Procedural generation of dungeons

Roland van der Linden, Ricardo Lopes and Rafael Bidarra

***Abstract*—The use of procedural content generation (PCG) techniques in game development has been mostly restricted to very specific types of game elements. PCG has been seldom deployed for generating entire game levels, a notable exception to this being dungeons, a specific type of game levels often encountered in adventure and role playing games. Due to their peculiar combination of pace, gameplay and game spaces, dungeon levels are among the most suited to showcase the benefits of PCG. This article surveys research on procedural methods to generate dungeon game levels. We summarize common practices, discuss pros and cons of different approaches, and identify a few promising challenges ahead. In general, what current procedural dungeon generation methods are missing is not performance, but more powerful, accurate and richer control over the generation process. Recent research results seem to indicate that gameplay-related criteria can provide such high-level control. However, this area is still in its infancy, and many research challenges are still lying ahead, *e.g.* improving the intuitiveness and accessibility of such methods for designers. We also observe that more research is needed into generic mechanisms for automating the generation of the actual dungeon geometric models. We conclude that the foundations for enabling gameplay-based control of dungeon-level generation are worth being researched, and that its promising results may be instrumental in bringing PCG into mainstream game development.**

***Index Terms*—Procedural content generation, procedural level generation, role playing games, gameplay semantics.**

I. Introduction

Procedural content generation (PCG) refers to the algorithmic creation of content. It allows content to be generated automatically, and can therefore greatly reduce the increasing workload of artists. Some procedural content generation methods are gradually becoming common practice in the game industry but are mostly confined to very specific contexts and game elements. For instance, SpeedTree [1] is becoming a standard middleware to procedurally generate trees, as demonstrated by its integration in games like Grand Theft Auto IV *(RockStar Games, 2008)*, Batman: Arkham Asylum *(Eidos Interactive, 2009)*, Battlefield 3 *(Electronic Arts, 2011)*, and many others. More complete PCG approaches (*e.g.* methods that generate complete game levels) exist, but mostly in the research domain.

The lack of commercial use of PCG techniques has most likely to do with their lack of control: designers, by giving away part of it to an algorithm, are often suspicious of the unpredictable nature of the results of an automatic generator. However, an increasing number of recent benefits can help establish PCG in mainstream game development. These benefits include: (i) the *rapid* generation of content that fulfills a designer's requirements [2], (ii) the possible *diversity* of generated content (even when using similar requirements), which may increase game replayability [3], [4], (iii) the amount of *time* and *money* that a designer/company can spare in their game development process [5], and (iv) the fact that PCG can provide a basis for games to automatically *adapt* to their players [6], [7].

Such advantages continue to motivate ongoing research on this increasingly active field. In this paper, we survey the current state of PCG for dungeons, a specific type of level for adventure and role playing games (RPG). Our focus on dungeon generation is justified by two important factors: the unique control challenges it raises and its close relationship with successful games.

We define adventure and RPG dungeon levels as labyrinthic environments, consisting mostly of inter-related challenges, rewards and puzzles, tightly paced in time and space to offer highly structured gameplay progressions. A close control over gameplay pacing is an aspect which sets dungeons apart from other types of levels. This notion of pacing and progression is sophisticated: although dungeon levels are open for free player exploration (more than *e.g.* platform levels), this exploration has a tight bond with the progression of challenges, rewards and puzzles, as desired by game designers (unlike *e.g.* sandbox levels, in open game worlds).

Since procedural generation is, in essence, automated design, dungeon generation shares the same challenges of manual dungeon design. Dungeons are unique due to the tighter bond between designing gameplay and game space. Unlike *e.g.* platform levels or race tracks, dungeons call for free exploration but with a strict control over gameplay experience, progression and pacing (unlike open worlds, where the player is more independent). For example, players may freely explore a dungeon, choosing their own path among different possible ones, but never encounter challenges that are impossible for their current skill level (since the space to back track is not as open as, for example, a sandbox city). Designing dungeons is thus a sophisticated exercise of emerging a complex game space from a predetermined desired gameplay, rather than the other way around. As such, dungeon generation is unique: it is more about achieving structured control over generation than about finding unexpected interesting generated results.

Typically dungeons can be classified in two types: for adventure games and for RPGs. Adventure games like The Legend Of Zelda *(Nintendo, 1986)* strictly follow our definitions above, as apparent in, *e.g.*, [8], [9]. Modern RPGs, like The Elder Scrolls V: Skyrim *(Bethesda Softworks, 2011)*, include more open dungeons, complex in many ways (*e.g.* exploration), but, when compared with adventure games, are less sophisticated in their strict control over gameplay

Roland van der Linden, Ricardo Lopes and Rafael Bidarra* are with the Computer Graphics and Visualization Group, Delft University of Technology, 2628 CD Delft, The Netherlands (*corresponding author: +31152784564, email: roland.vanderlinden@gmail.com, {r.lopes|r.bidarra}@tudelft.nl).

This work was partly supported by the Portuguese Foundation for Science and Technology under grant SFRH/BD/62463/2009.

experience, progression and pacing. Most current academic research is in line with dungeons for adventure games and, as such, the remainder of this survey will be focused on that scope.

Dungeons are intrinsically tied to the history of PCG, showcasing its potential in video games. Players of this type of adventure games and RPGs were introduced early on to the notion of procedural levels and already recognize its value. Rogue *(Troy and Wichman, 1980)*, The Elder Scrolls II: Daggerfall *(Bethesda Softworks, 1996)* and Diablo *(Blizzard Entertainment, 1998)* are some of the better known early examples of this relationship between PCG and dungeons. However, that original momentum is not quite apparent in more complex current adventure games and RPGs, which are still very successful genres. We reckon that this backlog can be largely overcome by the application of many current state-of-the-art research results surveyed here.

This survey aims at providing an overview of dungeon-generation methods, with a special focus on their main scientific challenge: how these methods can be controlled. The remainder of the paper is organized as follows; Section II gives a more detailed introduction to procedural dungeon generation methods and how they can be controlled. Sections III through VI elaborate on specific methods that are relevant for procedural dungeon generation, classified into Cellular Automata (III), Generative Grammars (IV), Genetic Algorithms (V), and Constraint-Based (VI). Section VII analyzes other related work which, although not directly related to dungeons, could improve dungeon generation. Section VIII contains an overview and discussion of the surveyed methods. Finally, Section IX provides concluding remarks.

II. Controlling procedural dungeon generation

In most adventure games and RPGs, dungeons basically consist of several rooms connected by hallways. While the term 'dungeon' originally refers to a labyrinth of prison cells, in games it may also refer to caves, caverns, or (abandoned) 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).

Typically, dungeon generation refers to the generation of the topology, geometry and gameplay-related objects of this type of levels. While generation of non-player-characters, decorations (and even ambient effects, like lightning and music) are themselves topics of PCG research, they fall outside the scope of this survey.

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.

Most surveyed research describes the method of constructing the representational model, but does not provide quite as much detail in their method for mapping the model to the actual geometry of the dungeon. Unfortunately, this step is often neglected by researchers in their publications. However, it is understandable, since this is typically a less sophisticated step than the generation of abstract representational models. Therefore this survey mostly focuses on the representational model and its generation.

