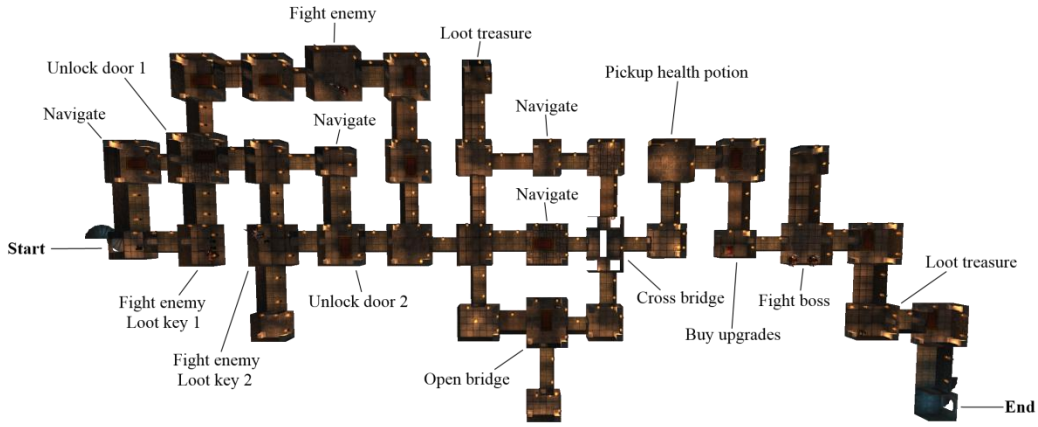
Designing Procedurally Generated Levels

Master's Thesis



Roland van der Linden

Designing Procedurally Generated Levels

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Roland van der Linden
born in Rotterdam, the Netherlands

Computer Graphics and Visualization Group
Intelligent Systems Department
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover figure: A procedurally generated dungeon in Dwarf Quest, indicating which actions the player can perform. The procedures to create this dungeon are visualized in Appendix B.

Designing Procedurally Generated Levels

Author: Roland van der Linden
Student id: 1361066
Email: roland.vanderlinden@gmail.com

Abstract

We aim to improve on the design of procedurally generated game levels. We propose a method which empowers game designers to author and control level generators, by expressing gameplay-related design constraints. Following a survey conducted on recent procedural level generation methods, we argue that gameplay-based control is currently the most natural control mechanism available for generative methods. Our method uses graph grammars, the result of the designer-expressed constraints, to generate sequences of desired player actions. These action graphs are used as the basis for the spatial structure and content of game levels; they guide the layout process and indicate the required content related to such actions. We showcase our approach with a case study on a 3D dungeon crawler game. Results allow us to conclude that our control mechanisms are both expressive and powerful, effectively supporting designers to procedurally generate game levels.

Thesis Committee:

Chair:	Prof. Dr. Ir. E. Eisemann, Faculty EEMCS, TU Delft
Supervisor:	Dr. Ir. R. Bidarra, Faculty EEMCS, TU Delft
Committee Member:	Dr. Ir. R. Heusdens, Faculty EEMCS, TU Delft
Committee Member:	Ir. R. Lopes, Faculty EEMCS, TU Delft

Preface

The research presented in this thesis is the final step in obtaining my *Master of Science* degree in *Computer Science: Media and Knowledge Engineering* at Delft University of Technology. It is the result of a nine month project at the *Intelligent Systems* department, and originated from an assignment focused on procedural content generation in the game 'The Elder Scrolls V: Skyrim'. This assignment evolved over time, becoming more abstract in nature and discarding the focus on Skyrim entirely. Such rigorous intermediate changes may seem bothersome, but they should be embraced as progress.

> "*Success consists of going from failure to failure without loss of enthusiasm*"
> - Winston Churchill

Even though failure is generally mocked and slated, we are hardwired to make assumptions based on incomplete data (*e.g.* I can eat this rectangular object), test hypotheses (*e.g.* take a bite), evaluate results (*e.g.* eating the object did not succeed), and adapt to our interpretation of those results (*e.g.* don't try to eat smartphones!). This model centers around failure as a learning mechanism; the only way to prohibit failure would be to stop testing hypotheses. A failed hypothesis should perhaps be considered more valuable than a successful one, since the notion of wrong assumptions stimulates us to investigate further, whereas supposed correct assumptions may leave us, cold beer in hand, satisfied.

I would like to thank everyone who supported me throughout my six years of failures. First and foremost, my fiancée Joyce Waarts, who inspires me to do good, and perhaps also for the spend-the-rest-of-our-lives-together-*thingy*, and my parents, whose continuous financial and moral support was invaluable. Furthermore, I would like to thank my family, friends, fellow students and department members for their interest and good times in general. A special thanks goes out to Ricardo Lopes, my daily supervisor, who helped shape the project and the papers presented in this thesis. Finally, I would like to thank Rafa Bidarra for his guidance, and Elmar Eisemann and Richard Heusdens for taking the time to evaluate my work.

Roland van der Linden
Delft, the Netherlands
August 20, 2013

Contents

Chapter 1

Introduction

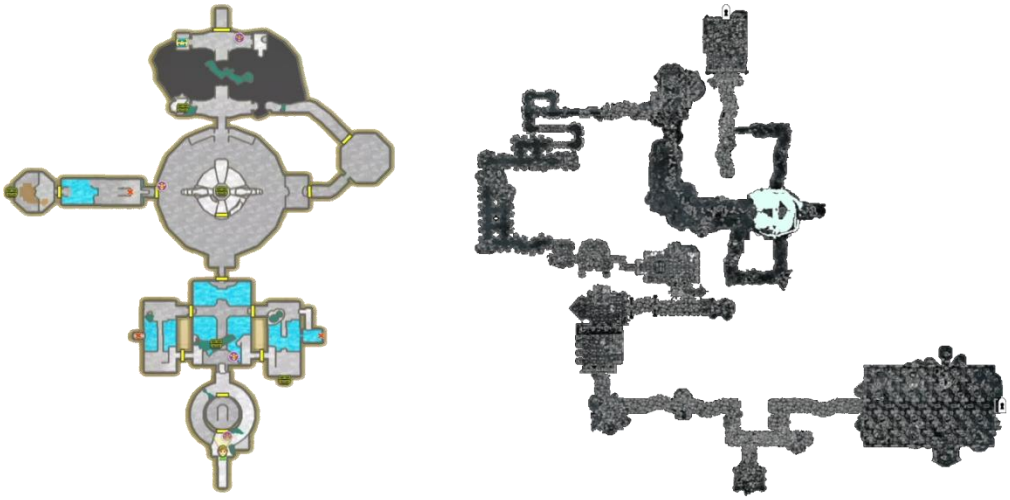
Levels in video games are virtual spaces in which the player character can maneuver and interact. For many game genres (*e.g.* platform, action, adventure, shooter), a level also refers to the precise composition of these spaces in combination with the content they contain (*e.g.* coins, weapons, enemies). This synthesis of spaces and content typically holds the distinctive sets of challenges players experience. Game levels are created by game designers, who are especially concerned with the coherence of (i) the spaces, (ii) the spaces in combination with content, and (iii) how this distributes, orders, and introduces such player challenges. Designing levels is thus a difficult process, which often requires many manual iterations. As such, the design of interesting and diverse, yet coherent, game levels is a time-consuming task.

Procedural content generation (PCG) is the research field that concerns itself with the algorithmic creation of virtual content. Commercially, it has recently been applied in games such as (i) *Battlefield 3* [EA 11], which uses *SpeedTree* [IDV 09] to generate and place vegetation, (ii) the *Borderlands* series [2KG 12], which use procedural generation to produce the in-game weapons, (iii) the *Left 4 Dead* series [Valve 09], which generate enemy and pickups spawn positions, (iv) the *Diablo* series [Blizzard 12], which has static locations but generates parts of the route and challenges in between them, and (v) *Minecraft* [Mojang 09], which generates entire virtual worlds for the player to explore. These examples and many others, show that PCG in commercial games is often applied to minor features like aesthetics, rather than to produce entire game levels. Minecraft is an exception to this principle, however its levels can be considered more random than any of the other mentioned games.

An important mechanism of any PCG technique is the *control* a user has over the results. Control determines what a generation algorithm can and cannot design, and it is required to be intuitive and reliable, providing results the user envisioned (such as in SketchaWorld [Smelik 11]). Control over procedural level generation could be improved, because as we observe (i) procedural level generation has hardly been used commercially, which we argue is due to incomplete control, and (ii) procedurally generated levels rarely seem as interesting as their manually designed counterpart.

1.1 Research Aim

For this thesis, we are concerned with improving the design of procedurally generated levels, *i.e.* improving the control over generators. We argue that generation procedures can be steered by gameplay requirements, due to the tight relationship between



(a) *(b)*

Fig. 1. Examples of dungeon topology. (a) 'Skyview Temple' from The Legend of Zelda: Skyward Sword [Nintendo 11]. (b) 'Bleak Falls Barrow' from The Elder Scrolls V: Skyrim [Bethesda 11].

gameplay and required game level content. Therefore, the research aim of this thesis is to *improve on gameplay-based control over procedurally generated levels*. We aim to do this by providing similar vocabularies and concepts game designers regularly apply in manual game design. As such, we propose the use of a gameplay-based vocabulary for control over game level features.

We focus on generating entire game levels, *i.e.* both the spaces in which the player can maneuver, and the content with which he can interact. The set of game levels is incredibly large and diverse, typically presenting distinct PCG challenges. In this thesis, we narrow our focus on *dungeons*, a type of level often encountered in role-playing-games (RPG). Dungeons are typically enclosed space structures, composed of rooms connected by hallways, both of which are filled with challenges (*e.g.* enemies, traps, puzzles) preceding rewards (*e.g.* treasure, equipment). Fig. 1 displays examples of dungeon layouts from (a) *The Legend of Zelda: Skyward Sword* [Nintendo 11], and (b) *The Elder Scrolls V: Skyrim* [Bethesda 11]. Both consist of rooms connected by hallways, regardless of the dungeon type; *temple* and *tomb* respectively. Dungeons are of particular interest due to (i) their dependency on player activity, which suits our gameplay focus, and (ii) the fact that they typically contain high amounts of gameplay-related content, as opposed to open world game levels such as cities or wastelands.

To validate and further specify our research aim, we investigate recent related work on procedural dungeon generation. This confirms that gameplay-based control yields promising results, *i.e.* a natural way to control the generation process. However, we identified opportunities to improve on existing research, for instance the use of player actions as a control mechanism. Player actions are the decisions and interactions available to the player during the game. We consider them to be an intuitive and natural control mechanism, since they are conceptually precisely what a player can do in a level, and they inherently indicate the required content and spaces of the level. For

example, the action *Fight Wizard* requires at least the content *wizard* and the space *wizard's tower*.

To improve on gameplay-based control over procedurally generated levels, we propose a method which empowers designers to express design constraints based on player actions. Graph grammars, resulting from the designer-expressed constraints, can generate sequences of desired player actions as well as their associated content. These action graphs are used to determine layouts and content for game levels. We test our method by applying it in a case study, which includes integration with a new dedicated generator for a 3D dungeon crawler game. We assess the quality of our method by evaluating how responsive and effective our control is. We do this by analyzing and measuring the features of generated levels and observing whether they fit the requirements indicated by the control.

1.2 Outline

In the next chapter, we survey previous work on recent procedural level generation, presented as a scientific paper, submitted for publication. The following chapter introduces our method, case study, and evaluation, also presented as a scientific paper, accepted for publication by *The Second Workshop on AI in the Game Design Process*, co-located with *The Ninth Annual AAAI Conference on AI in Interactive Digital Entertainment*. Chapter 4 provides a technical design overview of the software we developed for this project. We detail on the algorithmic pipeline in chapter 5, before conclusions are outlined in the final chapter. The final pages of this document are reserved for appendices; Appendix A contains screenshots of the developed applications; Appendix B visualizes the algorithmic procedures we perform to generate a dungeon.

Chapter 2

Related Work[1]

Procedural Generation of Dungeons: A Survey

Roland van der Linden, Ricardo Lopes and Rafael Bidarra

Computer Graphics and Visualization Group, Delft University of Technology, The Netherlands
roland.vanderlinden@gmail.com, r.lopes@tudelft.nl, r.bidarra@tudelft.nl

Abstract - The use of procedural content generation (PCG) techniques in the game industry is often restricted to very specific content, for only a small part of a game. This paper surveys research on procedural methods that generate entire game levels, specifically, dungeons, a type of level often encountered in role playing games. When considering complete game levels, dungeons can best demonstrate the benefits of PCG, specifically gameplay-based control over generation. In this paper, we identify common practices, pros and cons of different approaches and open challenges. We conclude that procedural dungeon generation is both rapid and diverse enough and that the foundations for enabling gameplay-based control are worth being researched. However, for the game industry to ever use PCG, research should focus on the most promising open challenges: (i) 3D generation, and (ii) more detailed and advanced gameplay-based control.

2.1 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. For instance, SpeedTree [IDV 09] is becoming a standard middleware to procedurally generate trees, as demonstrated by its integration in games like *Grand Theft Auto IV* [Rockstar 08], *Batman: Arkham Asylum* [Eidos 09] , *Battlefield 3* [EA 11], and many others. The use of PCG techniques in the game industry is often restricted to a very specific context, like SpeedTree, for only a small part of a game. 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 control:

[1] This chapter is presented as a scientific paper, which has been submitted for publication. All references can be found in the bibliography of this thesis.

designers, by giving away part of their control to an algorithm, are often suspicious of the unpredictable nature of the results of a generator.

However, both game studios and researchers are increasingly convinced of the benefits of establishing PCG as a mainstream method. These benefits include: (i) the *rapid* generation of content that fulfills a designer's requirements [Smith 11b], (ii) the possible *diversity* of generated content (even when using similar requirements), which may increase game replayability [Hastings 09; Smith 11a], (iii) the amount of *time* and *money* that a designer/company can spare in their game development process [Tutenel 08], and (iv) the fact that PCG can provide a basis for games to automatically *adapt* to their players [Lopes 11; Yannakakis 11].

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 often encountered in Role Playing Games (RPG). Dungeons are intrinsically tied to the history of PCG, as test beds showcasing PCG's potential in video games. *Rogue* [Toy 80], *The Elder Scrolls II*: *Daggerfall* [Bethesda 96] and *Diablo* [Blizzard 96] are some of the better known examples of this relationship between PCG and dungeons. In this paper, we investigate how, due to their characteristics and tradition, dungeon game levels are both a domain with a proven record on the above mentioned advantages of PCG and, reputedly, the most appropriate complete level category for exploring novel PCG contributions.

This survey provides an overview of dungeon generation methods presented so far, with a specific focus on how those generation methods can be controlled. The remainder of the paper is organized as follows; section 2.2 gives a more detailed introduction to procedural dungeon generation methods and how they can be controlled. Sections 2.3 through 2.8 detail on specific methods that are relevant for procedural dungeon generation, namely Cellular Automata (2.3), Generative Grammars (2.4), Genetic Algorithms (2.5), Occupancy Regulated Extension (2.6), Real-World Data Approach (2.7), and Constraint-Based Approach (2.8). Section 2.9 contains an overview of these methods and discusses various of their aspects. Finally, section 2.10 provides concluding remarks.

2.2 Controlling Procedural Dungeon Generation

Due to their characteristic topologic and geometric structure, dungeons are complete game levels which both *include* and *can influence* other gameplay elements. For example, the topology of a dungeon can influence how confused and lost a player feels, while the dungeon itself can contain such rich game entities as weapons, treasures or princesses to save. As such, dungeons have the potential to provide PCG methods with a very high level of reach.

A dungeon mostly consists 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.

A typical dungeon generation method consists of three elements:

- A representational model: an abstract, simplified representation of a dungeon, allowing a simple overview of the final dungeon structure (later on, it can serve as a guideline to create its geometry).
- A method for constructing the representational model.

- A method for mapping the model to the actual geometry of a dungeon.

Most surveyed research explains 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. Therefore this survey mostly focuses on the generation of the representational model.

A very important element in any procedural content generation method is the *control* it provides over the output. We refer to control as the set of options that a designer (or programmer) has to steer the level generation process, as well as the amount of effort that steering takes. Control refers also to how those options (parameters) cause understandable changes when altered, *i.e.* the intuitive responsiveness of a generator. The lack of proper parameters or the lack of understanding on what a parameter exactly does, may both contribute to poor control over a PCG method. This may lead to undesirable results or even catastrophic failures, preventing PCG from showcasing its potential.

When the first PCG methods were investigated (*e.g.* Perlin noise [Perlin 85]), the focus was more on how the methods worked and *what* could be generated. More recently, researchers became more concerned with *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 [Smelik 11] or controllable agents [Doran 10].

