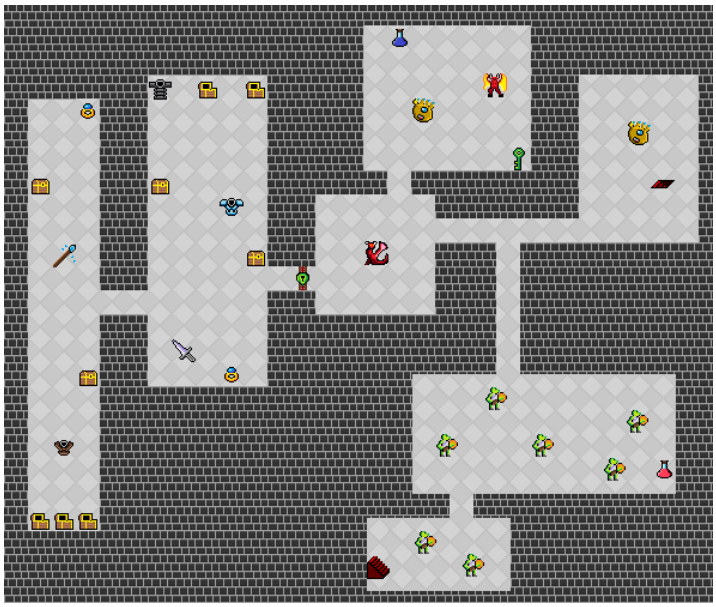


Fig. 3.5: The example dungeon from Figure 3.4, using the partitions to theme the room contents. Partition cells B and C are only connected by a single corridor; this allows the rooms of partition B to be locked away (green lock), requiring a key from cell C in order to be accessed (room L). Similarly, rooms of cell B contain only treasures and rewards, while rooms of partition C contain predominantly monsters. Moreover, the challenge rating of monsters in cell C is split between its child nodes: partition G contains weak goblins while cell F contains challenging monsters with magical powers. Further enhancements could increase the challenge of cell G by making it darker (placing fewer light sources), using different textures for the floor and walls of cell B, or changing the shape of rooms in cell C to circular

3.3 Agent-based dungeon growing

Agent-based approaches to 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, 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 organised dungeons of Section 3.2. 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. The 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 not create a dungeon with rooms overlapping 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.

There is an infinite number 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 three and seven 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 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%. Figure 3.6 shows an example run of the algorithm, and its pseudocode is below.

```
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 length 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

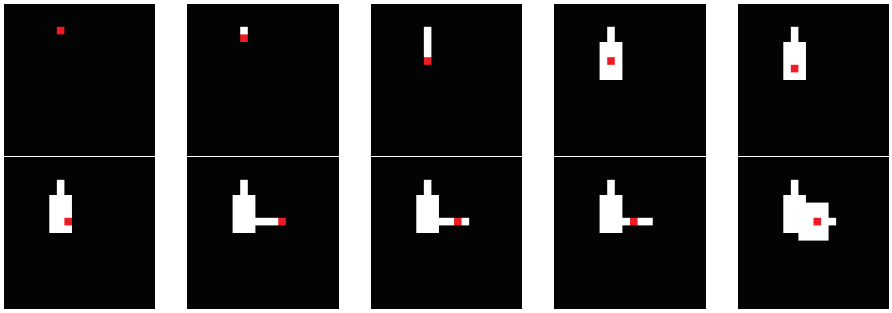


Fig. 3.6: A short run of the stochastic, "blind" digger. The digger starts at a random tile on the map (1st image), and starts digging downwards. After digging 5 tiles (3rd image), the chance of adding a room is 25%, and it is rolled, resulting in the 4th image. The agent continues moving downwards (4th 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 is 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

informed about the overall appearance of the dungeon and look ahead to see 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.

We will consider a less stochastic agent with look-ahead as a second example. As 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. Figure 3.7 shows an example run of the algorithm, and below is its pseudocode.