A very important element in any procedural method is the *control* it provides over the output and its properties. We refer to control as 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 to which extent 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. The lack of proper parameters or the lack of understanding of what a parameter exactly does, almost always imply poor control over a PCG method, which may lead to undesirable results or even catastrophic failures.

In early PCG methods (*e.g.* Perlin noise [10]), the focus was more on how the methods worked and *what* could be generated. More recently, researchers became more concerned about *how* they can achieve the results they want. Meaningful parameters were introduced to help generate content towards a specific output. As PCG methods grew in complexity, and different PCG methods were combined to form more complex generation processes, more control was needed as well. As an example, PCG control has evolved towards more natural interaction between designer and machine, with the use of techniques like declarative modeling [11] or controllable agents [12]. More recently, PCG control has expanded towards an even more natural interaction, including the use of gameplay as a meaningful control parameter [13].

As discussed before, the tight bond between gameplay and game space is an essential aspect of dungeons. Therefore, control over PCG methods for dungeons should preferably facilitate such bond, by steering space generation from gameplay requirements. This can be done in more (*e.g.* parameters referring to the difficulty of a level) or less (*e.g.* parameters referring to topology) explicit ways. Therefore, in this survey, we are specially interested in discussing this dungeon-specific generation challenge: *gameplay-based control*, where PCG parameters are related to gameplay data.

III. Cellular Automata

One of the methods for procedural dungeon generation is cellular automata. This self-organizing structure consists of a grid of cells in any finite number of dimensions. Each cell has a reference to a set of cells that make up its neighborhood, and an initial state at zero time (t = 0). To calculate the state of a cell in the next generation (t + 1), a rule set is applied to the current state of the cell and the neighboring ones. After multiple generations, patterns may form in the grid, which are greatly dependent on the used rules and cell states. The representational model of a cellular automaton is a grid of cells

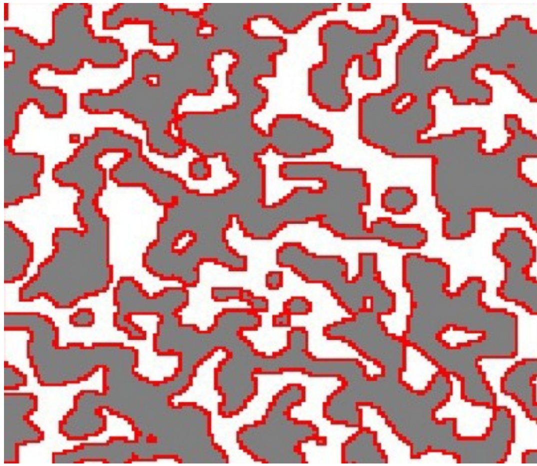

Fig. 1. A map generated with Cellular Automata. Grey areas represent floor, red areas, walls, and white areas, rocks [14].

and their states. An example set of allowed states would be {wall, path}, such that cells represent either a location where a player can go, or a location where the player cannot go.

Johnson *et al.* [14] use the self-organization capabilities of cellular automata to generate *cave* levels. They define the neighborhood of a cell as its eight surrounding cells (*Moore neighborhood*), where its possible states are {floor, rock, wall}. After an initial random cell conversion (floor to rock), the rule set is iteratively applied in multiple generations. This rule set states that: (i) a cell is rock if the neighborhood value is greater than or equal to T (*T=5*) and floor otherwise, and (ii) a rock cell that has a neighboring floor cell is a wall cell. Based on these rules, 'cave level'-like structures can be produced, as displayed in Fig. 1. This method allows real-time and infinite map generation.

As interesting features of Johnson *et al.*'s method, we can identify: (i) its efficiency (with the possibility of generating part of a level while the game is being played), (ii) the ability to generate infinite levels, (iii) the relatively straightforward creation algorithm (the used states and rules are simple), and (iv) the natural, chaotic feel that the levels created by this method have. Its main shortcomings are the lack of direct control of the generated maps and the fact this method only applies to 2D maps. The authors briefly discuss 3D generation, however the present control issues are likely to be worse in 3D. Additionally, connectivity between any two generated rooms (*i.e.* reachable areas) cannot be guaranteed by the algorithm alone, but has to be systematically checked for and added if non-existent.

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

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

The small number of parameters, and the fact that they are relatively intuitive is an asset of this approach. However, this is also one of the downsides of the method: it is hard 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, like a 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 carried out through a process of trial and error.

IV. Generative Grammars

Generative grammars were originally used to describe sets of linguistic phrases [15]. This method creates phrases through finite selection from a list of recursive transformational (or production) rules, which include words as terminal symbols. Based on generative grammars, other grammars have been developed, *e.g.* graph grammars [16] and shape grammars [17], [18]. These grammars only differ from the linguistic setup in that they use other terminal symbols (nodes and shapes) to allow graph and shape generation.

Graph grammars have been previously used by Adams [19] to generate dungeon levels. Although the author's work applies to first-person shooters (FPS), our definition of a dungeon level still applies directly to the generated content. This is clear in Adams' exclusive use of the term 'dungeon levels' throughout his work, even if only considering FPS. Rules of a graph grammar are used to generate topological descriptions of levels, in the form of a graph. Nodes represent rooms and edges its adjacencies. All further geometric details (*e.g.* room sizes) are excluded from this method. Most interestingly, graph generation can be controlled through the use of difficulty, fun and global size input parameters. Generating levels to match input parameters is achieved through a search algorithm which analyzes all results of a production rule at a given moment, and selects the most optimal one.

Adams' preliminary work is limited by the *ad-hoc* and hard-coded nature of its grammar rules and, especially, parameters. It is a sound approach for generating the topological description of a dungeon, but generalizing such method for games beyond the one used by the author would imply creating new input parameters and the rules that solve it. Regardless, Adams results showcase the early motivation and importance of controlling dungeon generation through gameplay.

Dormans [8], [20] uses 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. This mission is first abstracted to a network of nodes and edges, which is then used by a shape grammar to create a corresponding game space. In addition, the notion of 'keys' and 'locks' is a special feature, since together they are part of tasks in a mission, and allow players to navigate through the game space. Fig. 2(a) shows a mission network and Fig. 2(b) shows the space generated from the latter. The representational model of this method is a graph that represents the level connectivity by means of nodes and edges. A mission can also be seen as a very abstract representational model, although it comes down to a set of requirements or guidelines for the space.