As discussed before, for dungeons, a specific sort of game-related spaces, gameplay is an influential aspect. A dungeon is typically a hostile environment, where the player needs to overcome obstacles (maze structures, enemies, puzzles) to achieve some goal (treasure, player / story progression). When compared to a city, for example, a dungeon is a space that is much more dependent on the player's presence. Therefore, in this paper we are especially interested in discussing gameplay-based control methods, where PCG parameters are more dependent to some sort of gameplay data.

2.3 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 at 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 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*. [Johnson 10] 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

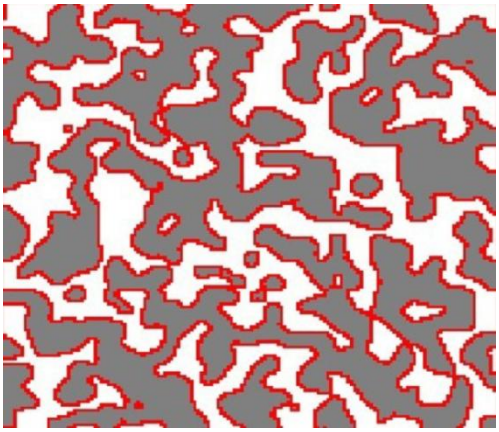


Fig. 2. A map generated with Cellular Automata. Colored areas represent: floor (grey), walls (red), and rocks (white) [Johnson 10].

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. 2. 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 certain connectivity. Therefore, gameplay features are rather 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.

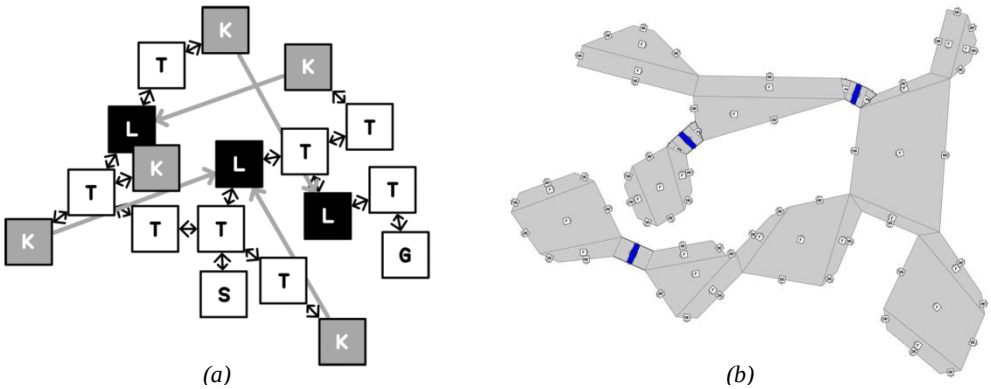


Fig. 3. (a) A mission with Tasks, Keys, and Locks. (b) Mission structure from (a), mapped to a spatial construction. Both images from [Dormans 11].

2.4 Generative Grammars

Generative grammars were originally used to describe sets of linguistic phrases [Chomsky 68]. This method creates phrases through finite selection from a list of recursive transformational rules, which include words as terminal symbols. Based on generative grammars, other grammars have been developed, *e.g.* graph grammars and shape grammars. These grammars only differ from the linguistic setup in that they use other terminal symbols (nodes and shapes) to allow graph and shape generation.

Dormans *et al.* [Dormans 11] use 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. 3(a) shows a mission network and Fig. 3(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 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 full 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 [Muller 06] encourages future work on this direction.

Dormans *et al.* do not directly use parameters in their approach, since control is exerted by the different rules in the graph and shape grammars. However, as with

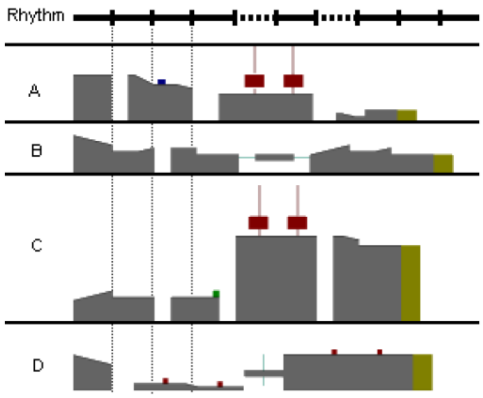


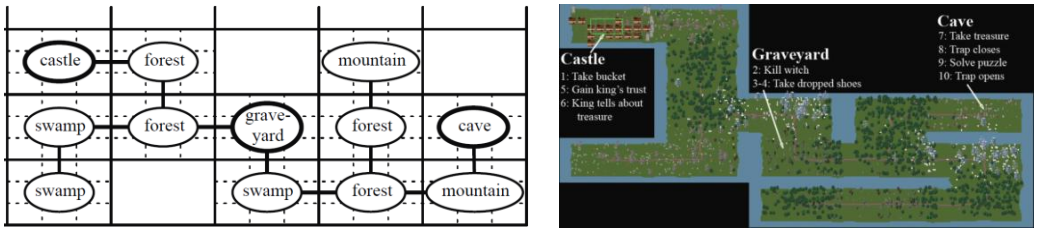
Fig. 4. Different geometry interpretations for a given rhythm [Smith 09].

Muller *et al*. [Muller 06], it takes a lot of effort to understand and work with these grammars. On a more positive note, the authors were able to introduce some 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, are all suggested as interesting future directions towards a richer gameplay-based PCG control.

Another interesting research direction is proposed by Smith *et al.* [Smith 09]. Although the authors generate complete 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. Fig. 4 showcases an example conversion. Many levels are created by connecting rhythm groups, and a set of implemented critics selects the best level.

Although or only creates platform levels, this approach ties in with dungeon generation. As with Dormans *et al.*, 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' is perhaps not very relevant 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 (like 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



(a) *(b)*

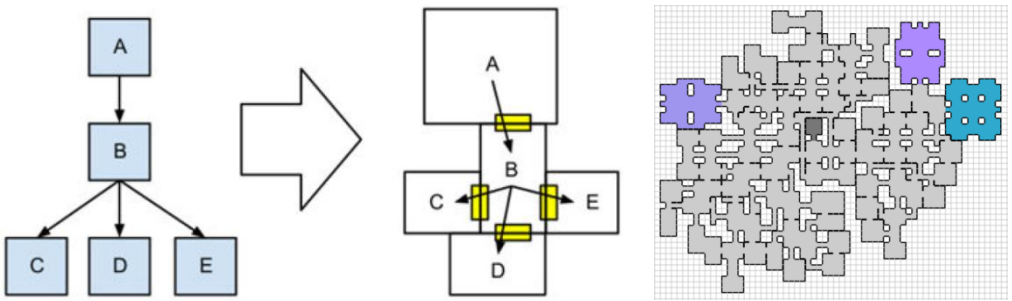
Fig. 5. (a) A space tree mapped to a grid; plot locations are bold. (b) A game world generated from a space tree, similar to (a). Both images from [Hartsook 11].

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 the different levels of abstraction provide a lot of control options, and allow for the versatile generation of very disparate levels. They relate quite seamless to gameplay (especially in its platform genre), although achieved at a lower level than, for example, the missions of Dormans *et al.*.

2.5 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 combinational 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, assuming the representational model and fitness function are flawless.

Hartsook *et al.* [Hartsook 11] 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 capturing that 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



(a) *(b)*

Fig. 6. (a) Translation from tree structure to map. (b) Example of a generated map. Event rooms are highlighted. Both images from [Valtchanov 12].

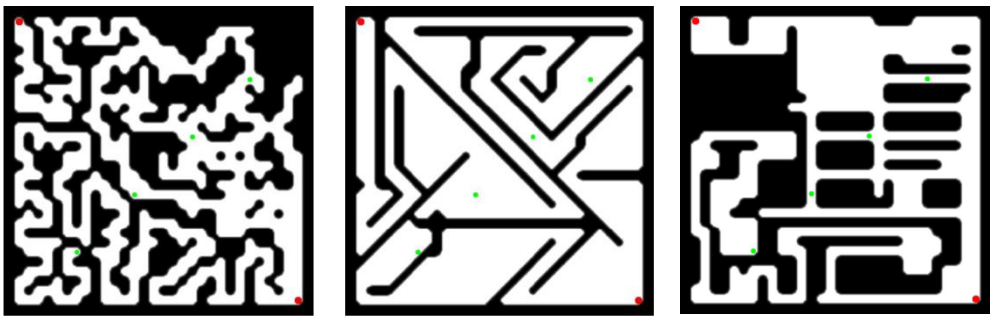
world, linearity of the world, likelihood of enemy encounters, and likelihood of finding treasures. The space tree that the genetic algorithm selects as the most optimal, after a set number of iterations, is then used to generate a 2D game world, where tree nodes are mapped to a grid using a recursive 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. 5(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. 5(b)).

As with Dormans *et al.*, 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 2.1, another advantage is that adding the element of story to 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 allows a game world directly suited to 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 *et al.* [Valtchanov 12] 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. 6(a) for an example of the tree structure (genetic representation).



(a) *(b)* *(c)*

Fig. 7. Mazes generated by [Ashlock 11]. All mazes have been created with fitness function F2. (a) Direct binary representation. (b) Indicrect positive representation. (c) Indirect negative representation. Green dots are checkpoints. Red dots are start and finish.

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. 6(b)). During the generation process, the maps are directly created to test if rooms can be placed the way the tree represents it. If that is not the case, that 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 clear motivation for their placement.

Ashlock *et al.* [Ashlock 11] 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. 7).

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

- 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”.

Fig. 7 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 want 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 *et al.* 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 actually investigate different fitness functions. Other genetic algorithm parameters are the mutation and crossover chance percentages 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 allow 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 final 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, still a standard in these methods.

2.6 Occupancy Regulated Extension

Although occupancy-regulated extension (ORE) was proposed by Mawhorter *et al.* [Mawhorter 10] 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.* [Smith 09] 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 partial designed levels would be of great value. Imagine a designer placing special event rooms and then having an algorithm adding the other parts of the level that are more generic in nature. This mixed-initiative approach would allow both level versatility, and a lot of control for designers, while still taking work out of their hands. Additionally, it would fit the principles of dungeon design, where special rooms are connected through more generic hallways. Also, using a chunk library fits well with 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 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 *et al.* 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.

2.7 Real-World Data

Another technique which can be suitable for dungeon generation is proposed by Merrell *et al.* [Merrel 10]. 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.

Requirements to the building include: (i) a list of the different rooms, (ii) their adjacencies, and (iii) their desired sizes; see Fig. 8(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

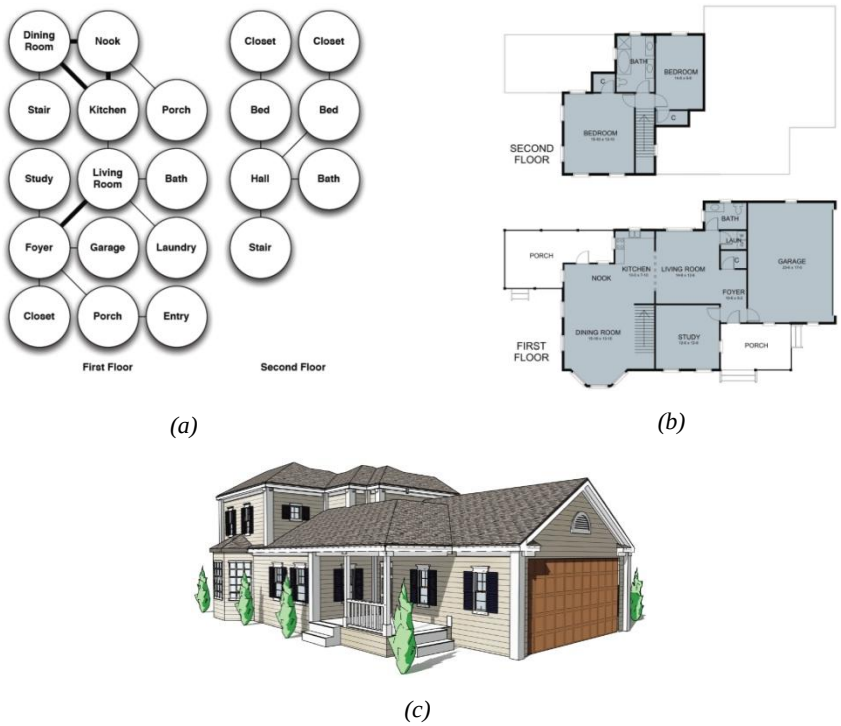


Fig. 8. Results from [Merrel 10]. (a) An architectural program. (b) A set of floor plans, based on (a). (c) A 3D residential building model, based on (b).

academic community. Another example is the generation of *Monopoly* boards using real economic and social data [Friberger 12].

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. 8(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. An example 3D model is displayed in Fig. 8(c).

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 seem to have really captured this, such that only some high-level parameters are needed to generate a realistic 3D model of a residential building. Still, although controlling the generation process can be very clear and easy, that 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 is dependent on the floor-plan algorithm.

From 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

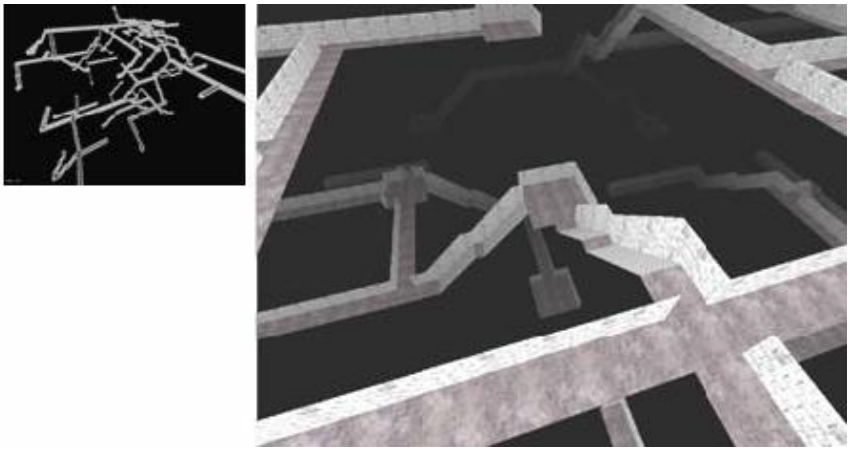


Fig. 9. A 3D dungeon generated based on constraints [Roden 04].

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 previous analyzed methods, the use of a stepwise 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 if dungeon types of a very similar structure are required. Otherwise, if dungeons in all kinds of shapes need to be generated, this approach is perhaps a bit too restrictive, unless the real-world data is large enough, and the Bayesian network can deal with it.

2.8 Constraint-Based

Roden *et al.* [Roden 04] proposed a pipeline for generating underground levels for 3D games. 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 3D geometry, using a set of prefabricated geometry sections (similar to the chunks from Mawhorter *et al.*). After the prefabricated sections are interconnected, objects (such as furniture) are placed in the dungeon. An example of a generated dungeon is displayed in Fig. 9.

This approach presents, to the best of our knowledge, the only method that explicitly considers 3D dungeon generation and topology/object integration. 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 terminal fixed nodes (entry/exits).

As for control, Roden *et al.* use as the input for their level generation an initial topology of a graph and a set of constraints. 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 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 3D 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 up to the designer to translate gameplay concepts (like pacing, difficulty) into topology, adjacency, etc.

2.9 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 dungeons, no single approach stands out to allow all types of dungeons to be generated. The existence of such multiple distinct methods demonstrates that dungeon generation can still typically involve multiple distinct requirements. The table in Fig. 10 provides an overview of the methods we surveyed. Such variation on dungeon generation strategies can be observed in the columns for the *Approach*, *Output types* and *Variety of results*.

From the table, we can also identify the less investigated challenges in dungeon generation: performance and 3D content. Methods with good enough performance are cellular automata (the only one supporting runtime generation), and generative grammars. Genetic algorithms are search-based algorithms and, due to their nature, the more optimization is required, the worse performance becomes. 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 open research questions in on-line dungeon generation, including for example, generating the intermediate spaces between some entry exit spaces, as the player advances through the dungeon.

As for generation of 3D dungeons, this seems to be absent in most methods. The exception is the work by Roden *et al.* [Roden 04], a preliminary but valuable step towards the generation of more complex dungeon structures. However, their results indicate that 3D generated dungeons are still far from designer-made dungeons. Their 3D intermediate graph structure seems good enough to generate valid abstract dungeon designs, but its realization to a 3D world will require further research.