```
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
```

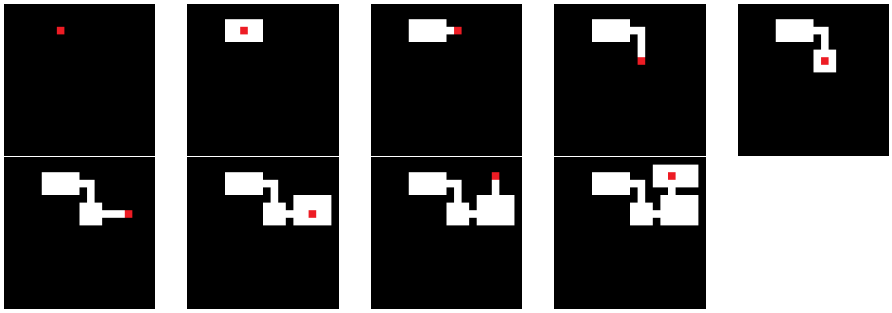
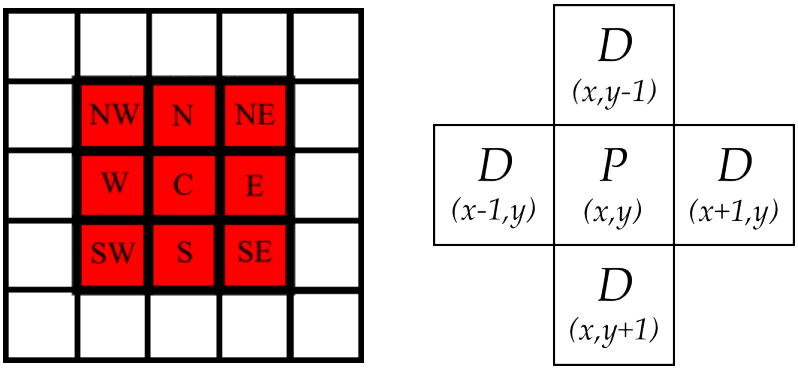


Fig. 3.7: A short run of the informed, "look ahead" digger. The digger starts at a random tile on the map (1st image), and places a room (2nd image) and a corridor (3rd image) since there can't be any overlaps in the empty dungeon. After placing the first corridor, there is no space for a room (provided 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

```
11:    break from for loop
12:if Fr=1 or Fc=1:
13:  go to 2
```

The examples provided with the "blind" and "look-ahead" digger agents show naive, simple approaches; Figures 3.6 and 3.7 show to a large degree worst-case scenarios of the algorithm being run, with resulting dungeons either overlapping or being prematurely terminated. While simpler or more complex code additions to the provided digger behaviour 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 it may also result in maps that are unplayable or unentertaining. More 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 artefact and tweaking such parameters to best effect is not a straightforward or easy task.



(a) Moore neighbourhood

(b) von Neumann neighbourhood

Fig. 3.8: Two types of neighbourhoods for cellular automata. Adapted from Wikipedia

3.4 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 many publications, 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 rules. Most cellular automata are either one-dimensional (vectors) or two-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 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 affect 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 3.8).

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 2 has $2^{25} = 33,554,432$ 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.

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 [28]. However, in this chapter we will mostly concern ourselves with how they can be used for procedural content generation.

Johnson et al. [4] describe a system for generating infinite cave-like dungeons using cellular automata. 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 in which to create a new room every time the player exits a room.

This method uses the following four parameters to control the map generation process:

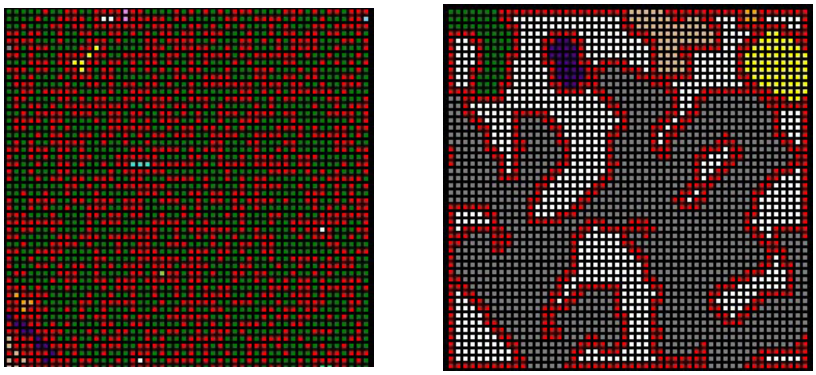
- A percentage of rock cells (inaccessible areas);
- The number of cellular automata generations;
- A neighbourhood threshold value that defines a rock (T=*5*);
- The number of neighbourhood cells.