The very clear integration between the motivation to generate space and the space generation itself very naturally ties in with

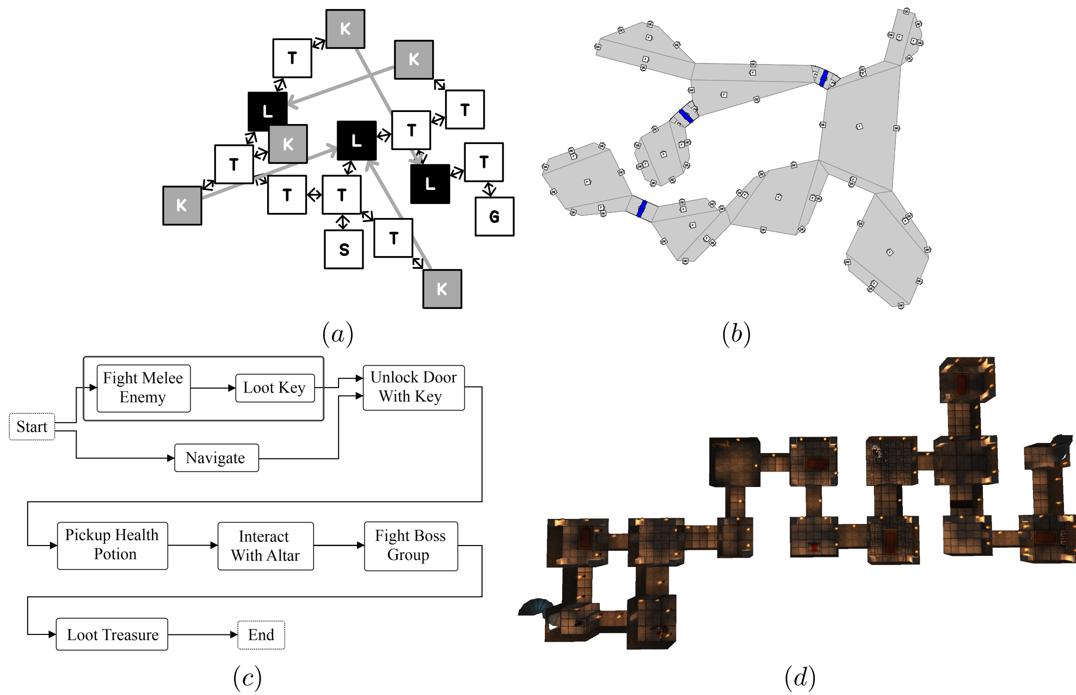

Fig. 2. (a) a mission with Tasks, Keys, and Locks [8]. (b) Mission structure from (a) mapped to a spatial construction [8] . (c) A gameplay graph generated in [9], (d) a dungeon layout, generated for (a)

the use of PCG in games. By considering the concept of missions, the PCG algorithm becomes more meaningful, and thus powerful, for both designers and players. Although this method allows versatile results, there is still a high complexity in setting up graph and shape grammars that suit specific needs, making it unclear whether this approach allows fully automatic generation for different domains. On the other hand, although there is no discussion on 3D dungeon generation, previous work on the use of shape grammars to generate city buildings [18] encourages future work on this direction.

Dormans does not directly use parameters in his approach, since control is exerted by the different rules in the graph and shape grammars. However, as with [18], it takes a lot of effort to understand and work with these grammars. On a positive note, the author was able to introduce gameplay-based control, most notably with the concept of a mission grammar. Constructing such a grammar, replacing it with a manually created mission, the direct specification of 'keys' and 'locks', or a mixed-initiative approach where a human designer can work with generated missions [20], are all interesting directions towards a richer gameplay-based PCG control.

Recently, Van der Linden *et al.* [9] use gameplay grammars, an approach similar to Dormans, to generate dungeons for a commercial game. Game designers specify, *a priori*, design constraints expressed in a gameplay-oriented vocabulary, consisting of player actions to perform in-game, their sequencing and composition, inter-relationships and associated content. These designer-authored so-called constraints directly result in a generative graph grammar, a *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. Fig. 2(c) and (d) show a graph and dungeon generated from this method.

The authors focus on improving gameplay-based control and, particularly, on its generic nature. Concerning the latter, this approach aims at empowering 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. The authors demonstrated such a specialized generator for only one case study, yielding fully-playable 3D dungeon levels for the game Dwarf Quest *(Wild Card, 2013)*.

As for gameplay-based control, this approach empowers designers to fully specify and control dungeon generation with a more natural design-oriented vocabulary. Designers can specify their own created player actions vocabulary and use it to control and author dungeon generators. As required in dungeon design (see Section I), 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 an even more fine-grained control over generated dungeons. However, the authors have not yet evaluated the intuitiveness of their approach with game

designers.

V. Genetic Algorithms

Genetic algorithms are search-based evolutionary algorithms that try to find an optimal solution to an optimization problem. In order to use a genetic algorithm, a genetic representation and a fitness function are required. The genetic representation is used to encode possible solutions into strings (called genes or chromosomes). The fitness function can measure the quality of these solutions. A genetic algorithm goes through an iterative process of calculating fitness, and then selecting and combining the best couple of strings from a population into new strings. An often used method of selection is to make the probability to combine a string with another proportional to their fitness: the higher their fitness, the more likely they will be used for a combination. The combinatory process itself is called crossover. In addition, a mutation process can randomly change a single character in a string with some small probability. Mutations ensure that given infinite time, the optimal solution will always be found.

Hartsook *et al.* [21] presented a technique for automatic generation of role playing game worlds based on a story. The story can be man-made or generated. They map story to game space by using a metaphor of islands and bridges, and capture both in a space tree. Islands are areas where plot points of the story occur. Bridges link islands together (although they are 'locations' and not 'roads'). A space tree represents the connectivity between islands and bridges, and also holds information on location types (also called environmental types). Hartsook *et al.* use a genetic algorithm to create space trees. Crossover and mutation deal with adding and deleting nodes and edges in the tree. The fitness function used by Hartsook *et al.* uses both evaluation of connected environmental types (based on a model that defines which environmental types often occur next to each other), and data about the play style of the player (to determine the correct length and number of branches). Settings include the size of the world, linearity of the world, likelihood of enemy encounters, and likelihood of finding treasures. The space tree that the genetic algorithm selects as the most optimal, after a fixed number of iterations, is then used to generate a 2D game world, where tree nodes are mapped to a grid using a recursive backtracking algorithm (depth-first). If there is no mapping solution, the space tree is discarded and a new space tree has to be constructed. Fig. 3(a) shows an example space tree mapped to a grid. This grid is then used as a basis for the positioning of locations in the game space (Fig. 3(b)).

As with Dormans, the very clear integration between the motivation for creating a space and the space itself offers advantages. The game world is not randomly pieced together: as long as the story makes sense, so will the game world generated for it.

So, in addition to the benefits of PCG stated in Section I, another advantage is that using the story to steer a procedurally generated game brings it a step closer to the contents of a conventional manually-created game. In game development, the story can be the motivation to create a specific game world, and this PCG method captures that principle. However, the stories presented by Hartsook *et al.* seem too simple, given that they are the base of the entire technique. While being a very relevant first step towards story-based PCG control, it is unclear whether the generation process would still be easy to control with a lot more story properties and options. Apart from the story, Hartsook *et al.* considered another form of gameplay-based PCG control: adding a player model as a parameter for the world creation leads to a game world directly suited for the players' needs.

While the story-based approach may allow some form of 3D mapping, the current work only focuses on 2D. Furthermore, based on the discarding rule (no mapping solutions means a new space tree needs to be constructed), performance could potentially be a limitation for this method.