Although all these approaches are quite distinct, and performance and 3D content remain as open challenges, there are already some good common practices in procedural dungeon generation. For example, a stepwise approach is adopted by

Overview of Surveyed Methods and their Properties

		Properties						
Category	**Reference**	**Approach**	**Control provided**	**Performance**	**Output Types**	**Variety of Results**	**Gameplay features**	**Options for 3D**
Cellular Automata	Johnson 10	Grid cells state modification	Initial state: # of generation	Run-time generation	2D dungeon with floor, rock, wall cells	Chaotic maps: little variation	Separate	Future work: 3D cellular automata
Generative Grammar	Dormans 11	Graph and shape grammars	Input missions	Not run-time generated	2D dungeon with rooms, locks, keys	Maps as versatile as missions	Missions	-
	Smith 09	Rhythm-to-actions -to-geometry grammars	Path, rhythm, objects, critics	Not run-time generated	2D platformer level	Combination of premade level segments	Player-based rhythm	-
Genetic Algorithms	Hartsook 11	Space tree generation	Game story, fitness function, player model	Low: search-based and backtracking	2D dungeon-like game worlds	Combination of premade location types	Game story	Future work: 3D space tree
	Valtchanov 12	Tree mutation	Fitness function, genetic parameters	Low: search-based	2D dungeon	Tightly packed rooms connected by hallways	Special event rooms	Future work: 3D tree
	Ashlock 11	Combining 4 genetic representations and 5 fitness functions	(same as above)	(same as above)	2D mazes	Larger between combinations	-	-
Occupancy Regulated Extension	Mawhorter 10	Position-based combinations of level chunks	Chunk library, mixed-initiative	Not run-time generated	2D platformer level	Combination of premade level chunks	Chunks contain game-related items	-
Real-World Data	Merrel 10	Bayesian network trained with real data	# rooms, sizes, distances, style of the building	from a few seconds up to 7 minutes	3D residential building layout	Varied residential building models	-	-
Constraint-based	Roden 04	Constrained graph generatation	Topology, node placement	unknown	3D underground level	Small rooms connected by hallways	-	-

Fig. 10. Overview of the methods we surveyed, highlighting key characteristics of each method.

Dormans *et al.* [Dormans 11], Hartsook *et al.* [Hartsook 11], and Merrel *et al.* [Merrel 10]. A high level input is gradually transformed into a detailed result, where each step increases the amount of details in the model or even builds 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. An example is the approach by Mawhorter *et al.* [Mawhorter 10], 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.

Fig. 10 also provides interesting insights into the control features over the generation process (columns *Control provided* and *Gameplay features*). In general, most control parameters are specific to the technique used, and it is often unclear what adjusting a parameter will mean for the generated maps. Cellular automata and constraint-based approaches use a straightforward control mechanism over their algorithm's low level parameters, thus requiring a full understanding of their 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 room adjacency better supports 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 a lot of knowledge about the entire generation process: it is not very intuitive, especially from a gameplay perspective. On the other hand, the genetic algorithm by Hartsook *et al.* [Hartsook 11] 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 additional gameplay-based layer of control (story) is added on top of the fitness function, allowing for a more meaningful control and generation.

This story-based approach and Dormans *et al.* [Dormans 11] mission-based method are the most promising advances towards a more gameplay-based generation 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 and story, further gameplay-based control methods could include, for example, player challenges, player actions, performance or the context of the dungeon in an overlapping story. The more PCG becomes dependent on gameplay, the more apparent it becomes a need for encoding, in a generic way, the mapping between content and gameplay-data. An example of such an encoding mechanism is game object semantics, *i.e.* additional information about game content, beyond its geometry. We analyzed a good example of this in Merrel *et al.* [Merrel 10], where room functions are known, and because of this, more meaningful building layouts can be generated. For dungeons, more gameplay-related semantics could help support more powerful PCG methods. For example, enriching game content with the knowledge on which player

actions they can enable could help generating more meaningful dungeons. Gameplay semantics have already been applied successfully to the procedural generation of game worlds [Lopes 12; Kessing 12].

2.10 Conclusions

Dungeon levels for RPG games are probably among the few game world sorts that have been successfully generated 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 which showcases 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 achieve some designer's requirements. This generation is *rapid* enough, in a sense that it is faster than creating the dungeon content manually, thus leading to spare time and money in the game development process. However, as discussed in section 2.9, run-time performance is not essential due to the importance of player progression inside a dungeon. Therefore, dungeon generation can benefit from all the advantages of PCG methods which are not appropriate for runtime generation.

We found that there is a wide diversity in the generated results. This happens not only for similar requirements, but mostly among different PCG methods, which lead to a wide array of dungeon types. From this survey, we also conclude that gameplay-based control is already being successfully investigated, with the use of missions, stories and even player models to control PCG. This already provides a significant basis for games to automatically adapt to their players.

For future research, we conclude that the most promising challenges lie in the generation of 3D dungeons with further extended gameplay-based control. Contributing to either or both will further explore the huge diversity of the generated content and provide a solid basis for more advanced game adaptation. As discussed in section 2.9, current achievements show that these research goals are possible and that there still is a long way to go. We believe that dungeon generation is a domain where PCG can reap some low-hanging fruit. Investing research efforts in this direction is, therefore, a direct contribution to bring PCG into mainstream game industry use.

2.11 Acknowledgements

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

Chapter 3

Method[2]

Designing Procedurally Generated Levels

Roland van der Linden, Ricardo Lopes and Rafael Bidarra

Computer Graphics and Visualization Group, Delft University of Technology, The Netherlands
roland.vanderlinden@gmail.com, r.lopes@tudelft.nl, r.bidarra@tudelft.nl

Abstract - We aim to improve on the design of procedurally generated game levels. Our approach empowers game designers to author and control level generators, by expressing gameplay-related design constraints. Graph grammars, resulting from these designer-expressed constraints, can generate sequences of desired player actions as well as their associated target content. These action graphs are used to determine layouts and content for game levels. We showcase this approach with a case study on a dungeon crawler game. Results allow us to conclude that our control mechanisms are both expressive and powerful, effectively supporting designers to procedurally generate levels.

3.1 Introduction

It would be great if *computer-generated* levels could also be somehow *designed*. Procedural content generation (PCG) concerns itself with the algorithmic creation of content. The potential benefits of using PCG in games are already well established: (i) the rapid reliable generation of game content [Smith 11b], (ii) the increased variability of the generated content [Hastings 09; Smith 11a], and (iii) its use to support player-centered adaptive games [Lopes 11; Yannakakis 11]. However, these benefits highly depend on an essential feature of any generative method: the degree of control over the generator.

Proper control over generative methods ensures that the created content contains the features designers envision. In other words, control determines what a generation algorithm can and cannot design. Therefore, the lack of intuitive control over generators can partially explain the absence of procedural generation in commercial products [Smelik 11].

[2] This chapter is presented as a scientific paper, which has been accepted for publication by *The Second Workshop on AI in the Game Design Process*, co-located with *The Ninth Annual AAAI Conference on AI in Interactive Digital Entertainment*. All references can be found in the bibliography of this thesis.

The aim of this research is to improve on this control, and particularly, to find out how *designers* can use gameplay as the *vocabulary* to control the procedural generation of game levels. We argue that the geometry, topology and content of a game level should mostly follow from the specific ways in which a player can interact with a game (gameplay), and not the other way round. In this paper, we propose a generic method for designing procedurally generated levels by specifying their expected gameplay. A graph grammar, resulting from designer-expressed constraints, generates sequences of player actions as well as their associated content. This action graph can be then used to determine a game level layout.

To showcase our method, we apply it to a specific form of levels: dungeons. These are a type of game level often encountered in Role Playing Games, and mostly consist of challenges in enclosed space structures (*e.g.* caves, cellars). Dungeons are of particular interest since they heavily rely on player-centered gameplay. In contrast, in more open and active game levels (*e.g.* cities), player interaction is just one type of the many events occurring.

In the next section, we survey previous work on procedural level generation. The following two sections introduce our method. First, by proposing our generative graph grammar and then by discussing its integration in an existing game. The subsequent section discusses results and evaluation of our control method, before conclusions are outlined in the final section.

3.2 Related Work

Research in the procedural generation of game levels has advanced significantly. Most related work has a focus other than effective gameplay-based control over generative methods. Johnson *et al*. [Johnson 10] use the self-organization capabilities of cellular automata to generate natural and chaotic infinite cave levels. For platform games, Mawhorter *et al.* [Mawhorter 10] propose a mixed-initiative approach, where level chunks are assembled to generate level sections in between manually designed ones. Search-based evolutionary algorithms were investigated for the generation of game levels. In one example, Valtchanov *et al*. [Valtchanov 12] use their fitness function to optimize topology generation, by specifying a strong preference for dungeons composed of small, tightly packed clusters of rooms, inter-connected by hallways. Roden *et al*. [Roden 04] proposed a pipeline for generating underground levels. The authors also use constraints to control graph (level) generation. However, their constraints directly relate to the topology and geometry of a level, and not to gameplay.

For our purposes, gameplay-based control over generative methods is more interesting and relevant, as it has allowed player-based rhythm, game narratives and game missions to steer level generation. Smith *et al*. [Smith 09] propose a two-layered grammar-based approach to generate platform levels. Player actions (like *jumping*) are also used, but only to define desired interaction rhythms, which then constrain level generation. Hartsook *et al*. [Hartsook 11] use a genetic algorithm to create 2D role playing game worlds. The initial genetic representation captures linked narrative events and the fitness function optimizes correct sequencing. This way, narratives that are meaningful to the player steer the generation.

Dormans' [Dormans 10] work shares most similarities with our research. The author proposes grammars to generate dungeons. Missions are generated through a graph

grammar, representing the sequential tasks a player should perform. This mission graph is used by a shape grammar to create a corresponding game space. Our approach differs from this method through three main contributions: (i) designers fully specify and control grammar rule re-writing with a more natural design-oriented vocabulary (and not grammar-oriented), (ii) they can create additional spatial relationships between tasks, and (iii) because of this and because of its semantic nature, our method has the potential to have a wider range of generic application in games and genres, than done before.

3.3 Grammars for Player Actions

Typically, the geometry and content of a designed game level follow from gameplay requirements, not the other way round. This happens because gameplay naturally determines which unique content is required, whereas content alone can be ambiguous as to which gameplay it sustains. Optimizing content to match gameplay is more natural, since it is more appropriate in the level design setting.

Our approach allows designers to author procedurally generated levels, empowering them with intuitive control over the generative methods. Control is realized *a priori*, by specifying all the *design constraints*, expressed in a gameplay design-oriented vocabulary. Player actions to perform in game (*e.g.* fighting), their sequencing, relationships and content (*e.g.* fighting a *dragon*) can be expressed as (design) constraints. These designer-authored constraints directly result in a generative graph grammar, and multiple grammars can be expressed through different sets of constraints. A grammar is thus tailored by a designer to fit a specific game. It is able to generate graphs of player actions which subsequently determine layouts for game levels. For each generated graph, specific content should be synthesized by following the graph's constraints, for example, by placing in a game level the objects required by each action, in the appropriate sequence.

3.4 Expressing Design Constraints

In our grammar-based method, designers author level generators by expressing their design constraints, specified as *player actions*, their relationships and related content. A player action, also considered by Smith *et al.* and Dormans, describes gameplay by inherently indicating what a player can do in a level. There is no universal set of actions, so for each game, designers have to specify their own. Constraints were implemented into Entika [Kessing 12], a semantic library editor used to express semantic attributes and relationships as constraints to layout solving.

Individual player actions are specified as a verb and a target, *e.g.* kill a dragon. Targets typically relate to game content, *e.g.* the *dragon* in *kill a dragon*. Content refers to the objects, non-playing characters (NPC) and their relationships. *Entika* allows you to directly specify this target as a semantic entity linked to content (*e.g.* a 3D model, a procedure). This can even be expressed in more abstract terms, like other constraints to be solved later (*e.g.* "some animal with scales").

Player actions are most interesting and useful if they are considered in logical groups and not individually. Sequences of actions, and even branching sequences (representing player choices), can capture more complex and intricate gameplay. As

such, player actions can also be specified as compound, by which a name represents the whole composition of sub-actions which fulfill it. For example, as seen below, *acquiring a key* can be fulfilled by *killing an enemy* and then *looting the key from its body*.

Action 1: **Acquire key**
Sub-actions: *Kill an enemy → Loot key from body*

Action 2: **Entering locked chamber**
Sub-actions: (**Acquire key** → *Unlock related door → Move through doorway*)
|| (*Climb on roof → Enter chimney*)

Furthermore, each sub-action can itself be an individual or compound action. Sequences of actions can also include branching, to capture player choices or alternatives (resembling the logical 'or' operator, ||). Below we see an example, with two alternatives of action sequences to fulfill *entering a locked chamber*.

Action 1: **Acquire key**
Sub-actions:
Option 1: if *Difficulty* == 25 and *Length* > 25
1.1: *Kill an enemy → Loot key from body*
1.2: *Distract enemy → Steal key*
Option 2: else
Look under doormat → Pickup key

Additionally, there may exist totally disjoint alternatives of fulfilling a compound action, which depend on *designer choice*, rather than on player choice. In other words, at design time (generation), and not at game time, several options for fulfilling a compound action may be specified. Selecting one and only option among them (*i.e.* re-writing that player action) can increase variability and flexibility. This selection can be done randomly or controlled by the designer. For the latter, designers steer selection by (i) making re-writing options dependent on given conditions, and (ii) by specifying, for each generation, a set of parameter values to evaluate against those conditions. For example and as shown below for re-writing *Acquire key*, if specific matching values for Difficulty and Length are input at generation time, then option 1 is selected: a random selection between option 1.1 and 1.2.

Expressing all these design constraints enables designers to author how gameplay can progress in a level. To increase this expressive power, we defined two types of explicit relationships: (a) *co-located* actions (*e.g.* killing an enemy, looting a key from his body), and (b) *semantically connected* action pairs (*e.g.* a key and its lock, like in Dormans' work).

Actions that must be co-located are a special example of spatial relationships. Game spaces refer to the bounded areas in which the player can navigate, and in which the content is located. Typically, determining spaces is highly game dependent. Furthermore, the link between spaces and actions can be unclear (a *dragon* can be killed in different spaces). Therefore, our approach does not include constraints on the spaces where actions (and their target content) should occur.

However, that does not preclude that, as mentioned above, some actions *must* be together in the same space, regardless of what that space is. For example, two individual actions targeting the same object instance. This object exists in a single space, and therefore the actions are also required to be in the same space (*e.g. killing a dragon* and *looting a dragon*). This is why the co-location of two individual player actions can be expressed as a constraint.

3.5 Graph Generation

Once all design constraints have been expressed by designers, they result in an instance of a grammar, able to re-write a set of initial action(s) into an action graph. Nodes in that graph represent (groups of) player actions and edges indicate the order of encounter. The information in the final graph will determine a game level layout.

Different sets of designer-specified constraints result in distinct generative grammars. Building such grammars is a way of controlling generation. Additionally, generation can still be controlled on a single grammar basis, by inputting initial parameter values which, as explained before, will steer the selection of the appropriate re-writing options.

The initial graph is composed of a set of start action node(s). The generative algorithm re-writes compound actions into a graph of linked individual ones. It takes the following steps while a single compound action still exists:

1. Select the first compound action in the graph
2. Select an option based on parameter values (randomly, if conditions are absent)
3. If needed, randomly select sub-options
4. Convert the selected rewriting option to a graph of sub-action nodes (sub-graph).
5. Add the compound action as the parent of all sub-graph nodes
6. Replace the compound action with the subgraph. Connect all the predecessors of the compound action with the first nodes in the subgraph. Connect all the successors of the compound action with the last nodes in the subgraph.

The next step is to group actions into the same space, *i.e.* solve co-location of actions. New group nodes are created from merging the individual nodes which must be co-located. These new nodes are groups of actions which represent a space. Merging nodes has some particularities. If either or both nodes were already in a group, all nodes are merged into a new group. Merging must occur because part of a longer co-location sequence may be cut in half due to branching combined with depth-first recursion. If the two nodes to be merged exist in the same tree level (they share a parent or a child node), more duplicates of one of them might theoretically exist in that same level. The algorithm inspects all the stored parent compound actions (step 5 above) which originated each node. Merging only occurs within these compound actions hierarchies. Finally, semantically connected pairs are marked by inspecting all actions and backtracking their compound action parent-hierarchy. Fig. 11(a) displays an example of a generated action graph, where a co-located group node for *Fight Melee Enemy* and *Loot Key* can be observed.