Each room is a 50×50 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 rocks: 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 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 3.9 shows a comparison between a random map (sprinkled with rocks) 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 rocks, and



(a) Random map

(b) CA map

Fig. 3.9: 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 white and red colour respectively. Coloured areas represent different tunnels (floor clusters). Adapted from [4]

there is no guarantee that any exits align with entrances to the adjacent rooms. 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 3.10 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.

We can conclude that the small number of parameters, and the fact that they are relatively intuitive, is an asset of cellular automata approaches like Johnson et al.'s. However, this is also one of the downsides of the method: for both designers and programmers, it is not easy to fully understand the impact that a single parameter has on the generation process, since each parameter affects multiple features of the generated maps. It is not possible to create a map that has specific requirements, such as a given number of rooms with a certain connectivity. Therefore, gameplay features are somewhat disjoint from these control parameters. Any link between this generation method and gameplay features would have to be created through a process of trial and error.

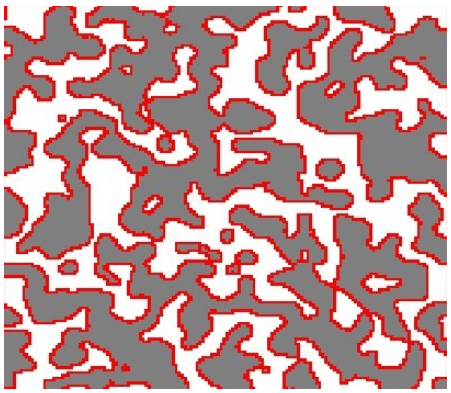


Fig. 3.10: Cave generation: a 3×3 base grid map generated with CA. Rock and wall cells are represented by white and red colour respectively. Grey areas represent floor. ($M = 2; T = 13; n = 4; r = 50\%$). Adapted from [4]

3.5 Grammar-based dungeon generation

Generative grammars were originally developed to formally describe structures in natural language. These structures—phrases, sentences, etc.—are modelled by a finite set of recursive rules that describe how larger-scale structures are built from smaller-scale ones, grounding out in individual words as the terminal symbols. They are *generative* because they describe linguistic structures in a way that also describes how to generate them: we can sample from a generative grammar to produce new sentences featuring the structures it describes. Similar techniques can be applied to other domains. For example, graph grammars [15] model the structure of graphs using a similar set of recursive rules, with individual graph nodes as the terminal symbols.

Back to our topic of dungeon generation, Adams [1] uses graph grammars to generate first-person shooter (FPS) levels. FPS levels may not obviously be the same as dungeons, but for our purposes his levels qualify as dungeons, because they share the same structure, a maze of interconnected rooms. He uses the rules of a graph grammar to generate a graph that describes a level's topology: nodes represent rooms, and an edge between two rooms means that they are adjacent. The method doesn't itself generate any further geometric details, such as room sizes. An advantage of this high-level, topological representation of a level is that graph generation can be controlled through parameters such as difficulty, fun, and global size. A search algorithm looks for levels that match input parameters by analyzing all results of a production rule at a given moment, and selecting the rule that best matches the specified targets.

One limit of Adams' work is the ad-hoc and hard-coded nature of its grammar rules, and especially the parameters. It is a sound approach for generating the topological description of a dungeon, but generalizing it to a broader set of games and

goals would require creating new input parameters and rules each time. Regardless, Adams' results showcase the motivation and importance of controlling dungeon generation through gameplay.