Valtchanov and Brown [22] use a genetic algorithm to create 'dungeon crawler levels'. They use a tree structure to represent (partial) levels, which is also the genetic representation. Nodes in the tree represent rooms, and edges to children of the node represent connections to other rooms. See Fig. 3(c) for an example of the tree structure (genetic representation).

The fitness function has a strong preference for maps composed of small, tightly packed clusters of rooms which are inter-connected by hallway paths. Maps with up to three event rooms near the perimeter of the map are also highly favored by the fitness function (see Fig. 3(d)). During the generation process, the maps are directly created to test if rooms can be placed the way the tree represents it. If it is not the case, the corresponding branch of the tree is removed.

Interesting features of this method are the elegant and orderly placement of rooms, even allowing specific distance requirements for special 'event rooms'. Although results show that levels indeed converge to tightly packed clusters of rooms, it should be mentioned that this specific fitness function only allows chaotic placement of rooms, most of which may be redundant (in their role): there is no explicit *gameplay-oriented* motivation for their placement.

Ashlock *et al.* [23] investigate multiple methods to create maze-like levels with genetic algorithms. They explore four representations of mazes;

- Direct binary: gene with bits, representing a wall or accessible area;
- Direct colored: gene with letters that represent colors;
- Indirect positive: chromosome represents structures to be placed;
- Indirect negative: chromosome represents structures to be removed.

The authors' results show us that direct mazes are the most interesting. These are made up of a grid, where the operators of crossover and mutation are used to flip cells into becoming a wall or an accessible area. Additionally, a valuable technique was proposed to improve overall maze generation: the use of checkpoints, along which the maze generation can be guided and which can be used in the fitness function (checkpoints are represented in green in Fig. 4).

The authors detail five fitness functions that are tested on the genetic maze generation approach for the different representations. The most promising fitness functions are:

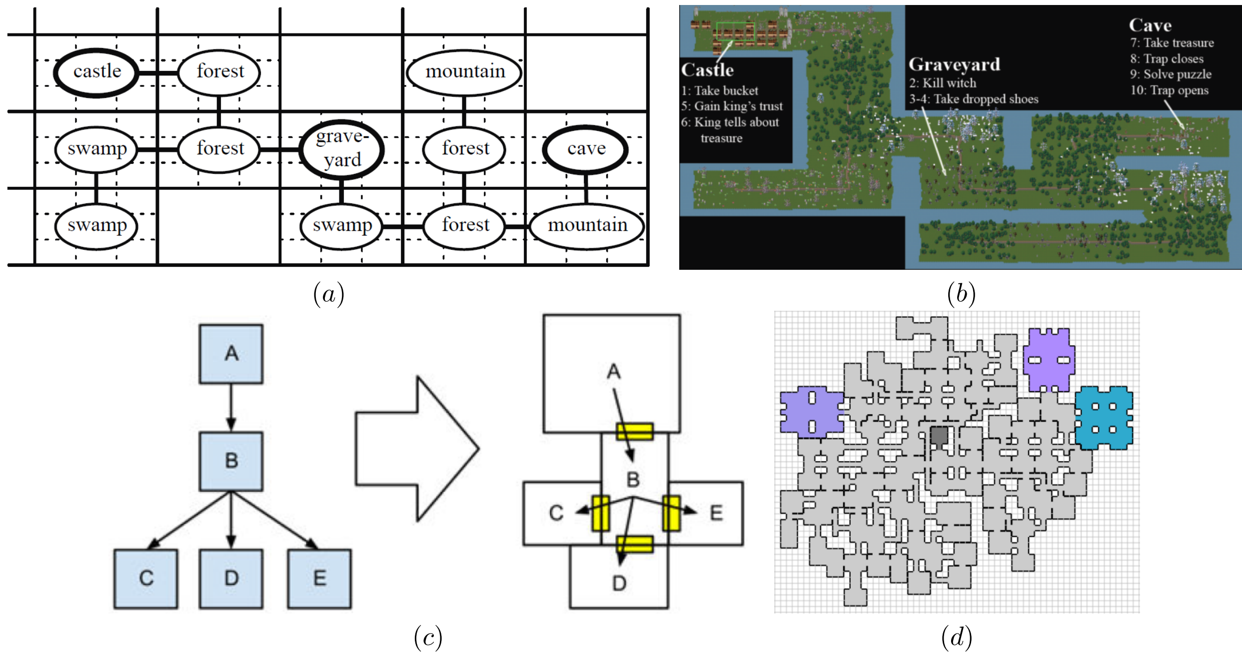

Fig. 3. (a) A space tree mapped to a grid; plot locations are bold [21]. (b) A game world generated from a space tree, similar to (a) [21]. (c) Translation from tree structure to map [22]. (d) Example of a generated map [22]. Event rooms highlighted.

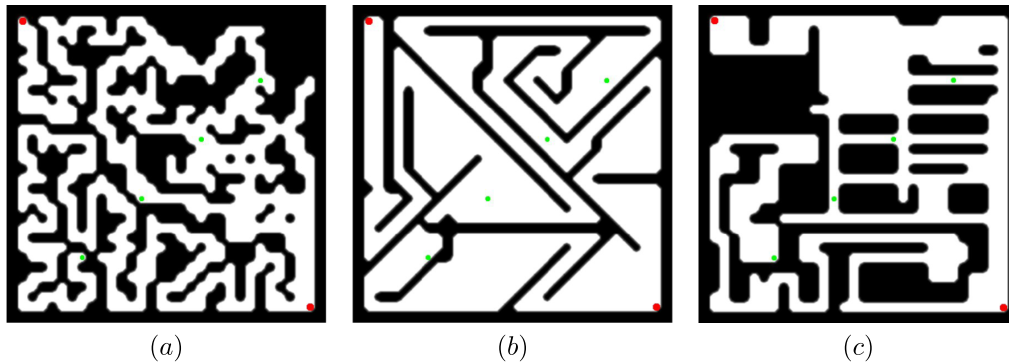

Fig. 4. Mazes generated by [23]. All mazes have been created with fitness function F2. (a) Direct binary representation. (b) Indirect positive representation. (c) Indirect negative representation. Green dots are checkpoints. Red dots are start and finish.

- F1 - maximize path length from entrance to exit;
- F2 - maximize accessibility of checkpoints;
- F3 - "encourage paths to branch, run over checkpoints, and then meet up again later".

The checkpoint based fitness functions [23] are extended by the authors [24], which can compose larger dungeon levels (exits on one tile must match entrances on adjacent ones). The new fitness function simply maximizes the sum, over all pairs of entrances, of the distances between the entrances. Thus, using tiles, entrances on each tile become checkpoints.

Fig. 4 displays several maps generated with fitness function F2. The results of this method can be very versatile. Both natural looking (chaotic) mazes and more structured mazes can be created, and connectivity is assured for both types. However, it appears to be hard to determine a fitness function that always does what you expect it to do, while creating different looking levels. For instance, the use of fitness function F3 allows multiple paths that branch and meet up again - but the length of the path from entrance to exit then suddenly becomes much shorter than with fitness function F1. Another drawback is that the generated levels seem very random.