With this generative algorithm, multiple grammars and parameters can generate a variety of action graphs. These not only indicate the sequence of actions that must occur in-game, but also other requirements as their target content, the groups where

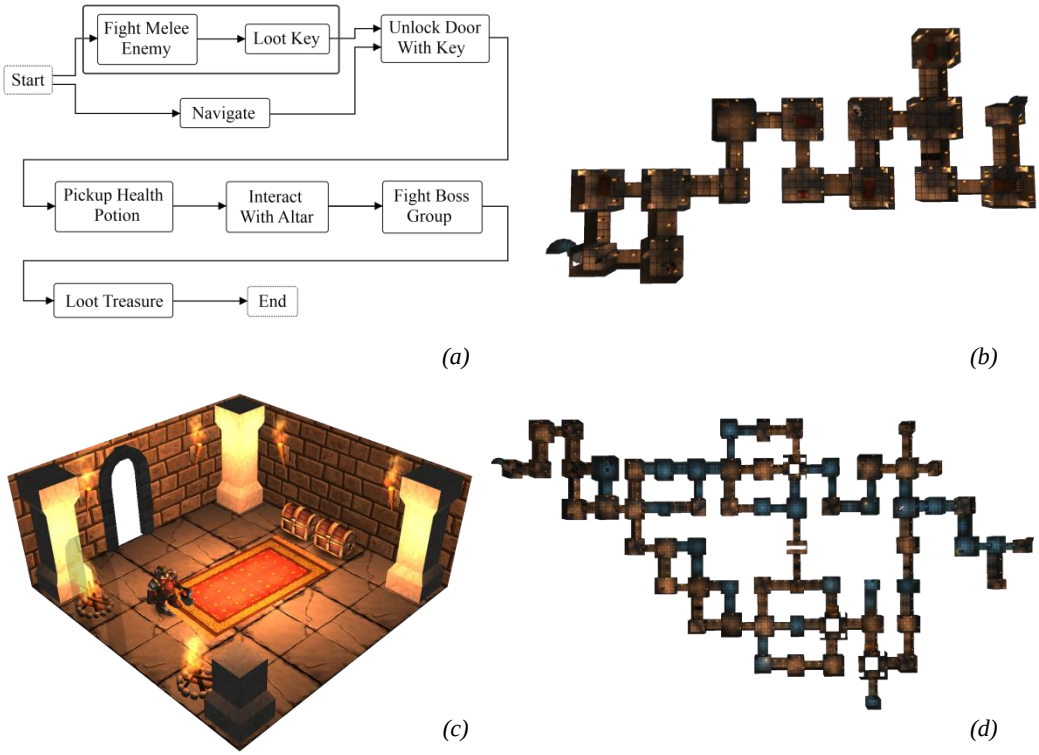


Fig. 11. (a) Graph of player actions for an example generated dungeon. (b) Dwarf Quest dungeon layout, generated for (a). (c) 'Loot Treasure' room, generated for (a). (d) Another (unrelated) example of a dungeon layout (colors are decorative).

some actions must occur in the same space, as well as semantically connected pairs of actions.

3.6 Case Study: Dwarf Quest

With the approach described so far, designers can express gameplay-related constraints which ultimately result in action graphs describing game level requirements. This approach can be considered generic, in the sense that player actions and related design constraints can be created and manipulated across different games. Furthermore, generated graphs can even be made re-usable and game independent as long as the target content of each action is abstract enough.

However, the full realization of our approach still needs that such abstract action graphs be converted into an actual specific game level. For use in a game, those action graphs should be integrated with a dedicated generator. Designers working with *Entika* and player actions still need algorithms to actually synthesize levels. Given the information stored in the graph, these algorithms can be, for example, simple layout solving techniques [Tutenel 10].

For our case study, we used Dwarf Quest [WCG 12], a typical dungeon crawler game. The player explores dungeons, fighting enemies, solving key-lock challenges, finding treasures and boosting skills. Dungeons are composed of rooms, in which the main content is located, and hallways, connecting rooms.

We implemented a Dwarf Quest generator, which converts action graphs, generated by some grammar, into dungeon levels. Several Dwarf Quest features were essential for constraining the dedicated generator. First, the rooms and hallways have to be orthogonally placed, with a maximum of four connections (doors) per room. Second, due to game engine and camera reasons, rooms cannot be made too large, implying that spaces cannot hold too many actions. More details on the algorithms of the dedicated generator can be found in Chapter 5.

The Dwarf Quest generator takes an action graph as input and yields a room graph. The algorithm takes the following steps:

- *space assignment* converts nodes of the action graph into rooms and edges into hallways;
- *layout pre-processing* converts the graph into a planar graph (without overlapping edges) and reduces edges per node to four (by adding new intermediate nodes, *i.e.* rooms);
- *layout solving* converts the planar graph into an orthogonal graph mapped onto the 2D grid map of the game;
- *layout post-processing* still needs to optimize the resulting layout. As with other orthogonal planar graph drawing techniques, long edges (hallways) are a side-effect. Long hallways are then compressed, and rooms added into the ones which cannot be further compressed.

Finally, the geometry of rooms, hallways, objects and NPC is actually created and placed. Dwarf Quest's designer had already randomly generated levels as a basis which he then manually finished for inclusion in the game. We extended this generator with our control layer. Predefined room configurations indicate size, entrances and possible object locations. Configurations are selected according to the original action graph, matching the target content of an actions node (*i.e.* the content associated to that action) to a possible room configuration. Room configurations instruct the generator to instantiate rooms (geometry, lights, doors) and content (objects, and NPC) in the location defined with the layout solving steps. To maintain player immersion, decorations, thematically related to the created objects, are instantiated. Finally, semantically connected pairs are marked so the game engine knows how to deal with them, *e.g.* so that a lever actually lowers a closed bridge. Fig. 11 displays examples of generated dungeons (b, d), and one of their rooms (c).

3.7 Results and Discussion

The aim of this research is to provide a more intuitive control over procedurally generated levels, through a gameplay-based vocabulary. Before evaluating our approach with designers, we sought to measure the responsiveness and effectiveness of our control mechanisms. For this, we analyzed the expressive range of its generative space (*i.e.* the variety of generated levels and the impact of changing parameters), as introduced by Smith *et al*. [Smith 10].

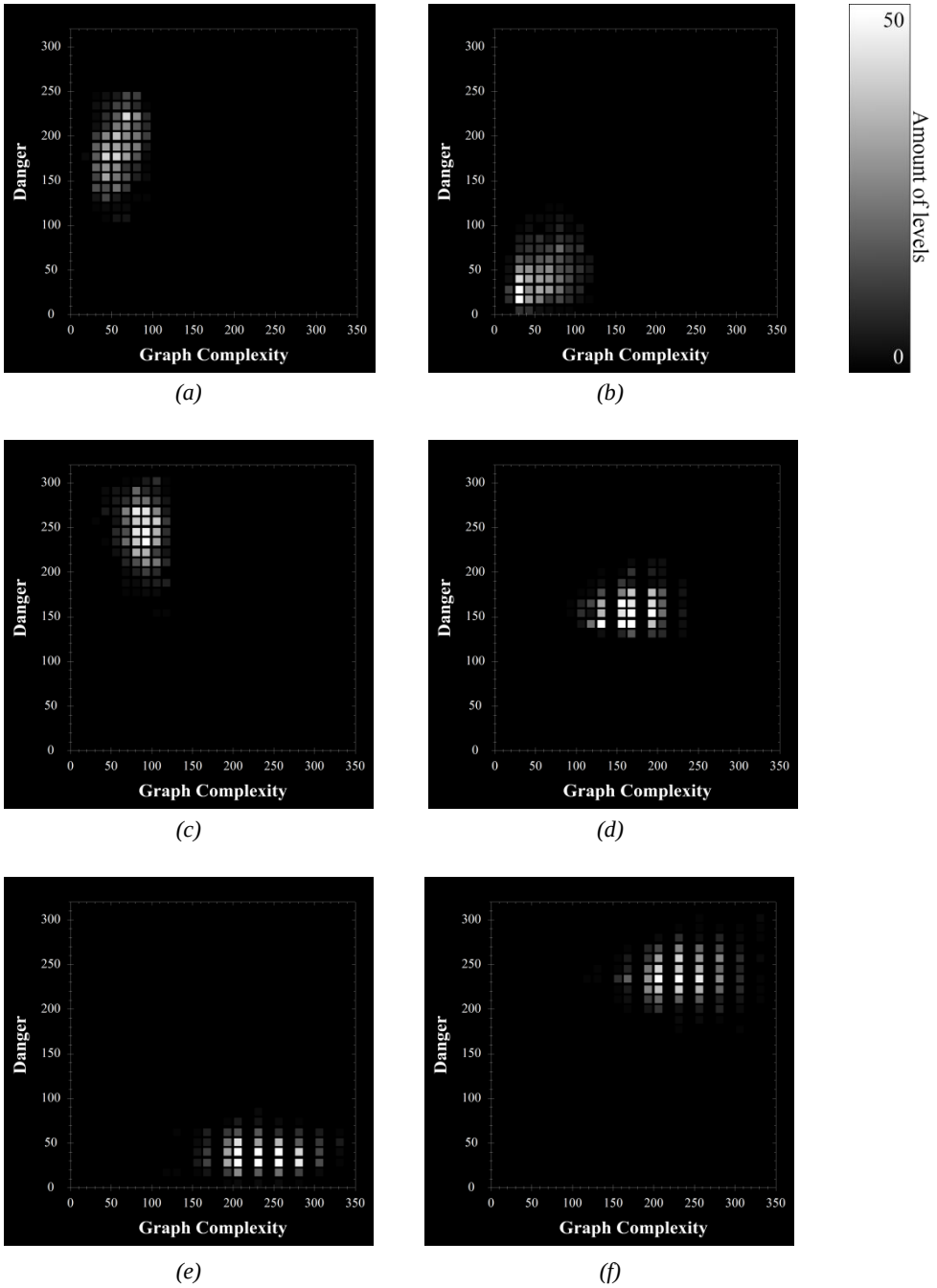


Fig. 12. Histograms for graph complexity and danger, measured for 1000 generated dungeons. Graph complexity is the number of sub-graphs in the final level layout. Danger is the total amount of damage points a level can inflict. Results are displayed for: (a) a grammar without parameters. Another grammar was created with control parameters for dungeon length and challenge difficulty, expressed in a designer-created scale (0 to 100). Results are displayed for parameter inputs of, respectively, length and difficulty: (b) 12-48 and 20-50 (c) 40 and 100 (d) 70 and 40 (e) 90 and 5-25 (f) 90 and 60. As an indication of performance, the average processing time for the levels generated in (b) and (f) are 1 and 2 seconds, respectively.

The generative space can be shaped by our control tools, *i.e.* the graph grammars and the parameters created by designers (in this case for Dwarf Quest). We represent the generative space by a 2D histogram, where the axes are defined by the range of metric scores measuring level features. This allows to view peaks of commonly created content and possible holes in the generative space [Smith 10]. As metrics we use *graph complexity* and *danger*, as we believe that these indicate important gameplay features of a designed Dwarf Quest level.

Graph complexity indicates the structural complexity of the generated level. It captures the duration of the level, as well as the amount of choices a player can do and decide. Previously used for molecular complexity in chemistry [Bertz 97], graph complexity is, for our purposes, the number of subgraphs of the rooms graph.

Danger quantifies the capacity of the whole level to inflict harm to the health of the player character. Like in Smith *et al.*, it captures how the level can potentially kill players. We have based danger on Valve's *game intensity* metric [Tremblay 13], which measures the amount of player health lost during two intensity updates. Danger is an estimate of the *expected game intensity* for a generated level. It is calculated by summing the average amount of damage dealt to the player, for all damage-dealing components.

The histograms in Fig. 12 show the measurements performed on dungeons from two different grammars. For each histogram, 1000 dungeons were generated. Fig. 12a plots generated levels from a first grammar, featuring no parameters. This grammar yields a dungeon that is 5 or 6 challenges long, each belonging to the harder segment of Dwarf Quest's challenges spectrum (*e.g.* fight a boss). The resulting dungeons have a rather linear structure, but do pose a challenging experience. With such a grammar, designers can control all generated levels to these features while maintaining some variation, as seen in the figure. This shows they can create highly specialized level generators, to be used, for example, in games where a very consistent gameplay experience is desired.

Fig. 12b through Fig. 12f show levels generated from a second grammar, featuring control parameters. The parameters *dungeon length* and *challenge difficulty* were specified for this grammar, with a designer created scale ranging from 0 to 100. As explained before, parameters are added to compound player actions to constrain which options are available to rewrite them. In this second grammar, higher length values correspond to rewriting options with longer action sequences. And challenge difficulty values correspond to the difficulty a designer perceived for that option. As outlined in Fig. 12b-f, different input parameter values were used to generate levels. This resulted in the following dungeon features: (Fig. 12b) a simple structure and minor danger, (Fig. 12c) a simple structure and very high danger, (Fig. 12d) a medium complex structure and medium danger, (Fig. 12e) a complex structure and low danger, and (Fig. 12f) a complex structure and high danger.

Parameters add flexibility to this second grammar, allowing fine-grained control over dungeon features. The grammar can potentially create any level in the generative space visible in Fig. 12b through Fig. 12f, maintaining variation as observed. Parameters add control over what and when level generators can specialize in. This functionality can be used by designers: (i) as a design tool, to select a number of procedurally generated levels with specific features and include them in their game, (ii) to give away some of that control to players, where the parameters can be used as game

options, and (iii) for adaptive games, where the parameters are derived by some algorithm (*e.g.* based on a player model).

3.8 Conclusions and Future Work

We proposed an approach that enables designers to exercise control over procedurally generated levels by means of a gameplay vocabulary. With our approach, procedurally generated levels can be *designed* through the expression of gameplay-related design constraints which ultimately steer content generation.

Through our case study, we conclude that these design constraints are expressive enough, able to cover a wide generative space of possible *Dwarf Quest* game levels. Furthermore, we conclude that this degree of control is powerful enough to steer generation into distinct sets of desired *Dwarf Quest* level features. This control opens several possibilities of new game design applications, as discussed in the previous section. We believe these conclusions hold for other action games beyond our case study.

As for future work, our next step is to evaluate how intuitive this method is for game designers, by conducting user studies and/or interviews. On a longer term, we consider our approach eligible for adaptivity, where level generation is based on the performance of the player. Our grammar parameters, as specified by the designers, can be adjusted between generation sessions. As such, the performance of the player in a single dungeon may define the parameters of the next generated dungeon. Our focus on gameplay, as the vocabulary to design procedurally generated levels, supports control over generated interactive content. However, it does not fully support control over all aesthetic content (*e.g.* decorations). We believe that storytelling would provide an interesting extension atop our action-based vocabulary. Not only is storytelling an even more natural concept for game designers, but they can also capture both gameplay and aesthetic features.

In short, the current form of player actions is already quite expressive and powerful to effectively contribute to a better design of procedurally generated levels.

3.9 Acknowledgements

We gratefully acknowledge Dylan Nagel for giving us valuable feedback in several occasions, as well as full access to Dwarf Quest's source code. This work was partly supported by the Portuguese Foundation for Science and Technology under grant SFRH/BD/62463/2009.

Chapter 4

Technical Design

This chapter provides an overview of the technical components developed during this project. Our focus is on overall design, not implementation. The purpose of this chapter is to introduce the design we developed, some of the technical challenges we solved, and clarify design decisions we made.

4.1 Requirements and Principles

The main requirement of the system is to produce playable Dwarf Quest dungeons based on designer-expressed design constraints. Our goal is to be as generic as possible, which means we want a large portion of components to be independent of Dwarf Quest-specific features. Our action graphs describing Dwarf Quest's dungeons could, in theory, be used as levels for other games, because we believe actions are applicable to many other games. We refer to these components as the *generic subsystem*, and we consider it an important benchmark of the project. The generic subsystem is supposed to produce action graphs, groups, and related information. We aim to visualize the output and process of this subsystem for evaluation, clarification and debugging purposes. Components specific to Dwarf Quest (designated the *Dwarf Quest subsystem)* are required to use the output of the generic subsystem to produce playable dungeons. In addition, we are required to produce internal representations of these dungeons in quick succession, in order to perform measurements for the evaluation (as seen in section 3.7). As with many other procedural generation techniques, the exact same inputs to the system should always produce the exact same output. This includes the seed of the pseudorandom number generator, used as a basis for all relevant algorithmically randomized selections.

In terms of software, we are required to use the C#-based programs: (a) Entika to specify design constraints, and (b) Unity [Unity 05] to create playable Dwarf Quest levels, since Dwarf Quest was developed in it. Their common component is a SQLite-based [SQLite 00] database, which contains the design constraints defined in Entika.