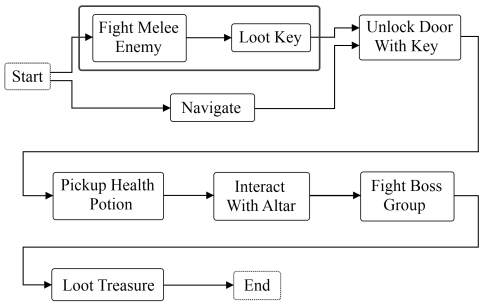
Dormans' work [3] is more extensively covered in Chapter 5, so here we only briefly refer to his use of generative grammars to generate dungeon spaces for adventure games. Through a graph grammar, missions are first generated in the form of a directed graph, as a model of the sequential tasks that a player needs to perform. Subsequently, each mission is abstracted to a network of nodes and edges, which is then used by a shape grammar to generate a corresponding game space.

This was the first method to successfully introduce gameplay-based control, most notably with the concept of a mission grammar. Still, the method does not offer real control parameters, since control is actually exerted by the different rules in the graph and shape grammars, which are far from intuitive for most designers.

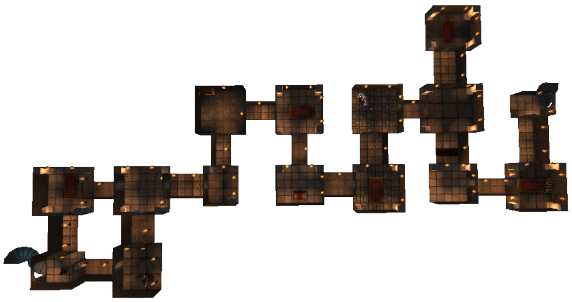
Inspired by the work of Dormans, van der Linden et al. [8] proposed the use of gameplay grammars to generate dungeon levels. Game designers express a-priori design constraints using a gameplay-oriented vocabulary, consisting of player actions to perform in-game, their sequencing and composition, inter-relationships and associated content. These designer-authored constraints directly result in a generative graph grammar, a so-called *gameplay grammar*, and multiple grammars can be expressed through different sets of constraints. A grammar generates graphs of player actions, which subsequently determine layouts for dungeon levels. For each generated graph, specific content is synthesized by following the graph's constraints. Several proposed algorithms map the graph into the required game space and a second procedural method generates geometry for the rooms and hallways, as required by the graph.

This approach aims at improving gameplay-based control on a generic basis, as it provides designers with the tools to effectively create, from scratch, grammar-based generators of graphs of player actions. The approach is generic, in the sense that such tools are not connected to any domain, and player actions and related design constraints can be created and manipulated across different games. However, integration of graphs of player actions in an actual game requires a specialized generator, able to transform such a graph into a specific dungeon level for that game. Van der Linden et al. demonstrated such a specialized generator for only one case study, yielding fully playable 3D dungeon levels for the game *Dwarf Quest* [27]. Figure 3.11 show (a) a gameplay graph and (b) a dungeon generated from this method.

As for gameplay-based control, this approach empowers designers to specify and control dungeon generation with a more natural design-oriented vocabulary. Designers can create their own player actions and use them as the vocabulary to control and author the dungeon generator. For this, they specify the desired gameplay which then constrains game-space creation. Furthermore, designers can express their own parameters (e.g. difficulty), which control rule rewriting in the gameplay grammar. Setting such gameplay-based parameters allows for even more fine-grained control over generated dungeons.



(a)



(b)

Fig. 3.11: (a) A gameplay graph created by van der Linden et al. [8] and (b) a corresponding dungeon layout generated for it

3.6 Advanced platform generation methods

In this section, we turn our attention to platform generation methods, by discussing two recent methods that were originally proposed for generating platform levels. Unlike the previous sections, there is no single category or family to characterize these methods. Interestingly, as we will point out, the central concepts of each of them could very well contribute to improve the generation of dungeons as well.

The first method, proposed by Smith et al. [23], is *rhythm-based platform generation*. It proposes level generation based on the notion of rhythm, linked to the timing and repetition of user actions. They first generate small pieces of a level, called rhythm groups, using a two-layered grammar-based approach. In the first layer, a set of player actions is created, after which this set of actions is converted into corresponding geometry. Many levels are created by connecting rhythm groups, and a set of implemented critics selects the best level.