Regarding control, Valtchanov and Brown and Ashlock *et al.* mostly use the parameters associated with genetic algorithms to steer the level generation process. In both cases the most important one is the fitness function, which evaluates the levels that fill the requirements the best way, relatively to other competing generated levels. Changing the fitness function causes entirely different levels to be generated. This becomes very clear from the work of Ashlock *et al.*, where they investigate different fitness functions. Other genetic-algorithm parameters are the mutation and crossover probabilities and the number of generations to be used.

The effect the parameters can have on the generated levels is quite high. A different fitness function may lead to different sorts of layout, for instance neatly divided over a specific space, or as thin-stretched as possible. However, finding and creating a suitable fitness function can be a hard task, especially for

designers with no background in programming or mathematics.

The genetic algorithm parameters mostly control how well (and how fast) the solution reaches an acceptable result based on the fitness, including the possibility of reaching local optima. The main problem with these genetic algorithm parameters is their apparent mismatch with backtracking: if a certain result is obtained and needs to be adjusted, it is unclear which parameters need to be altered.

It is encouraging to observe that gameplay can already have a small role in controlling these search-based PCG methods. Using special event rooms and checkpoints directly in the fitness functions, allows a more gameplay-oriented degree of control, which goes beyond the traditional pure topological approach. Events and checkpoints have a more explicit meaning in terms of gameplay (what happens in-game) than other topological constraints.

Sentient Sketchbook is a tool proposed by Liapis *et al.* [25], aiming at supporting designers during the creation of game levels. Game designers sketch low resolution maps, where tiles represent passable or impassable sections, player bases and resources. *Sentient Sketchbook* is able to add detail to the coarse map sketch and test maps for playability, navigable paths, game pace and player balance. Dungeons are one of the available possible results of automatically detailing coarse map sketches into final level maps. Tiles are converted into rooms, dead ends and corridors.

Additionally, this tool automatically suggests new alternatives for the map being sketched by a designer. These alternatives are automatically generated via genetic search algorithms performing constrained optimization, where the current maps' appearance is the starting point. Besides map diversity, fitness functions optimize game pace and player balance, by measuring resource safety, safe areas distribution, exploration difficulty and its relative balance between a player and its enemy. Via user studies, the authors concluded that industry experts identified map suggestions as an important feature, having selected a generated suggestion to replace the manual design in 92% of the sessions.

It is most interesting to verify that the generated dungeons are very close visually to human-designed levels. These results seem to demonstrate that a mixed-initiative method, where manual design and generation algorithms are integrated, is able to add such a quality to generated levels. However, it is still unclear how low-resolution sketching and genetic-based generation (steered by this sketching) would scale to larger and more detailed maps.

Sentient Sketchbook is very explicit in its gameplay-related criteria to generate suggestions. All the fitness functions directly maximize gameplay qualities, by measuring playability, pace and balance. Most interestingly, generation is steered by an initial state of the designer's sketch. We argue that, when manually designing such a dungeon sketch, designers are creating a game space for a predetermined desired gameplay experience (as mentioned in Section I). As such, this becomes an influential aspect in control, since that initial sketch directly maps from the gameplay concepts in a designer's mind. Such control is less automatic than the fitness functions of Valtchanov and Brown and Ashlock *et al.* but, nevertheless, goes beyond their specialized topological approach.

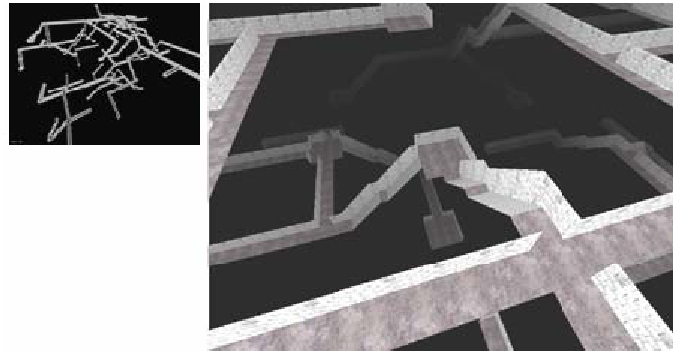

Fig. 5. Dungeon in 3D space generated based on constraints [26].

VI. Constraint-based

Roden and Parberry [26] proposed a pipeline for generating underground levels. They start by generating an undirected 3D graph which represents the level structure, where each node in the graph represents a portion of the actual level geometry. Constraints on the topology of the graph and node properties (distance, adjacencies) are used to steer generation of the graph. The generated 3D graph is then used to place geometry, using a set of prefabricated geometry sections. After these are interconnected, objects (such as furniture) are placed in the dungeon. An example of a generated dungeon is displayed in Fig 5.

To the best of our knowledge, this approach was the only one (*i.e.* until [9]) to describe a method that explicitly considered dungeon generation for 3D games. Generation is based on a constraint-solving approach where constraints are expressed as rules (with parameters as distance, for example) which place nodes in relation to fixed terminal nodes (entry/exits).

As for control, the authors use an initial graph topology and a set of constraints as the input for their level generation. They use standard graph topologies (like tree, ring, star), but also allow combinations of sub-graphs. Constraint parameters can be related to one or more fixed nodes, such as min/max distance, fixed connections to adjacent nodes, and the use of prefabricated geometry specific to a node. They can also specify that a given sequence of nodes can only be visited by the player in a specific order.

Interesting about this control approach is that such constraints can be defined for a level. This allows control in such a way that important high-level features can be first defined and then the rest can be generated around it. A drawback of this approach is the fact that these constraints do not capture gameplay data as effectively as missions or narratives. It is still up to the designer to not only create all the prefabs for level geometry sections and objects, but also to express a meaningful set of constraints for the generator. It is not straightforward for a designer to translate gameplay concepts (like pacing or difficulty) into topology, adjacency, etc.

VII. Other related approaches

In this section, we survey various related research results identified as relevant for the procedural generation of dungeons.

Unlike the previous sections, there is no single category or family to characterize these methods. Such methods are not full-fledged procedural generators of dungeons, since they are either very preliminary or they do not generate dungeons levels, as we defined them. Still, their inclusion in this survey is important since they could potentially contribute specific improvements, related and relevant for dungeon generation.

Togelius *et al.* [27] proposed a PCG approach to generate dungeons. However, this method is very preliminary since, as far as we know, there is no current evaluation yet for their claims. Simple 2D dungeons are generated using hybrid compositional PCG, where two distinct content generation methods are combined, one nested inside the other. The authors argue that distinct PCG methods have distinct advantages and disadvantages and that, through composition, their strengths are reinforced and weaknesses eliminated.

An experiment was performed using an Answer Set Programming (ASP) and Evolutionary Search hybrid approach. In essence, a constructive or solver-based algorithm (ASP) is used as the genotype-to-phenotype mapping of an evolutionary search-based algorithm. The advantage is that, with a nested composition of these PCG methods, the ”inner” algorithm (ASP) could better guarantee a low-level of desirable properties (well-formedness, playability and winnability of dungeons), while the ”outer” algorithm can pursue higher-lever desirable properties (optimizing challenge and skill differentiation).