Procedural content generation techniques may take considerable time to be performed. Since generation occurs right before the output is required, (*e.g.* when a player wants to start a new level), we aim to keep the waiting time for potential users short, although this needs to be balanced with the quality of the generated dungeons. In case the waiting time is noticeable, we aim to keep the program active, be it to show progress, or to provide some visual distraction. The best method to keep the program active is to perform (heavy) generation procedures *asynchronously* to the standard

operations of the program. This brings the additional benefit that procedural generation techniques are encapsulated in a separate process, indicating clear boundaries between components. In general, our generation techniques receive input data, perform a (heavy) operation, and return output data. Altogether, this allows us to adopt the model-view-control (MVC) design pattern [Krasner 88] to maintain a clear separation between this data (model), the applied techniques (control), and any visualization (view) constructed from the data.

4.2 System Design

Based on the system requirements, we can distinguish three clearly separated generation components; (i) a generic set of generation procedures, (ii) a set of generation procedures specific to Dwarf Quest which produce internal representations of dungeons, and (iii) another set of generation procedures specific to Dwarf Quest which produce playable dungeons. Fig. 13 displays these components in the system design. Furthermore, the figure shows a clear distinction between the *Entika* system and the *DunGen* system, which exists because (i) they require little overlap from a software design perspective, and (ii) they are required to operate within a different .NET version, 4.0 and 3.5 respectively.

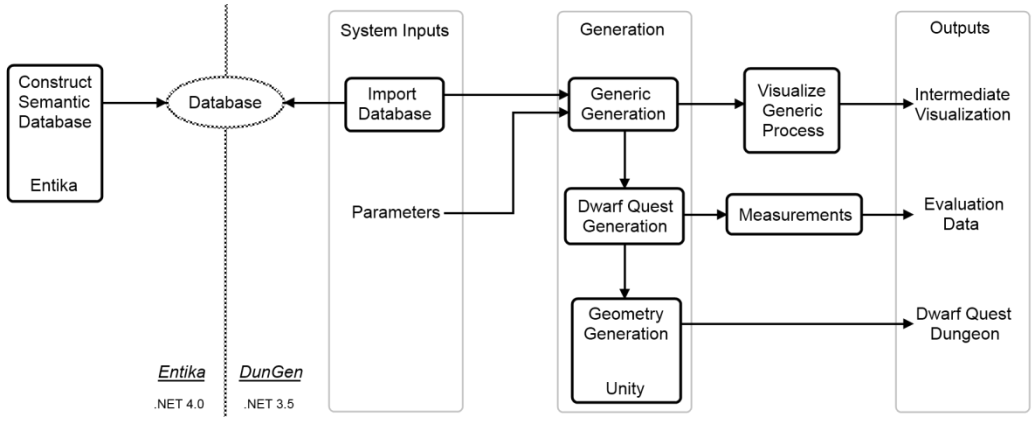


Fig. 13. System design. The Entika and DunGen systems are completely separate systems, connected through a database. The DunGen system consists of several generation components, able to produce (i) intermediate visualizations, used for debugging, (ii) generate dungeons and perform measurements in quick succession, used for the generative space analysis, and (iii) create Dwarf Quest dungeons, which can be played.

Entika allows designers to create abstractions, add relevant semantic information, and indicate relationships between abstractions. Our work expanded on the existing system, allowing designers to specify gameplay abstractions in the form of player actions, parameters, rewriting rules, and their relationships. Entika by default stores information in a relational database, and we extended this database with new required tables to hold our expansions. The database is independent of the .NET version, and therefore serves as an appropriate bridge between the two systems.

The DunGen system has been created specifically for this project, and uses the information from the database to form grammars, which ultimately lead to Dwarf Quest dungeons. This database needs to be imported (*i.e.* loaded into memory) once for a set of generation sessions, typically when booting up a game. The SQLite importing procedure takes considerable time, even when performed asynchronously. Therefore, we created a component that rewrites the database into a simple *xml* file. This file still needs to be imported, but the operation has become hardly noticeable for the user.

Aside the imported database, the second input is a set of parameter values, which correspond to the designer-expressed parameters. These parameters allow the designer, a player, or some algorithm to select a specific set of level features, depending on the designed constraints. While the database only needs to be imported once, the parameter values can be altered in between generation sessions.

The presented system design fulfils the requirements of (i) producing playable Dwarf Quest dungeons based on designer-expressed constraints, (ii) being generic, at least up to and including the *Intermediate Visualization* (Fig. 13), (iii) allowing measurements to be performed in quick succession on internal representations of dungeons, and (iv) the exact same inputs to the system always produce the exact same output.

For this project, we designed five applications in total; (i) (extending) Entika and its database, (ii) a database converter program, (iii) an intermediate visualization application concerning the process of the generic subsystem, (iv) the application that measures dungeon features for the evaluation (as seen in section 3.7), and (v) (extending) Dwarf Quest dungeon generation to produce fully playable dungeons. Screenshots of these applications can be found in Appendix A. We will not detail on their design, as (i) the applications strictly follow the supposed recognisable MVC design pattern, which makes them rather similar in structure, (ii) differences and choices of their design lies in details, which would require an abundance of implementation-related information, and (iii) we believe the system design already provides some high-level concept of their components. Instead, we will detail on the asynchronous design we applied to components of almost each application.

4.3 Asynchronous Design

We adopt a generalized method of dealing with heavy operations, such as procedural content generation. In order not to stall the application during such activities, we perform the operations asynchronously. The regular .NET solution to such a procedure is to use a *background worker*, which provides a background thread to run time-consuming operations. We make use of this class, but determined that it requires an extended structure. Introducing asynchronous operations enriches a program with continued interactivity and possible performance gain, but it also adds additional risk and complexity. We want to reduce both to a minimum, which we pursue by encapsulating all asynchronous operations, *i.e.* creating a clear separation between the background worker and the application activity.

Fig. 14 displays the concept; the *main execution thread* will never communicate with the *worker thread*. We include a *manager* that acts as a façade, which handles universal requests (*e.g. start, cancel*) and applies appropriate operations to the worker thread. A controller making these requests has no need for knowledge about

asynchronous operations. In fact, the manager could be starting the operations synchronously, asynchronously, or relay the operations to a server, without changes to its interface. This abstraction reduces the complexity of the asynchronous operations, since they are now available as simple function calls.

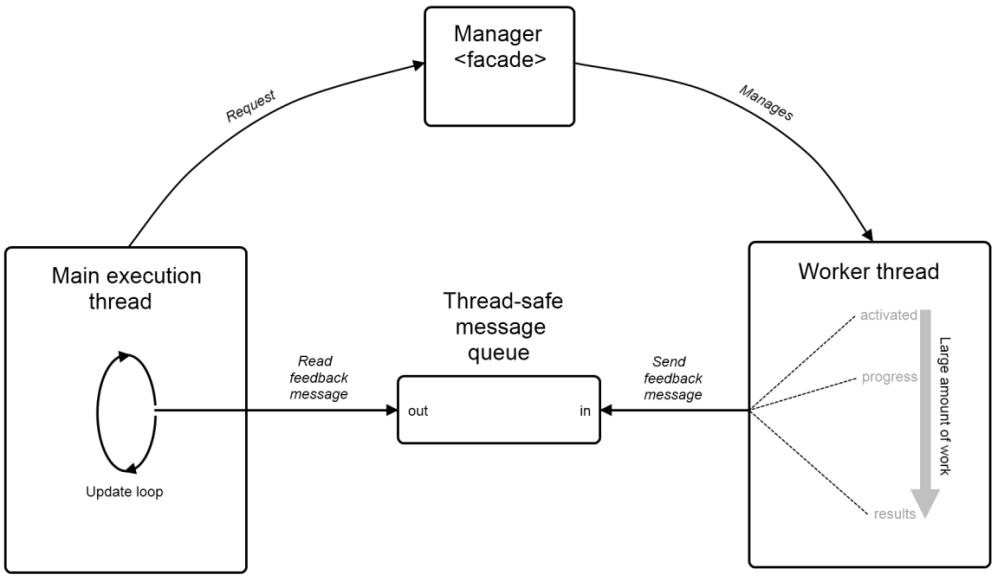


Fig. 14. Asynchronous design. A worker thread is encapsulated to reduce risk and complexity. The result is an interface representing simple function calls with delayed answers. In addition, it is flexible enough to provide intermediate progress updates and error information.

Some communication between the worker thread and the main execution thread is still required. The main execution thread can have an interest in information such as: (i) the results of the operation, (ii) the percentage of completion, to inform users, and (iii) intermediate results and operation descriptions, for visualization and debugging purposes. This communication can be considered one-directional, since the worker thread has no interest in information from the main execution thread. Therefore, we merely require an information dump from the worker thread, without it knowing what happens with that information. We refer to individual information dumps as messages, following the convention from networking. We use a synchronized (*i.e.* thread-safe) version of a queue as the intermediate data structure to store messages, so no direct connection between the two threads is required. The worker thread inserts messages at specific moments during its life cycle, by default these are *start*, *cancel*, *error*, and *successful finish* messages. The specific implementation allows any other message to be sent. The main execution thread can retrieve these messages at its own leisure, typically during an update loop. This further reduces the complexity of the asynchronous operations, essentially representing simple function calls with delayed answers.

The largest remaining risk is that of race conditions, *i.e.* situations in which multiple concurrent operations try to access or change the same instance of data. We solve this by simply not sharing the data between threads; The worker thread receives a clone of the input-data, and returns a clone of the output data. Even though this constrains the

data to be clonable, which requires careful implementation, it mostly removes the risk of race conditions.

An example of the use and flexibility of this structure resides in the application that visualizes the process of the generic subsystem, which is depicted in Appendix A, Fig. 22. One of the purposes of this application is to test a created grammar, both to see whether it works at all, and whether it creates the envisioned action graphs. We provide this form of debugging by visualizing each rewriting step, effectively changing the existing action graph. In this case, the main execution thread maintains the visualization, and handles timer events that allows it to regularly check for messages. The worker thread executes the rewriting procedures, and besides the default messages, reports about each rewriting step it takes in the form of a message. For this example, the benefits of our asynchronous approach are (i) the continued interactivity of the visualization components, *e.g.* buttons are interactive and the view will keep refreshing, (ii) the visualization can already be started before the entire operation is completed on the worker thread, because the steps will consecutively be stored in the message queue during the operation, and (iii) these messages can be read and visualized at different paces, *e.g.* we can directly show the steps, show them at a slower regular interval, or only show steps when the user presses a button. This allows for a solid debugging experience.

This concludes the design of a low-risk and encapsulated approach for performing heavy operations asynchronously. It largely fulfills the requirement that a user does not have to wait long, although this is also dependent on the applied techniques. Its real strength is that even if the waiting time does become noticeable, some visual distraction can be displayed. The operations adopting the asynchronous approach consist of all generation steps, which will be explained in the next chapter, and the database importing procedure. The only time-consuming task that could not be performed asynchronously is the final step, *geometry generation*. This is a limitation imposed by Unity, since it does not support multi-threaded operations.

Chapter 5

Algorithms

This chapter provides an overview of the algorithms applied in our method. We list the different steps we take to go from user input to playable Dwarf Quest dungeons, explaining the concept of the algorithms rather than the specific implementation. The first section contains a visual overview, and the remainder of the sections each explain a single step in more detail. Appendix B provides a visual representation of the progression made in relevant steps for an example Dwarf Quest dungeon, and is best read in conjunction with this chapter.

5.1 Algorithmic Pipeline: Visual Overview

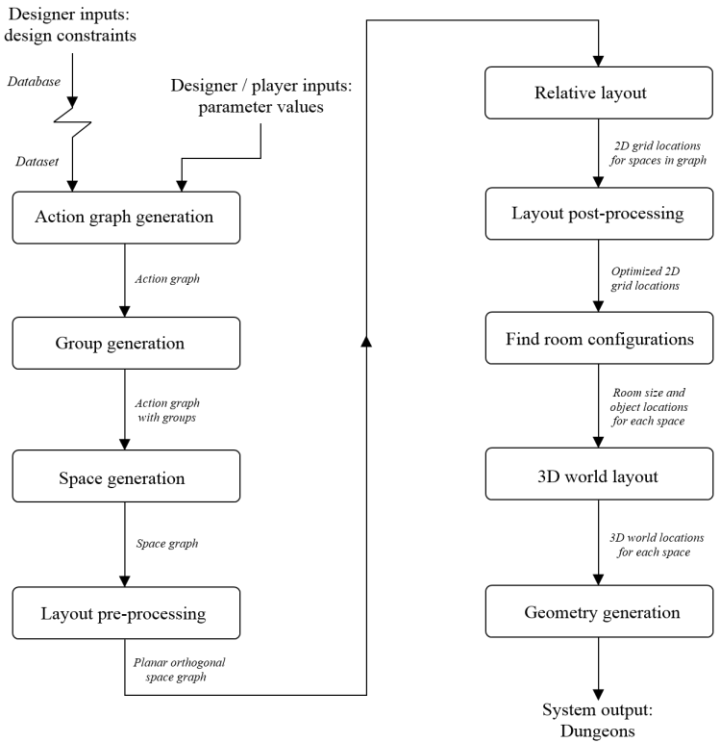


Fig. 15. The algorithmic pipeline. It depicts the steps we take to go from designer-defined constraints to fully playable 3D dungeons.

5.2 Inputs

As stated before, Entika allows designers to express gameplay-based design constraints. Their vocabulary consists of: (i) *player actions*, and their required *target content*, (ii) *parameters*, (iii) *compound actions*, which are the basis of rewriting rules for the grammar, based on relationships between parameters and player actions, (iv) *co-located action relationships*, and (v) *connected pair relationships*. Designers can first introduce such abstractions with an appropriate verb, and then create relationships between these abstractions. Examples of such relationships for Dwarf Quest are depicted in Appendix B, which also shows how they are used in succeeding steps.

The other input consists of a set of parameter values, one for each of the parameters defined in Entika. These were implemented for Dwarf Quest, as depicted in Appendix A, Fig. 23. Other settings are available as well, such as whether to use a seed and which one, and settings with a minor impact on the generators (*e.g.* dead-ends, decorative settings).

5.3 Action Graph Generation

We construct a graph grammar from the compound actions defined in Entika. We do this by assessing the use of the AND and OR operators in rewriting rules, respectively forming consecutive and branching connections. The grammar follows the indicated designer choice (*i.e.* parametric conditions indicate which rewriting options are suitable for selection), and generator choice (*i.e.* a random option can be selected from the remaining options) to establish its rewriting rules. An example of the conversion from compound action to graph grammar rules can be found in Fig. 16.

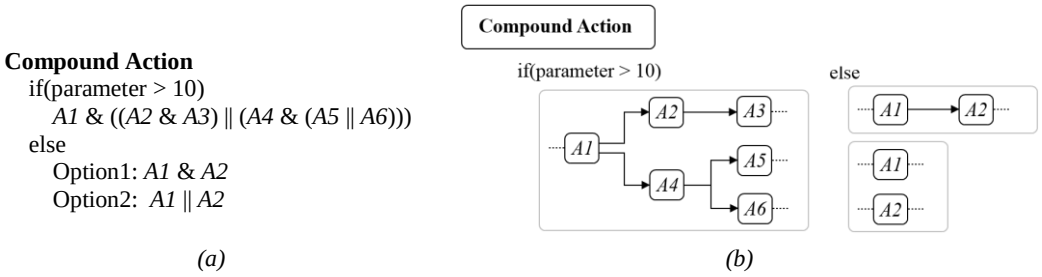


Fig. 16. An example conversion from (a) a compound action definition, to (b) graph grammar rewriting rules.

Action graph generation consists of creating an initial graph from a designer-specified set of initial compound actions, and rewriting any compound action in that graph with an appropriate graph grammar rule until no compound action remains.

5.4 Group Generation

The defined co-located relationships between actions are used to find groups of actions in the graph that must be in the same space. We find groups with the following structures: (i) sequential co-located actions, representing succeeding player actions

targeting the same instance of content, *e.g. Kill Enemy* and *Loot Key From Enemy*, (ii) parallel co-located actions, representing player choice targeting the same instance of content, *e.g. Pay For Key* or *Steal Key*, (iii) overlap between these groups, *e.g.* (*Kill Enemy* and *Loot Key)* or *Steal Key From Enemy*, and (iv) connected pairs, *e.g. Loot Key* and *Unlock Door With Key*, which are not necessarily co-located. This step concludes the set of algorithms from the generic subsystem.

5.5 Space Graph Generation

The groups found in the previous step (except the connected pairs), all contain actions (and thus content) that must be in the same space. We use this information to construct a graph of spaces, where each space contains one or more actions. Spaces are created consecutively for (i) overlapping groups, (ii) remaining parallel action groups, (iii) remaining sequential action groups, and (iv) any remaining action. We then connect the newly created spaces to form a graph. Connections between spaces are found by assessing the unique connections between actions in different spaces. This leads to a graph that looks very similar to the action graph, except that grouped actions now form a single node. The difference is that nodes now represent spaces, which will ultimately contain content for actions, rather than the actions themselves.