Smith et al. propose a set of 'knobs' that a designer can manipulate to control the generation process, including (i) a general path through the level (i.e. start, end, and intermediate line segments), (ii) the kinds of rhythms to be generated, (iii) the types and frequencies of geometry components, and (iv) the way collectables (coins) are

divided over the level (e.g. coins per group, probability for coins above gaps, etc.). There are also some parameters per created rhythm group, such as the frequency of jumps per rhythm group, and how often specific geometry (springs) should occur for a jump. Another set of parameters provides control over the rhythm length, density, beat type, and beat pattern.

The large number of parameters at different levels of abstraction provides many control options, and allows for the versatile generation of very disparate levels. Furthermore, they relate quite seamlessly to gameplay, especially in the platformer genre. However, this approach could nicely tie in with dungeon generation as well. As with Dormans, a two-layered grammar is used, where the first layer considers gameplay (in this case, player actions) and the second game space (geometry). The notion of *rhythm* as defined by Smith et al. is not directly applicable to dungeons, but the pacing or tempo of going through rooms and hallways could be of similar value in dungeon-based games. The decomposition of a level into rhythm groups also connects very well with the possible division of a dungeon into dungeon-groups with distinct gameplay features such as pacing.

Our second method, proposed by Mawhorter et al. [11] is called Occupancy-Regulated Extension (ORE), and it directly aims at procedurally generating 2D platform levels. ORE is a general geometry assembly algorithm that supports human-design-based level authoring at arbitrary scales. This approach relies on pre-authored chunks of level as a basis, and then assembles a level using these chunks from a library. A chunk is referred to as level geometry, such as a single ground element, a combination of ground elements and objects, interact-able objects, etc. This differs from the rhythm groups introduced by Smith et al. [23], because rhythm groups are separately generated by a PCG method whilst the ORE chunks are pieces of manually created content in a library. The algorithm takes the following steps: (i) a random potential player location (occupancy) is chosen to position a chunk; (ii) a chunk is selected from a list of context-based compatible chunks; (iii) the new chunk is integrated with the existing geometry. This process continues until there are no potential player locations left, after which post-processing takes care of placing objects such as power-ups.

This framework is meant for general 2D platform games, so specific game elements and mechanics need to be filled in, and chunks need to be designed and added to a library. Versatile levels can only be generated given that a minimally interesting chunk library is used.

Mawhorter et al. do not mention specific control parameters for their ORE algorithm, but a designer still has some control. Firstly, the chunks in the library and their probability of occurrence are implicit parameters, i.e. they actually determine the level geometry and versatility, and possible player actions need to be defined and incorporated in the design of chunks. And above all, their mixed-initiative approach provides the largest amount of control one can offer, even from a gameplay-based perspective. However, taken too far, this approach could come too close to manually constructing a level, decreasing the benefits of PCG. In summary, much control can be provided by this method, but the generation process may still be not very

efficient, as a lot of manual work seems to still be required for specific levels to be generated.

This ORE method proposes a mixed-initiative approach, where a designer has the option to place content before the algorithm takes over and generates the rest of the level. This approach seems very interesting also for dungeon generation, where an algorithm that can fill in partially designed levels would be of great value. Imagine a designer placing special event rooms and then having an algorithm add the other parts of the level that are more generic in nature. This mixed-initiative approach would increase both level versatility, and control for designers, while still taking work off their hands. Additionally, it would fit the principles of dungeon design, where special rooms are connected via more generic hallways. Also, using a chunk library fits well in the context of dungeon-level generation (e.g. combining sets of template rooms, junctions and hallways). However, 3D dungeon levels would typically require a much larger and more complex chunk library than 2D platform levels, which share a lot of similar ground geometry.

3.7 Example applications to platform generation

3.7.1 Spelunky

Spelunky is a 2D platform indie game originally created by Derek Yu in 2008 [29]. The PC version of the game is available for free. An updated version of the game was later released in 2012 for the Xbox Live Arcade 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. To win the game, the player needs to have good skills in managing different types of resources such as ropes, bumps and money. Losing the game at any level requires the game to be restarted from the beginning.

The game consists of four groups of maps of increasing level of 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 Figure 3.12.