This research raises interesting questions. By having two different methods, nested and responsible for different desired properties, control over generation could improve. In theory, by having compositions of generators, composition of the respective parameters is also possible. Since ”more is more”, this could mean a finer-grained level of control. However, the authors did not compare their results (generated dungeons, used as an example) with any other method. It remains to be seen if their claims of improving the control over the desirable properties of generated content are correct.

An interesting research direction is proposed by Smith *et al.* [28]. Although the authors generate 2D platform levels, we see their method as highly applicable to dungeons. Levels are generated 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.

Although this approach only creates platform levels, it ties in well with dungeon generation. 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 exactly defined by Smith *et al.* is not applicable for 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, *e.g.* pacing.

For more control, Smith *et al.* have a set of ”knobs” that a designer can manipulate, such as: (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. The critics also have a set of parameters to provide control over the rhythm length, density, beat type, and beat pattern. Overall, the large amount of parameters for different levels of abstraction provide a lot of control options, and allow for the versatile generation of very disparate levels. They relate quite seamlessly to gameplay (specially in its platform genre), although achieved at a lower level than, for example, the missions of Dormans.

Although occupancy-regulated extension (ORE) was proposed by Mawhorter and Mateas [29] to procedurally generate 2D platform levels, their method seems very interesting to apply to dungeon generation, without needing many adjustments.

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.* [28] because rhythm groups are separately generated by a PCG method whilst the 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 needs to be 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.

A mixed-initiative approach is proposed for this ORE method, 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 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 have 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 out of their hands. Additionally, it would fit to 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, dungeon levels (especially in 3D) are generally required to be more complex than 2D platform levels that have a lot of similar ground geometry.

Potential player locations are used as a basis for chunk placement to ensure playability. The chunks themselves can still cause unplayable levels, though. For example, if chunks

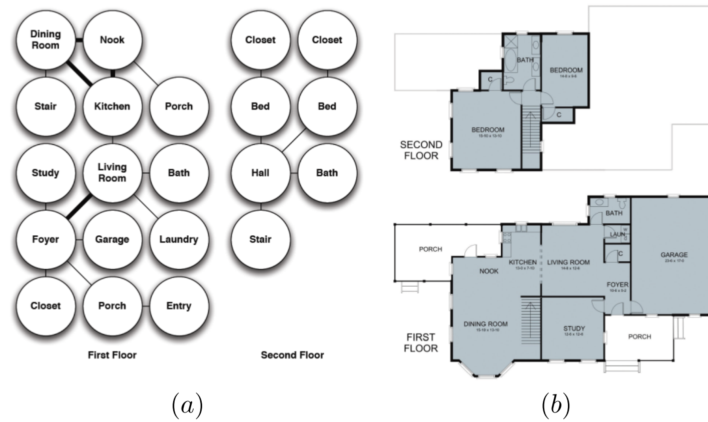

Fig. 6. Results from [30]. (a) An architectural program. (b) A set of floor plans based on (a).

without ground geometry are positioned next to each other, there is no place for the player to go, making a level unplayable. Their 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 an interesting decently-sized chunk library is used.

Mawhorter and Mateas do not mention specific parameters for their ORE algorithm. However, a designer still has a lot of control. Besides the mixed-initiative approach, the chunks in the library and their probability of occurrence are implicit parameters (*i.e.* they determine the level geometry and versatility), and possible player actions need to be defined and incorporated in the design of chunks. The mixed-initiative is still the biggest amount of control one can have, even from a gameplay-based perspective. However, this approach can become at times too similar to manually constructing a level, decreasing the benefits of PCG. In summary, a designer has the potential to have a lot of control over the level generation process, but the available control might not be very efficient. It seems that, at this point, a lot of manual work is still required for specific levels to be generated.

Another technique which can be suitable for dungeon generation is proposed by Merrell *et al.* [30]. They apply machine learning techniques to generate 3D residential building layouts, with a focus on floor plans, *i.e.* the internal organization of spaces within the building.

Building requirements include: (i) a list of the different rooms, (ii) their adjacencies, and (iii) their desired sizes; see Fig 6(a). These are the input to generate architectural programs by means of a Bayesian network that has been trained on real-world data, *i.e.* from actually existing residential buildings. Such data includes, for example, rooms that are often adjacent, and whether the adjacency is open, or mediated by a door. PCG methods based on real-world data are a recent advancement in the academic community.

From the architectural programs, sets of floor plans are generated using stochastic optimization, where the Metropolis algorithm is used to optimize the space. An example of such floor plans is displayed in Fig 6(b). The floor plans are then used to construct the 3D model of the building. Different styles can be applied that define the type of geometric and material properties of building elements, as well as the spacing of windows, roof angle, etc.

In terms of control, Merrell *et al.* take a set of high-level requirements as the input to their residential building generation. Input options can be defined in a flexible manner and include the number of bedrooms, the number of bathrooms, approximate square footage, and (after the layout has been generated) the style of the building. The 'residential' type of building always has some similar structure, and the authors identified only a few high-level parameters to generate a realistic 3D model of a residential building. Still, although controlling the generation process can be very clear and easy, this strength can also be a disadvantage. This technique can generate 'standard' residential buildings, but any desired deviation from that type of building seems hard to control, since everything depends on the floor-plan algorithm.

Of all surveyed non-dungeon methods, this is perhaps the most interesting one, in terms of application. First of all, a dungeon map can behave like a floor plan, using the same definition as before: the internal organization of spaces, in this case, within the dungeon. The proposed method could generate dungeon floor plans, by considering different types of rooms and constraints, and by using different 3D models. In that case, dungeon level data from existing games could be used to train a Bayesian network on the adjacencies or connections between different rooms in dungeons, and even be used to position rooms in a the dungeon layout. Generated levels would then mimic the best practices in dungeon level design. As with the previously analyzed methods, the use of a step-wise approach (*i.e.* connecting requirements together, creating a basic floorplan, and then creating the geometry with different possible styles) is also applicable to dungeons.

As mentioned above for buildings, the technique only seems appropriate to define dungeon types of similar structure. If dungeons in all kinds of shapes need to be generated, this approach seems to be a bit too restrictive (unless the real-world design data is large enough, and the Bayesian network can deal with it).