For Dwarf Quest we decided that the groups and remaining actions are each appointed to a single space, because rooms in Dwarf Quest are generally quite small. For other projects, further grouping procedures could be incorporated, *e.g.* combining actions that fit well together.

5.6 Layout Pre-Processing

Dwarf Quest levels are required to be in a 2D grid. To lay out the constructed space graph in such a grid, the graph needs to satisfy two requirements; (i) *planarity*, *i.e.* it can be laid out so there are no overlapping edges, and (ii) *orthogonality*, *i.e.* a space can only have four connections in total. We include the requirement that a space can only have three incoming connections, and three outgoing connections, aside the four connections in total. This makes sure our layout algorithm always creates a planar layout.

The space graph may need to be adjusted to become planar and orthogonal. We identified a specific overlap structure, which we refer to as a *web* (Fig. 17a), that causes non-planarity. We resolve webs by placing a new, empty space at the center of the overlap, and then reconnecting the surrounding spaces to it (Fig. 17b). To allow an orthogonal layout, connections of spaces may need to be reduced (Fig. 17c), although the connectivity of the graph should not be compromised. We reduce the number of connections by introducing new, empty spaces that take over connections from spaces with too many connections, and then connect the new space to the original space (Fig. 17d). To find out which connections can safely be combined, we calculate the largest group of connections with the closest common predecessor (in either direction). This algorithm is iteratively applied until no more spaces with too many connections exist, and ensures that the graph remains planar.

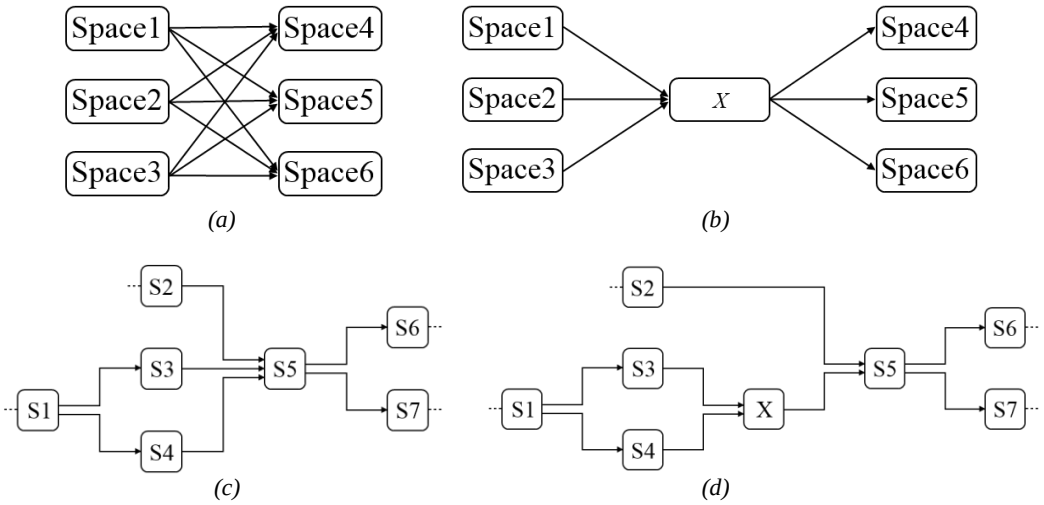


Fig. 17. Layout pre-processing, in the form of planarization and connection reduction. (a) A web, i.e. the non-planar structure we encounter in our generated graphs. (b) Planarization of the web from (a), by introducing new space 'X'. (c) An example graph, where space 'S5' has too many connections; 5 in total. (d) Reducing the connections of 'S5' from (c), by introducing a new space 'X' that takes over the largest group of connections with the closest common predecessor. In this case, 'S3' and 'S4' form the largest group of connections with closest common predecessor 'S1'.

5.7 Relative Layout

To lay out the space graph in a 2D grid, we must assign each space to a unique grid-cell. To achieve this, we calculate the appropriate *x* and *y* value of those cells separately. We calculate the y-value for each space by first determining constraints between spaces; for each space we determine the spaces that must have (i) a smaller y-value, (ii) a larger y-value, and (iii) the same y-value, which we put together and refer to as *equal constraint groups* (ECG), because they share constraints. The determination of such constraints goes hand in hand with the placement of connections between spaces. Connecting space A to the north entrance of space Z, forces constraint yA > yZ, as depicted in Fig. 18a. The placement of connections also determines constraints between the paths of A and Z; since their paths cannot overlap, all the spaces in the path of A now require the *larger than* constraint over all the spaces in the path of Z. Their paths are defined as all the spaces from the space itself to their first common predecessor. Selecting an entrance for each connection depends on (i) entrance availability, (ii) remaining incoming and outgoing connections, and (iii) randomization for remaining options. Note that hallways cannot have corners, because of which we introduce empty corner spaces if required. To reduce the number of corner rooms, we look for specific structures and optimize their layout, as depicted in Fig. 18b,c.

The final set of constraints can be solved to find appropriate y-values for each space, determined per ECG. The algorithm first finds the ECG's with no *smaller than* constraints. The spaces in these groups all receive y-value *zero*. The spaces are then removed from any constraints of other spaces, which reveals new ECG's without *smaller than* constraints. This process continues until no ECG's remain, increasing the y-value each round. Planarity and uniqueness of cells for spaces is ensured by the y-

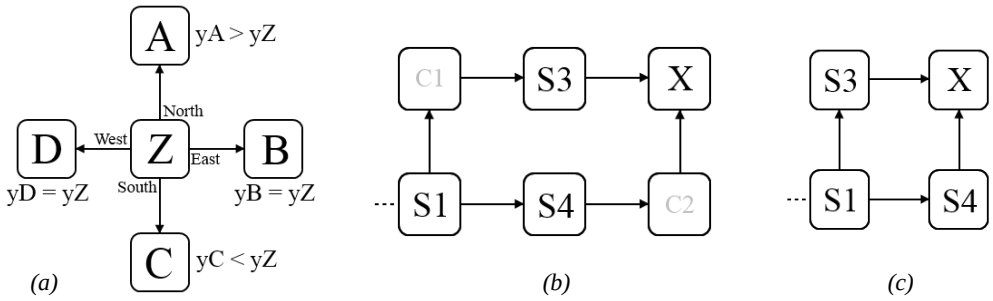


Fig. 18. Laying out the graph orthogonally in a 2D grid. (a) Indicates the relationship between the cardinal direction of an entrance and the y-constraint of the selected connection. (b) Shows how the 'S1-S3-S4-X' structure from Fig. 17d would result in a layout with at least two corner rooms; C1 and C2. (c) The actual layout we adopt for the example from (b), which indicates our effort to reduce the number of corner rooms where possible.

value in conjunction with the x-value.

The x-value is determined by recursively following the selected entrances of connections up the graph. The final x-value of a space is equal to the maximum x-value of the spaces attached to its incoming connections, plus one if that maximum has the same y-value. This rule makes connections to the north and south share their x-value, and connections to the east and west share their y-value, which fulfills the orthogonal requirement.

5.8 Layout Post-Processing

The constraint-based layout process has a focus on planarity and orthogonality, not on compactness. As with other graph-drawing techniques, this may result in long edges (hallways). We try to improve the layout with an algorithm that compresses these hallways. The algorithm adopts some principles from force-directed graph drawing algorithms [Fruchterman 91], and tries to reduce the long edges, *i.e.* hallways, in the graph. Due to the nature of the layout algorithm, the x-values of the spaces are already optimized, and only the compactness of the y-axis can be improved significantly. We do not change connectivity to improve the compactness of the graph, therefore we can work with the constructed ECG's. The compression algorithm tries to move ECG's with a large *pull*, *i.e.* it has a long edge towards one side of the y-axis, towards the direction that reduces this edge size. The algorithm pushes other ECG's out of the way, moves the ECG with large pull to reduce its edge size, and then tries to push the other ECG's back (without pushing the ECG with large pull). This process continues until every ECG has moved, or there is no ECG with a large pull remaining.

Although slightly reduced in size, a fully compressed layout can still have long hallways, which are not supported in Dwarf Quest. The camera's location is static per space (room or hallway), which means the full geometry of such a space needs to be visible. Therefore we fill long hallways with empty rooms. First, we distribute rooms in long hallways in such a way that they are spread out nicely over it, and then we fill in empty cells in the hallway with new rooms.

The current set of layout algorithms do not support dead-end rooms, because we fully connect spaces the way actions were connected. Therefore we add dead-end paths after the layout has been constructed, to at least offer the feature. We also add a special kind of dead-end room in between two paths, which cannot be crossed but allows a glimpse to the other side.

5.9 Matching Room Configurations

To map the spaces from a 2D grid to the 3D virtual world, we require their dimensions, which depends on their internal layout. We provide a list of such internal layout configurations with features: (i) room size, (ii) entrances, and (iii) locations, rotations and sizes for possible objects. These configurations are typically predefined by designers. The higher the number of configurations, the higher the variability of the encountered internal layouts. The target content of actions, as defined in Entika, is used to retrieve room configurations with matching content. In addition, we add decorative content requirements, which are mostly absent in our general approach. The required set of decorative content depends on (i) an overall dungeon theme (*e.g.* prison), which has a cohesive set of room themes, (ii) the selected room theme (*e.g.* torture-room), which has a cohesive set of decorative content options (*e.g.* ball-and-chain), and (iii) whether configurations with such decorative requirements exist. If no match can be found, decorative requirements are reduced, until completely removed.

If more than one matching room configurations exist, we select one from the relevant set based on two criteria: (i) spaces that are not too large for the content, found by creating an appropriate room-size range based on the smallest matching room configuration, and (ii) spaces that have a high *view-blocking-safety*, *i.e.* their content is less likely to block the view of the camera. If multiple configurations remain, we perform a random selection. Both criteria work with a range, which ensures variability for similar content requirements. In addition, room configurations are not rooms: they can have multiple options for object locations, so each configuration by itself can result in somewhat different internal layouts.

5.10 Exact Layout

The rooms can then be laid out in the 3D world. We do this by creating *parcels*, *i.e.* a designated rectangular area for each space, in the virtual world. The parcels receive a length equal to the largest required room length in the set. However, their width is determined by the largest required room width in that specific grid column. This keeps the final layout somewhat compact. The exact location of a room depends on its size. It is semi-centralized in its parcel, depending on an odd or even width or length. This is relevant because entrances in rooms need to be aligned in the final dungeon.

We select locations for objects based on the options in the room configuration. This selection depends on the relationship of the object with the camera, which has a static location within a room. Therefore some large objects (*e.g.* closets, objects attached to the wall) should not be placed in front of it, otherwise they would block the view. We calculate the view-blocking-safety of these objects based on their distance from the location of the camera, and then randomly pick one of the safer options.

5.11 Geometry Generation

The exact layout of the rooms is then used to actually place geometry resembling the determined location, size and configurations. We were able to re-use some of the geometry-instantiation algorithms already adopted by the creator of Dwarf Quest, and added our own where needed. Examples of generated geometry are floors, walls, doorways and their event triggers, and a large variety of objects. A minimap, quite important for navigation in this specific game, is also generated to make the game more playable. In addition, completing a level will kick off a new generation session (possibly with altered parameters), 'endlessly' providing the player with new dungeons. Finally, the content of connected pairs is actually connected, *e.g.* a key will open the door from the associated action.

Chapter 6

Conclusions and Future Work

The research aim of this thesis was to improve on gameplay-based control over procedural level generation. We proposed an approach that enables designers to exercise control over procedurally generated levels by means of a gameplay vocabulary. With our approach, procedurally generated levels can be designed through the expression of gameplay-related design constraints which ultimately steer content generation.

6.1 Conclusions

We surveyed research which showcases 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 effectively be controlled by designers. From this survey, we also conclude that gameplay-based control over procedurally generated levels has recently been investigated successfully, with the use of missions, stories, and player models to control PCG. In addition, we concluded that there are some opportunities to improve on these preliminary contributions.

We proposed an approach which empowers designers to exercise control over procedurally generated levels by means of a gameplay vocabulary, with player actions as the core expression mechanism. Designers can declare design constraints in the form of player actions to be performed in game, their sequencing, relationships, and required target content. Graph grammars, resulting from the designer-expressed constraints, can generate sequences of desired player actions as well as their associated content. These action graphs are used to determine layouts and content for game levels.

Through our case study, we conclude that these design constraints are expressive enough, able to cover a wide generative space of possible *Dwarf Quest* game levels. Furthermore, we conclude that this degree of control is powerful enough to steer generation into distinct sets of desired *Dwarf Quest* level features. This control opens several possibilities of new game design applications, such as (i) a design tool, which allows a selection of procedurally generated levels with specific features to be included in a game, (ii) giving away some of that control to players, with parameters used as game options, (iii) for adaptive games, where the parameters are derived by some algorithm (*e.g.* based on a player model), and (iv) to create highly specialized level generators.

Although we focus on dungeons, we believe our method may still apply to other types of game levels. We believe using player actions to describe gameplay can be

generic enough. As a design concept, controllable by designers, it can be applied to other game levels (*e.g.* buildings, mazes) and genres (*e.g.* platformers, shooters, adventure games).

6.2 Future Work

As for future work, our next step is to evaluate how intuitive this method is for game designers, by conducting user studies and/or interviews. On a longer term, we consider our approach eligible for adaptivity, where level generation is based on the performance of the player. Our grammar parameters, as specified by the designers, can be adjusted between generation sessions. As such, the performance of the player in a single dungeon may define the parameters of the next generated dungeon.

The current set of design constraints could be extended in the future. One of such extensions could be the use of parameter changes in a single level. A practical use for such constraints could be the evolution of difficulty over progress, *i.e.* a selection of increasingly harder challenges the further the player has progressed in a level. This could be done, for example, by allowing the designer to specify parameter evolution in conjunction with player actions during the compound action specification.

Our focus on gameplay, as the vocabulary to design procedurally generated levels, supports control over generated interactive content. However, it does not fully support control over all aesthetic content (*e.g.* decorations). We believe that storytelling would provide an interesting extension atop our action-based vocabulary. Not only is storytelling an even more natural concept for game designers, but they can also capture both gameplay and aesthetic features.

Bibliography

[2KG 12] 2K Games. 2012. "Borderlands 2". Available: *http://www.borderlands2.com/us/*

[Ashlock 11] Ashlock, D.; Lee, C.; McGuiness, C. 2011. "Search-Based Procedural Generation of Maze-Like Levels". *IEEE Transactions on Computational Intelligence and AI in Games* (3): 260-273.

[Bertz 97] Bertz, S.H.; Sommer, T.J. 1997. "Rigorous Mathematical Approaches to Strategic Bonds and Synthetic Analysis Based on Conceptually Simple New Complexity Indices". *Chemical Communications* (24): 2409-2410.

[Bethesda 96] Bethesda Softworks, LLC. 1996. "The Elder Scrolls II: Daggerfall". Available: *http://www.elderscrolls.com/daggerfall/*

[Bethesda 11] Bethesda Softworks, LLC. 2011. "The Elder Scrolls V: Skyrim". Available: *http://www.elderscrolls.com/skyrim/*

[Blizzard 96] Blizzard Entertainment, Inc. 1996. "Diablo". Information: *http://eu.blizzard.com/en-us/games/legacy/*

[Blizzard 12] Blizzard Entertainment, Inc. 2012. "Diablo 3". Available: *http://eu.blizzard.com/en-us/games/d3/*

[Champion 03] Champion, J.; O Sullivan, R.; Champion, C. 2003. "ZedGraph". Available: *http://sourceforge.net/projects/zedgraph/*

[Chomsky 68] Chomsky, N. 1968. "Language and Mind". Harcourt Brace & World, Inc.

[Doran 10] Doran, J.; Parberry, I. 2010. "Controlled Procedural Terrain Generation Using Software Agents". *IEEE Transactions on Computational Intelligence and AI in Games* (2): 111-119.

[Dormans 10] Dormans, J. 2010. "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games". *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*: 1:1-1:8.

[Dormans 11] Dormans, J.; Bakkes, S. 2011. "Generating Missions and Spaces for Adaptable Play Experiences". *IEEE Transactions on Computational Intelligence and AI in Games* (3): 216-228.

[EA 11] Electronic Arts, Inc. 2011. "Battlefield 3". Available: *http://www.battlefield.com/battlefield3/*

[Eidos 09] Eidos Interactive Ltd.; Warner Bros. Interactive Entertainment, Inc. 2009. "Batman: Arkham Asylum". Information: *http://rocksteadyltd.com/games.html/*

[Friberger 12] Friberger, M.G.; Togelius, J. 2012. "Generating Interesting Monopoly Boards from Open Data". *IEEE Conference on Computational Intelligence and Games*: 288-295.