The standout feature of *Spelunky* is the procedural generation of game content. The use of PCG allows the generation of endless variations of content that are unique in every playthrough.

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 Figure 3.13) and corridors connecting adjacent rooms. Not all the rooms are necessarily connected; in Figure 3.13 there are some isolated rooms such as the ones at the top left and bottom 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 Figure 3.13 can be seen in Fig-

Fig. 3.12: Snapshot from *Spelunky*

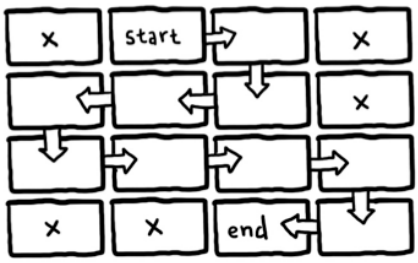


Fig. 3.13: Level generation in *Spelunky*. Adapted from [10]

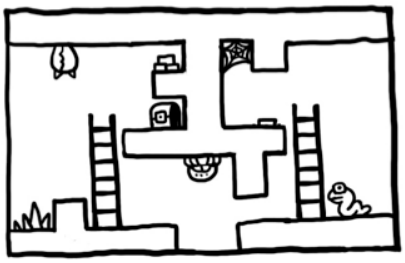


Fig. 3.14: Example room design in *Spelunky*. Adapted from [10]

ure 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 [10]. Following this method, a new variation of the level can be generated with each run of the algorithm.

More specifically, each room in *Spelunky* consists of 80 tiles arranged in an 8×10 matrix [6]. An example room template can be:

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

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 permitting the generation of different variations. The obstacles, or traps, are usually of 5×3 blocks of tiles that overwrite the original tiles. Example traps included in the game can be spikes, webs 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 is another set of random numbers that controls the generation of monsters. These numbers control the type and the frequency of 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 monster-generation phase, where another agent searches the level and places a monster when enough space is found and a set of conditions is satisfied.

3.7.2 Infinite Mario Bros.

Super Mario Bros. is a very popular 2D platform game developed by Nintendo and released in the mid 1980s [12]. A public domain clone of the game, named *Infinite Mario Bros.* (IMB) [14] was later published by Markus Persson. IMB features the art assets and general game mechanics of *Super Mario Bros.* but differs in level construction. IMB is playable on the web, where the Java source code is also available.[1] While implementing most features of *Super Mario Bros.*, the standout feature of IMB is the automatic generation of levels. Every time a new game is started, levels

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

are randomly generated by traversing the level map and adding features according to certain heuristics.

The internal representation of levels in IMB 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 for this game [21, 19, 16, 24]. In the following we describe one possible approach.

The Probabilistic Multi-pass Generator (PMPG) was created by Ben Weber [21] as an entry for the level generation track of the Mario AI Championship [17]. The generator is 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 changes the basic structure of the level by changing the height 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 on the items created and placed. For example, the width of generated gaps is limited by the maximum number of blocks that the player can jump over, and the height of pipes is limited to ensure that the player can pass through.

3.8 Lab session: Level generator for *InfiniTux* (and *Infinite Mario*)

InfiniTux, 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* [26]. The game was created to replace IMB, and is used in research [13, 22, 25, 7, 2] and the Mario AI Championship [20, 21, 5]. Since the level generator for *InfiniTux* 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 tuned 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 that used for the Level Generation Track of the Platformer AI Competition [18], a successor to the Mario AI Championship that is based on *InfiniTux*. The software provides an interface that eases 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 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 generators 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 matrix 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 matrix to your level and communicates with the simulator, should implement the *LevelGenerator* interface:

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

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 [17].