TABLE I
OVERVIEW OF SURVEYED METHODS AND THEIR PROPERTIES

		Properties				
Category	**Reference**	**Approach**	**Control provided**	**Gameplay features**	**Output Types**	**Variety of results**
Cellular Automata	[14]	Grid cells state modification	Initial state; # of generations	-	2D dungeon with floor, rock, wall cells	Chaotic maps: little variation
Generative Grammar	[19]	Graph grammars	Difficulty, size, fun	Difficulty, fun parameters	2D room graph	Dependent on hard-coded graph grammar
	[8], [20]	Graph and shape grammars	Input missions	Missions	2D dungeon with rooms, locks, keys	Maps as versatile as missions
	[9]	Gameplay grammar and 3D geometry generator	Customizable grammars, parameter-based rule selection	Player actions as nodes, gameplay parameters for rule re-writing	3D dungeon	As versatile as grammars and parameters
Genetic Algorithms	[21]	Space tree mutation	Game story, fitness function, player model	Game story	2D dungeon-like game world	Combination of premade location types
	[22]	Tree mutation	Fitness function, genetic parameters	Special event rooms	2D dungeon	Tightly packed rooms connected by hallways
	[23], [24]	Combining 4 genetic representations and 5 fitness functions	(same as above)	-	2D maze	Larger between combinations
	[25]	Mixed-initiative: map sketch and constrained optimization	User sketching and fitness function	Playability, pace, balance	2D dungeon	Large, even beyond dungeons
Constraint-based	[26]	Constrained graph generation	Topology, node placement	-	3D underground level	Small rooms connected by hallways
			Other related approaches			
Hybrid Approach	[27]	Compositional PCG: ASP and evolutionary search	ASP constraints, fitness functions	-	2D dungeon	-
Generative Grammar	[28]	Rhythm-to-actions-to-geometry grammars	Path, rhythm, objects, critics	Player-based rhythm	2D platformer level	Combination of premade level segments
Occupancy Regulated Extension	[29]	Position-based combination of level chunks	Chunk library, mixed-initiative	Chunks contain game-related items	2D platformer level	Combination of premade level chunks
Real-World Data	[30]	Bayesian network trained with real data	# of rooms, sizes, distances, style of the building	-	3D residential building layout	Varied residential building models

VIII. DISCUSSION

As surveyed in the previous sections, there are many approaches that can be applied to procedural dungeon generation. Even for a specific domain as dungeon levels, no single approach stands out to generate all possible types of dungeons. The existence of such a variety of methods demonstrates that dungeon generation typically involves multiple distinct requirements. Table 1 provides an overview of the methods we surveyed. The variety on dungeon generation strategies can be observed in the columns for the *Approach*, *Output types* and *Variety of results*.

Efficiently generating dungeons on-line, as the player advances through them, is still a largely unexplored problem. In reality, the nature of a dungeon might actually discourage this: a dungeon typically works as a (narrative) progression towards a *fixed* final goal at its end (*e.g.* a boss, important item or exit in its final room). However, we believe there are still interesting open research questions in on-line dungeon generation, including for example, generating the intermediate spaces between some entry and exit spaces, as the player advances through the dungeon.

From Table 1, we can identify the less investigated challenge in dungeon generation: 3D content, which is considered in just two proposals [9], [26]. Because they depend on a particular game, both methods are *ad-hoc* solutions, focused on mapping their representational model to a 3D geometric level. Still, the results of these methods indicate that 3D-generated dungeons are still far from designer-made dungeons. An interesting future direction to improve upon this limitation is to consider gameplay-based control not only for the representational model generation [9], but also for the geometric level creation, including *e.g.* decorations or ambiance effects.

Apart from on-line generation and 3D content, which remain open challenges, we can identify some useful common practices in procedural dungeon generation. For example, a stepwise approach is adopted by Dormans [8], [20], Hartsook *et al.* [21], and Merrel *et al.* [30]. A high level input is gradually transformed into a refined result, where each step increases the amount of details in the model, possibly using intermediate representational models. This strategy also favors a mixed-initiative approach, in which designer and machine work alongside each other in the generation process. The gradual steps allow designers to include specific details they want at different levels of abstraction, without the need to specify the rest of the details. Even better, they could allow the designer to act on only one of the levels of abstraction, typically the highest level, leaving the remaining steps to an automatic generator. A good example of that is the approach by Mawhorter and Mateas [29], which allows designers to place geometry chunks in a 2D platformer level, and then lets the generation process take care of filling up the rest of the level.

From Table 1, we can also gain interesting insights regarding current features for controlling the generation process (columns *Control provided* and *Gameplay features*). In general, most control parameters are rather specific to the technique used, and it is often non-intuitive what adjusting a parameter will mean

for the generated model. Cellular automata and constraint-based approaches use a straightforward control mechanism over their algorithm's low level parameters, thus requiring a full understanding of the generation process. For designers to control them in a meaningfully way, they always need to explicitly map that understanding to gameplay-related goals (*e.g.* how does room adjacency better support the designed mission).

As for genetic algorithms, the variety of generated maps greatly depends on the fitness function. Fitness functions do control the generation process, but understanding which fitness function to use requires considerable knowledge about the entire generation process: it is not very intuitive, specially from a gameplay perspective. Liapis *et al.* [25] have made some significant contributions in this regard by not only proposing more gameplay-based fitness functions, but also allowing for a mixed-initiative approach, based on designer sketches. The genetic algorithm by Hartsook *et al.* [21] also allows a mixed-initiative approach, in which a story guides the initial map generation process, so that the fitness function is not the exclusive control mechanism. This gameplay-based layer of story control is added on top of the fitness function, allowing for a more meaningful control of generation.

The story-based approach of Hartsook *et al.* [21], the mission-based method of Dormans [8] and the gameplay-grammar approach of van der Linden *et al.* [9] are closer to providing a richer gameplay-based PCG control. By encoding and capturing the mapping between gameplay data and space generation, they (i) relieve designers from having to regularly deal with that mapping, (ii) offer a control vocabulary closer to the designers' way of expressing their intent, and (iii) provide an intuitive basis for generating more meaningful and diverse content.

Further extending this type of control still remains an interesting challenge. Beyond missions, player actions and story, further gameplay-based control methods could include, for example, detailed data on the player performance or skills, an important gameplay concept in adventure games and RPGs. The more PCG becomes dependent on gameplay, the more apparent it becomes the need for encoding some mapping between content and gameplay data in a generic way. One such encoding mechanism is the use of object semantics, *i.e.* additional information about game content, beyond its geometry. For dungeons, richer gameplay-related semantics could help support more powerful PCG methods. This was, for example, combined with the gameplay grammar approach mentioned above [9], where player actions are associated with the type of content that can enable their execution. Still, it would be very interesting to investigate its reverse, *i.e.* enriching game content with the knowledge on which player actions it can enable. This would likely help in generating even more meaningful dungeons. Such gameplay semantics has previously been applied to the procedural generation of game worlds [13], [31]. Another good example of the use of object semantics is the work of Merrel *et al.* [30], where known room functions and their interrelations are used, leading to more meaningful floor layouts being generated.

IX. Conclusions

Dungeon levels for adventure games and RPGs are probably among the few types of game worlds that have been generated very successfully in the past by applying PCG methods. However, procedurally generating a dungeon is a large and complex problem, and each of its stages has multiple possible solutions.

In this paper, we surveyed research on a variety of PCG methods that are suitable for procedural dungeon generation. From the analyzed papers, we conclude that a variety of different PCG methods and dungeon types can already help in achieving some designer's requirements. These methods are fast enough, in the sense that their procedural generation is faster than creating the dungeon content manually, thus leading to spare time and money in the game development process. As discussed in Section VIII, real-time performance is not essential and, as such, dungeon generation can benefit from the advantages of all PCG methods that are not appropriate for on-line generation.