[Fruchterman 91] Fruchterman, T.M.J.; Reingold, E.M. 1991. "Graph Drawing by Force-Directed Placement". *Software - Practice & Experience* (21): 1129-1164.

[Hartsook 11] Hartsook, K.; Zook, A.; Das, A.; Riedl, M.O. 2011. "Toward Supporting Stories with Procedurally Generated Game Worlds". *IEEE Conference on Computational Intelligence and Games*: 297-304.

[Hastings 09] Hastings, E.; Guha, R.; Stanley, K. 2009. "Automatic Content Generation in the Galactic Arms Race Video Game". *IEEE Transactions on Computational Intelligence and AI in Games* (1): 245-263.

[IDV 09] Interactive Data Visualization, Inc. 2009. "SpeedTree". Available: *http://www.speedtree.com/*

[Johnson 10] Johnson, L.; Yannakakis, G.N.; Togelius, J. 2010. "Cellular Automata for Real-Time Generation of Infinite Cave Levels". *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*: 10:1 - 10:4.

[Kessing 12] Kessing, J.; Tutenel, T.; Bidarra, R. 2012. "Designing Semantic Game Worlds". *Proceedings of the 2012 workshop on Procedural Content Generation in Games*.

[Krasner 88] Krasner, G.E.; Pope, S.T. 1988. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". *Journal of Object-Oriented Programming* (1): 26-49.

[Lopes 11] Lopes, R.; Bidarra, R. 2011. "Adaptivity Challenges in Games and Simulations: A Survey". *IEEE Transactions on Computational Intelligence and AI in Games* (3): 85-99.

[Lopes 12] Lopes, R.; Tutenel, T.; Bidarra, R. 2012. "Using Gameplay Semantics to Procedurally Generate Player-Matching Game Worlds". *Proceedings of the 2012 Workshop on Procedural Content Generation in Games*.

[Mawhorter 10] Mawhorter, P.; Mateas, M. 2010. "Procedural Level Generation Using Occupancy-Regulated Extension". *IEEE Symposium on Computational Intelligence and Games*: 351-358.

[Merrel 10] Merrel, P.; Schkufza, E.; Koltun, V. 2010. "Computer-Generated Residential Building Layouts". *ACM Transactions on Graphics* (29): 181:1-181:12.

[Mojang 09] Mojang AB. 2009. "Minecraft". Available: *http://minecraft.net/*

[Muller 06] Muller, P.; Wonka, P.; Haegler, S.; Ulmer, A.; Gool, L.V. 2006. "Procedural Modeling of Buildings". *Proceedings of the 33rd Annual Conference on Computer Graphics and Interactive Techniques*: 614-623.

[Nachmanson 07] Nachmanson, L.; Robertson, G.; Lee, B. 2007. "Drawing Graphs with GLEE". *Proceedings of the 15th International Conference on Graph Drawing*: 389-394.

[Nintendo 11] Nintendo Co., Ltd. 2011. "The Legend of Zelda: Skyward Sword". Available: *http://zelda.com/skywardsword/*

[Perlin 85] Perlin, K. 1985. "An Image Synthesizer". *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* (19): 287-296.

[Rockstar 08] Rockstar Games, Inc. 2008. "Grand Theft Auto IV". Available: *http://www.rockstargames.com/IV/*

[Roden 04] Roden, T.; Parberry, I. 2004. "From Artistry to Automation: A Structured Methodology for Procedural Content Creation". *Proceedings of the 3rd International Conference on Entertainment Computing*: 151-156.

[Smelik 11] Smelik, R.M.; Tutenel, T.; de Kraker, K.J.; Bidarra, R. 2011. "A Declarative Approach to Procedural Modeling of Virtual Worlds". *Computer & Graphics* (35): 352-363.

[Smith 09] Smith, G.; Treanor, M.; Whitehead, J.; Mateas, M. 2009. "Rhythm-Based Level Generation for 2D Platformers". *Proceedings of the 4th International Conference on Foundations of Digital Games*: 175-182.

[Smith 10] Smith, G.; Whitehead, J. 2010. "Analyzing the Expressive Range of a Level Generator". *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*.

[Smith 11a] Smith, G.; Gan, E.; Othenin-Girard, A.; Whitehead, J. 2011. "PCG Based Game Design: Enabling New Play Experiences through Procedural Content Generation". *Second International Workshop on Procedural Content Generation in Games*.

[Smith 11b] Smith, A.M.; and Mateas, M. 2011. "Answer Set Programming for Procedural Content Generation: A Design Space Approach". *IEEE Transactions on Computational Intelligence and AI in Games* (3): 187-200.

[SQLite 00] SQLite Consortium. 2000. "SQLite". Available: *http://www.sqlite.org/*

[Toy 80] Toy, M.; Wichman, G.; Arnold, K.; Lane, J. 1980. "Rogue". Information: *http://en.wikipedia.org/wiki/Rogue_(video_game)/*

[Tremblay 13] Tremblay, J.; Verbrugge, C. 2013. "Adaptive Companions in FPS Games". *Proceedings of the 8th International Conference on Foundations of Digital Games*: 229-236.

[Tutenel 08] Tutenel, T.; Bidarra, R.; Smelik, R.M.; de Kraker, K.J. 2008. "The Role of Semantics in Games and Simulations". *ACM Computers in Entertainment* (6): 1-35.

[Tutenel 10] Tutenel, T.; Smelik, R.M.; Bidarra, R.; de Kraker, K.J. 2010. "A Semantic Scene Description Language for Procedural Layout Solving Problems". *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

[Unity 05] Unity Technologies. 2005. "Unity". Available: *http://unity3d.com/unity/*

[Valtchanov 12] Valtchanov, V.; Brown, J.A. 2012. "Evolving Dungeon Crawler Levels With Relative Placement". *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*: 27-35.

[Valve 09] Valve Corporation. 2009. "Left 4 Dead 2". Information: *http://www.l4d.com/game/html/*

[WCG 12] Wild Card Games. 2012. "Dwarf Quest". Available: *http://www.dwarfquestgame.com/*

[Yannakakis 11] Yannakakis, G.N.; Togelius, J. 2011. "Experience-Driven Procedural Content Generation". *IEEE Transactions on Affective Computing* (99): 147-161.

Appendix A

Developed Applications: Overview

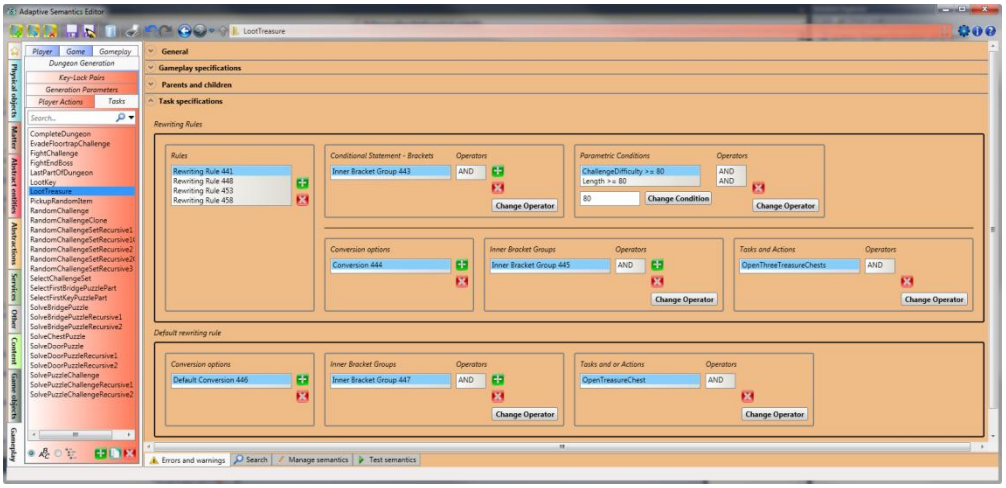


Fig. 19. Expressing design constraints in Entika. The left hand side displays the current list of compound actions. The right hand side allows a designer to define the selected compound action, by specifying relationships between conditional statements (e.g. length > 80 && challengeDifficulty > 80) and player actions (e.g. lootTreasure).

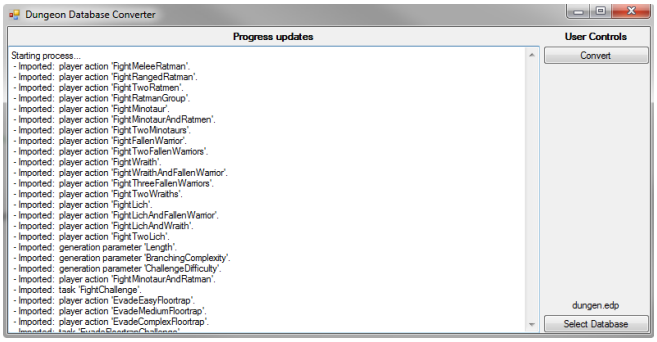


Fig. 20. The database converter. Imports an Entika database, and writes it to an xml file, which can be imported much faster. The application shows the importing progress on the left hand side.

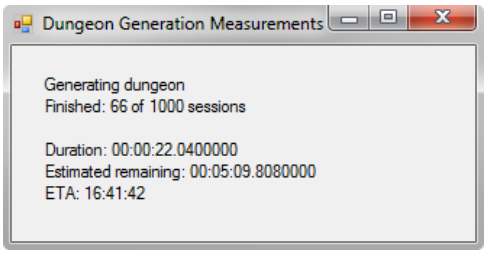


Fig. 21. The measurement application. Performs the generative space analysis measurements as described in section 3.7. It displays some basic information about measurement progress and estimated time of completion. In addition, it produces histogram figures based on its findings. These are generated by extending the ZedGraph [Champion 03] library, which originally does not support 2D histograms, but does provide graph drawing tools.

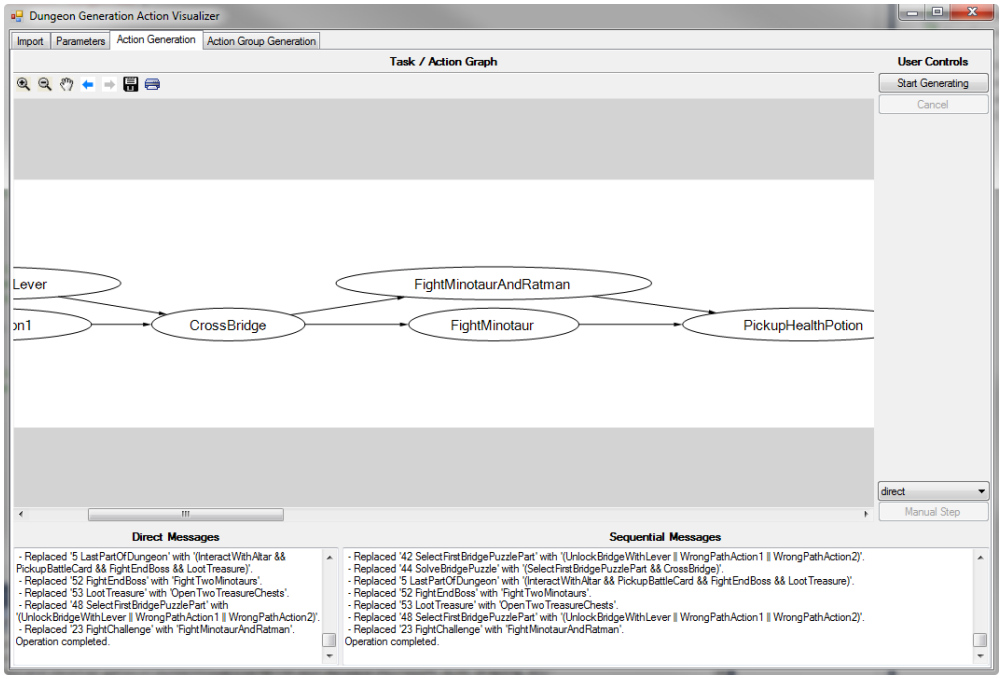


Fig. 22. The intermediate visualizer of the generic subsystem. It provides the following functonality: (i) importing the database defined in Entika, or its xml equivalent, (ii) specifying parameter values and other generation settings, (iii) displaying action graph generation, by showing each rewriting step taking place, (iv) displaying the final generated action graph, partially visible in the center of the figure, (iv) providing additional text-based debug information, partially visible in the lower part of the figure, (v) displaying group generation, also showing each rewriting step, (vi) displaying the final generated groups, and (vii) debugging speeds 'direct', 'interval', and 'manual', which change the pace at which rewriting steps are visualized. The graph visualization has been created with graph-layout library GLEE [Nachmanson 07].

(a)

(b)

(c)

(d)

Fig. 23. The extended Dwarf Quest game. (a) A simple menu interface to select a preset of generation parameter values, matching 'easy', 'normal', or 'hard' difficulty during gameplay. (b) A more extensive menu interface to manually select parameter values and other settings. (c) Gameplay in a generated dungeon. (d) The generated minimap, which provides a layout overview for easier navigation. Both (c) and (d) are not new features, but rather try to imitate regular Dwarf Quest features.

Appendix B

Algorithmic Pipeline Example

B.1 Inputs[3]

Player actions	Required content
Buy upgrades	Upgrade shop
Cross bridge	Bridge
Fight boss	Boss NPC
Fight hard enemy	Enemy NPC (hard)
Fight enemy	Enemy NPC (normal)
Fight easy enemy	Enemy NPC (easy)
Loot key	Key
Loot treasure	Treasure chest
Navigate[4]	< *nothing* >
Open bridge	Lever
Pickup health potion	Cabinet with potion
Unlock door	Door

Co-located relationships
Fight hard enemy - Loot key
Fight enemy - Loot key
Fight easy enemy - Loot key
Fight boss - Loot key

Connected pair relationships
Loot key - Unlock door
Loot key - Loot treasure
Open bridge - Cross bridge

Parameters & session values		
Length	30%	Determines sequential length of dungeon
BranchingComplexity	50%	Determines branches of dungeon
ChallengeDifficulty	40%	Determines difficulty of encountered challenges

Compound actions (small example)

FightChallenge

```
if(ChallengeDifficulty >= 75)
        Fight hard enemy
else if(ChallengeDifficulty >= 25
        Fight enemy
else
        Fight easy enemy
```

SelectBranchingChallenges

```
if(BranchingComplexity >= 66)
        SolveChallenge || SolveChallenge
                || SolveChallenge
else if(BranchingComplexity >= 33
        SolveChallenge || SolveChallenge
else
        SolveChallenge
```

[3] Our created Dwarf Quest grammar supports more actions and information than depicted here, however in this example we keep it to a minimum for clarity.

[4] In our design, we use this action to indicate the player can pick a wrong path, *e.g.* it allows the player to move towards a locked door, before finding its key. It is a short name for action *Navigate through wrong path*.

B.2 Action Graph Generation

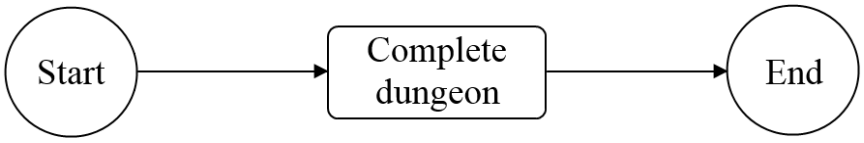


Fig. 24. Initial action graph. The compound action 'Complete dungeon' will be rewritten based on the designer-expressed gameplay constraints. This initial graph is merely an example; it is possible to start with more or different compound actions in the graph.

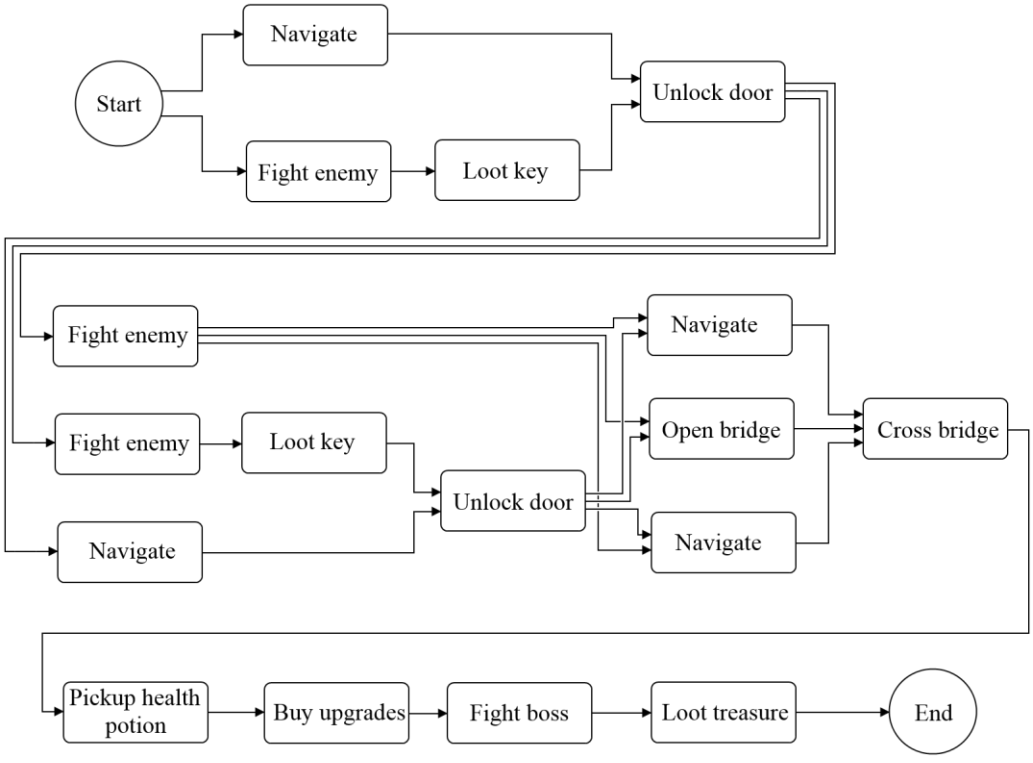


Fig. 25. Example graph, generated with our Dwarf Quest grammar. Originated from Fig. 24.