3.9 Summary

Constructive methods are commonly used for generating dungeons and levels in roguelike games and certain platformers, because such methods run in predictable, often short time. One family of such methods is based on binary space partitioning: recursively subdivide an area into ever smaller units, and then construct a dungeon by connecting these units in order. Another family of methods is based on agents

that "dig out" a dungeon by traversing it in some way. While these methods originate in game development and might be seen as somewhat less principled, other methods for dungeon or level generation are applications of well-known computer-science techniques. Grammar-based methods, which are more extensively covered in Chapter 5, build dungeons by expanding from an axiom using production rules. Cellular automata are stochastic, iterative methods that can be used on their own or in combination with other methods to create smooth, organic-looking designs. Finally, several related methods work by going through a level in separate passes and adding content of different types according to simple rules with probabilities. Such methods have been used for the iconic roguelike platformer *Spelunky* and also for the Mario AI framework, but could easily be adapted to work for dungeons.

References

1. Adams, D.: Automatic generation of dungeons for computer games (2002). B.Sc. thesis, University of Sheffield, UK
2. Dahlskog, S., Togelius, J.: Patterns as objectives for level generation. In: Proceedings of the Second Workshop on Design Patterns in the 8th International Conference on the Foundations of Digital Games (2013)
3. Dormans, J.: Adventures in level design: generating missions and spaces for action adventure games. In: PCG'10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, pp. 1 –8. ACM (2010)
4. Johnson, L., Yannakakis, G.N., Togelius, J.: Cellular automata for real-time generation of infinite cave levels. In: Proceedings of the 2010 Workshop on Procedural Content Generation in Games (2010)
5. Karakovskiy, S., Togelius, J.: The Mario AI benchmark and competitions. IEEE Transactions on Computational Intelligence and AI in Games **4**(1), 55–67 (2012)
6. Kazemi, D.: URL http://tinysubversions.com/2009/09/spelunkys-procedural-space/
7. 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. IEEE (2012)
8. Van der Linden, R., Lopes, R., Bidarra, R.: Designing procedurally generated levels. In: Proceedings of the the 2nd AIIDE Workshop on Artificial Intelligence in the Game Design Process, pp. 41–47 (2013)
9. Van der Linden, R., Lopes, R., Bidarra, R.: Procedural generation of dungeons. IEEE Transactions on Computational Intelligence and AI in Games **6**(1), 78–89 (2014)
10. Make Games: URL http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky
11. Mawhorter, P., Mateas, M.: Procedural level generation using occupancy-regulated extension. In: IEEE Symposium on Computational Intelligence and Games (CIG), pp. 351 –358 (2010)
12. Nintendo Creative Department: (1985). Super Mario Bros., Nintendo
13. Ortega, J., Shaker, N., Togelius, J., Yannakakis, G.N.: Imitating human playing styles in Super Mario Bros. Entertainment Computing pp. 93–104 (2012)
14. Persson, M.: Infinite Mario Bros. URL http://www.mojang.com/notch/mario/
15. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific (1997)
16. Shaker, N., Nicolau, M., Yannakakis, G.N., Togelius, J., O'Neill, M.: Evolving levels for Super Mario Bros. using grammatical evolution. In: IEEE Conference on Computational Intelligence and Games (CIG), pp. 304–311 (2012)

17. Shaker, N., Togelius, J., Karakovskiy, S., Yannakakis, G.: Mario AI Championship. URL http://marioai.org/
18. Shaker, N., Togelius, J., Yannakakis, G.: Platformer AI Competition. URL http://platformerai.com/
19. 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). AAAI (2010)
20. 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)
21. 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)
22. Shaker, N., Yannakakis, G.N., Togelius, J., Nicolau, M., O'Neill, M.: Fusing visual and behavioral cues for modeling user experience in games. IEEE Transactions on Systems Man, and Cybernetics pp. 1519–1531 (2012)
23. Smith, G., Treanor, M., Whitehead, J., Mateas, M.: Rhythm-based level generation for 2D platformers. In: Proceedings of the 4th International Conference on Foundations of Digital Games, FDG 2009, pp. 175–182. ACM (2009)
24. 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. 131–140. Springer LNCS (2010)
25. 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)
26. SuperTux Development Team: SuperTux. URL http://supertux.lethargik.org/
27. Wild Card: (2013). URL http://www.dwarfquestgame.com/
28. Wolfram, S.: Cellular Automata and Complexity: Collected Papers, vol. 1. Addison-Wesley Reading (1994)
29. Yu, D., Hull, A.: (2009). Spelunky