We found that there is a wide diversity in the generated results. This happens not only for similar requirements within a certain method, but mostly among different PCG methods, which lead to a wide array of dungeon types and topologies. We also observe that gameplay-based control is now being successfully investigated, *e.g.* using missions, stories, player actions or even player models to control PCG. This, in turn, provides a significant basis for games to automatically adapt to their players.

We conclude that the most promising challenges lie in the generation of complete dungeons for 3D games, preferably with extensive gameplay-based control. As discussed in Section VIII, current achievements show that these research goals are possible, although there are still quite some open challenges ahead. We believe that dungeon generation is a privileged PCG domain able to yield substantial results with a moderate research investment, eventually contributing to bring PCG into mainstream game development.

Acknowledgments

We thank Joris Dormans and Elmar Eisemann for various exciting discussions on what is a dungeon and why should you care. We also thank the anonymous reviewers of this article for their insightful comments.

References

[1] Interactive Data Visualization, Inc. , "SpeedTree." [Online]. Available: http://www.speedtree.com/

[2] A. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187 –200, 2011.

[3] E. Hastings, R. Guha, and K. Stanley, "Automatic content generation in the Galactic Arms Race video game," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 245–263, 2009.

[4] G. Smith, E. Gan, A. Othenin-Girard, and J. Whitehead, "PCG-based game design: enabling new play experiences through procedural content generation," in *Second International Workshop on Procedural Content Generation in Games (PCG 2011)*, Bordeaux, France, June 28, 2011 2011.

[5] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker, "The role of semantics in games and simulations," *ACM Computers in Entertainment*, vol. 6, pp. 1–35, 2008.
[6] R. Lopes and R. Bidarra, "Adaptivity challenges in games and simulations: a survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 2, pp. 85 –99, june 2011.
[7] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 99, pp. 147–161, 2011.
[8] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, pp. 1:1–1:8.
[9] R. van der Linden, R. Lopes, and R. Bidarra, "Designing procedurally generated levels," in *Proceedings of the the second workshop on Artificial Intelligence in the Game Design Process.*, 2013.
[10] K. Perlin, "An image synthesizer," in *SIGGRAPH '85: Proceedings of the* 12th *Annual Conference on Computer Graphics and Interactive Techniques*, vol. 19, no. 3. San Francisco, CA, USA: ACM, July 1985, pp. 287–296.
[11] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352–363, April 2011.
[12] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, June 2010.
[13] R. Lopes, T. Tutenel, and R. Bidarra, "Using gameplay semantics to procedurally generate player-matching game worlds," in *PCG '12: Proceedings of the 2012 Workshop on Procedural Content Generation in Games*. Raleigh, North Carolina, USA: ACM, 2012.
[14] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *PCG '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. New York, NY, USA: ACM, 2010, pp. 10:1–10:4.
[15] N. Chomsky, *Language and Mind*. Harcourt Brace & World, Inc., 1968.
[16] G. Rozenberg, Ed., *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997.
[17] G. Stiny and J. Gips, "Shape grammars and the generative specification of painting and sculpture," in *Proceedings of the Workshop on Generalisation and Multiple Representation*, 1971.
[18] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool, "Procedural modeling of buildings," in *SIGGRAPH '06: Proceedings of the* 33rd *Annual Conference on Computer Graphics and Interactive Techniques*. Boston, MA, USA: ACM, July-August 2006, pp. 614–623.
[19] D. Adams, "Automatic generation of dungeons for computer games," 2002, B.Sc. thesis, University of Sheffield, UK.
[20] J. Dormans, "Level design as model transformation: a strategy for automated content generation," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. New York, NY, USA: ACM, 2011, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2000919.2000921
[21] K. Hartsook, A. Zook, S. Das, and M. Riedl, "Toward supporting stories with procedurally generated game worlds," in *IEEE Conference on Computational Intelligence and Games (CIG)*, September 2011, pp. 297 –304.
[22] V. Valtchanov and J. A. Brown, "Evolving dungeon crawler levels with relative placement," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, ser. C3S2E '12. New York, NY, USA: ACM, 2012, pp. 27–35.
[23] D. Ashlock, C. Lee, and C. McGuinness, "Search-based procedural generation of maze-like levels," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 260 –273, September 2011.
[24] C. McGuinness and D. Ashlock, "Decomposing the level generation problem with tiles," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 2011, pp. 849–856.
[25] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient sketchbook: Computer-aided game level authoring," in *Proceedings of ACM Conference on Foundations of Digital Games*, 2013.
[26] T. Roden and I. Parberry, "From artistry to automation: a structured methodology for procedural content creation," in *Proceedings of the 3rd International Conference on Entertainment Computing*, Eindhoven, The Netherlands, September 2004, pp. 151–156.
[27] J. Togelius, T. Justinussen, and A. Hartzen, "Compositional procedural content generation," in *PCG '12: Proceedings of the 2012 Workshop on Procedural Content Generation in Games*, Raleigh, NC, USA, May 2012.
[28] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2D platformers," in *Proceedings of the 4th International Conference on Foundations of Digital Games, FDG 2009*. ACM, April 2009, pp. 175–182.
[29] P. Mawhorter and M. Mateas, "Procedural level generation using occupancy-regulated extension," in *IEEE Symposium on Computational Intelligence and Games (CIG)*, August 2010, pp. 351 –358.
[30] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," *ACM Transactions on Graphics*, vol. 29, no. 5, pp. 181:1–181:12, 2010.
[31] J. Kessing, T. Tutenel, and R. Bidarra, "Designing semantic game worlds," in *PCG '12: Proceedings of the 2012 Workshop on Procedural Content Generation in Games*, Raleigh, NC, USA, May 2012.

Roland van der Linden received his B.Sc. and M.Sc. degrees in computer science from Delft University of Technology, The Netherlands. His M.Sc. dissertation topic was "Designing procedurally generated levels."

His current research interests include: procedural generation techniques for game levels, and data visualization.

Ricardo Lopes received the B.Sc. and M.Sc. degrees in information systems and computer engineering from the Technical University of Lisbon, Lisbon, Portugal, in 2007 and 2009, respectively. He is a Ph.D. candidate at Delft University of Technology, The Netherlands, and his research topic is "Semantics for the generation of adaptive game worlds."

His current research interests include adaptivity in games, player modelling, interpretation mechanisms for in-game data, and (online) procedural generation techniques.

Rafael Bidarra graduated in 1987 in electronics engineering at the University of Coimbra, Portugal, and received his Ph.D. in 1999 in computer science from Delft University of Technology, The Netherlands.

He is currently associate professor Game Technology at the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology. He leads the research lab on game technology at the Computer Graphics and Visualization group. His current research interests include procedural and semantic modeling techniques for the specification and generation of both virtual worlds and gameplay; serious gaming; game adaptivity and interpretation mechanisms for in-game data. Rafael has numerous publications in international journals and conference proceedings, integrates the editorial board of several journals, and has served in many conference program committees.