B.3 Group Generation

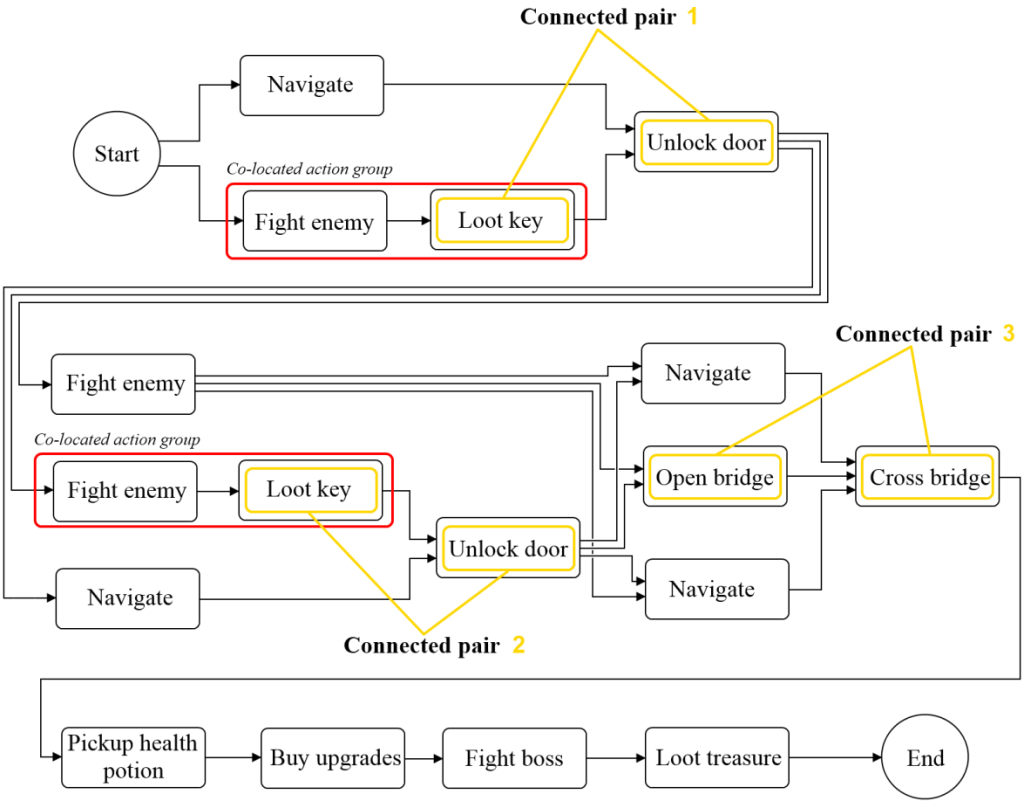


Fig. 26. Determining co-located actions and connected pairs in the action graph.

B.4 Space Generation

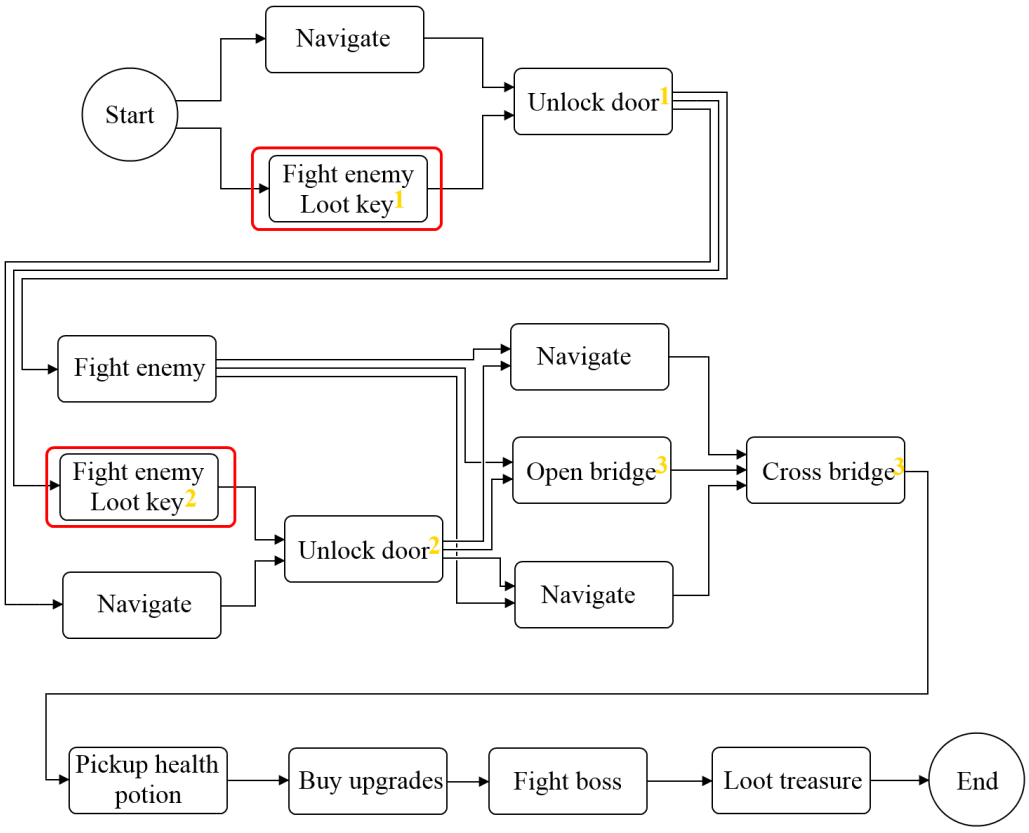


Fig. 27. The action graph converted into a space graph. Action groups are combined into a single space.

B.5 Layout Pre-Processing

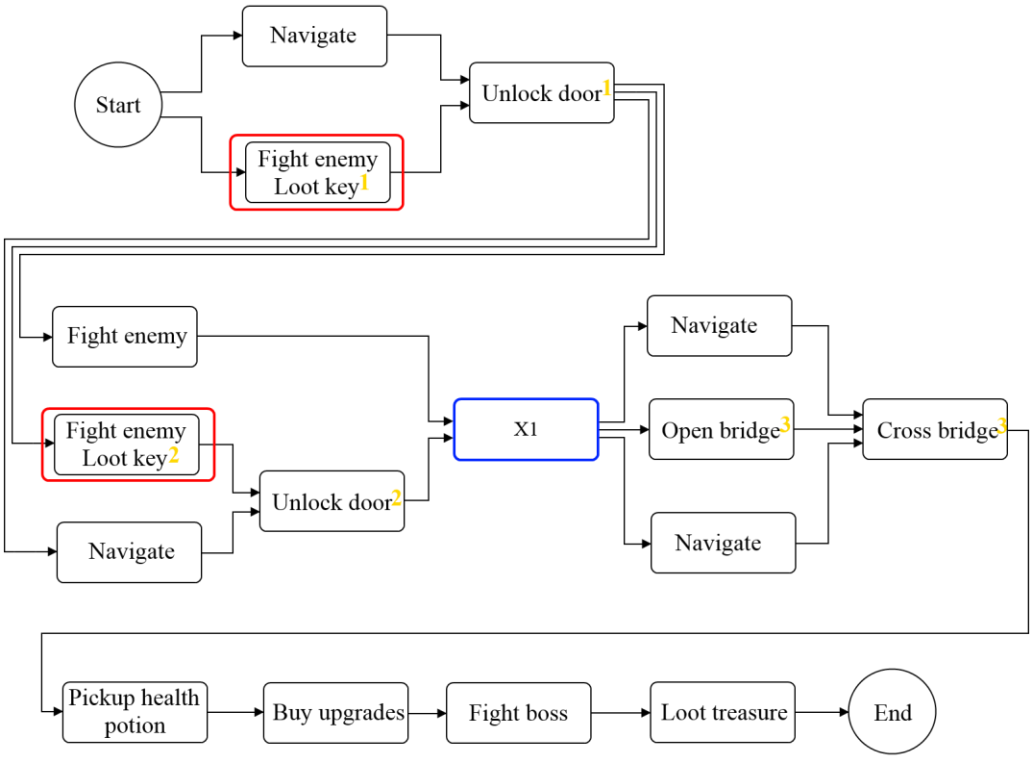


Fig. 28. Planarization of the space graph. Space 'X1' has been added to remove the web in between the five surrounding spaces.

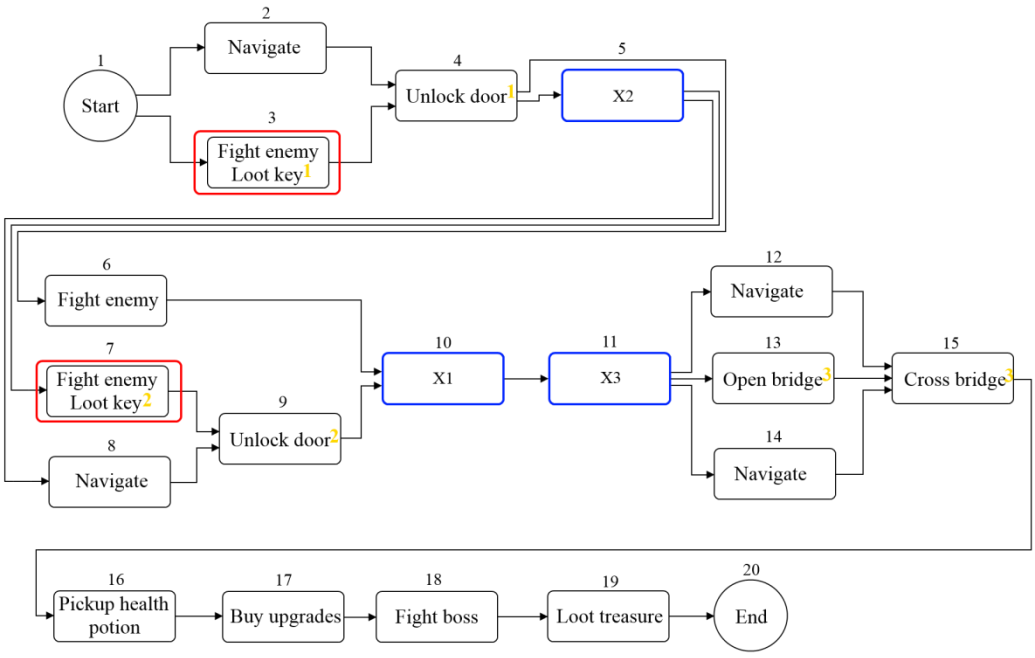


Fig. 29. Preparing for the orthogonal layout requirements by reducing the number of connections to a maximum of 4 in total, or 3 of either incoming or outgoing connections. Spaces 'X2' and 'X3' have been added for this purpose. We reduce the largest set of connections with the closest common predecessor. This explains the connections of space 'X2', which takes over two connections with a closer common predecessor than the original set of three outgoing connections. The spaces have been numbered for a more compact visualization in the next step.

B.6 Relative Layout

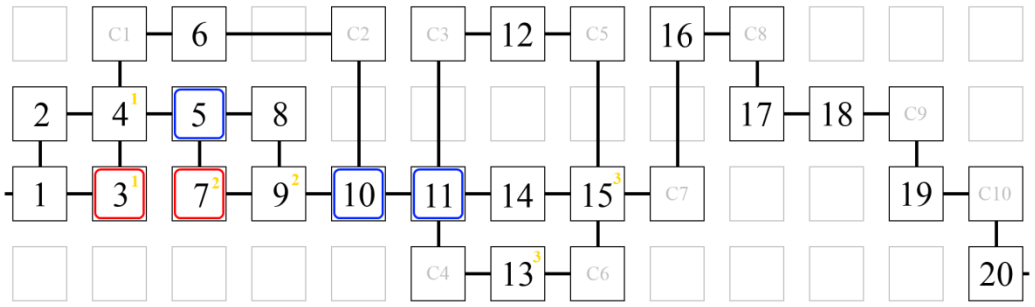


Fig. 30. The spaces laid out in a 2D grid. The y-value of each space is calculated by determining constraints between spaces, based on the selection of entrances for required connections (e.g. connecting 3 to the east-entrance of 1 constrains 1 and 3 to have an equal y-value). The x-value of each space is calculated by following the selected entrances through the graph, making sure a succeeding east-connected space is always at least one higher than its predecessors. Note that 'c-rooms' are empty corner-rooms added during the algorithm because only straight hallways exist.

B.7 Layout Post-Processing

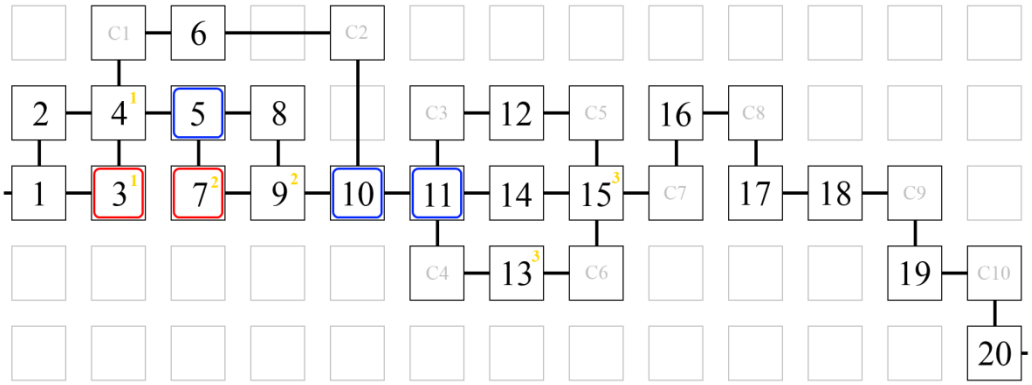


Fig. 31. Applying a force-directed based compression algorithm to make the layout more compact. Equal constraint group (ECG) 'C3-12-C5' were moved down directly. ECG '16-C8' could only be moved down by first pushing the ECG's '17-18-C9', '19-C10', and '20' away from it.

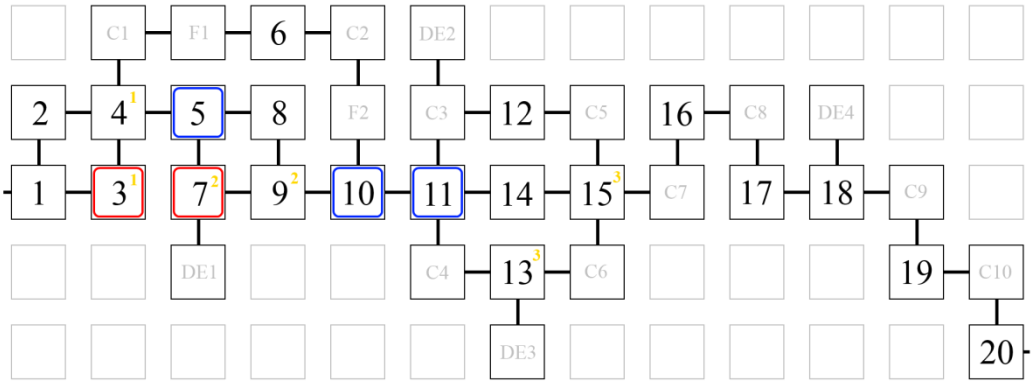


Fig. 32. Finalizing the relative layout: (i) distributing spaces over long hallways (e.g. space 6), (ii) completely removing long hallways by adding empty 'filler' spaces (e.g. F1, F2), and (iii) adding dead-end paths (e.g. DE1-DE4).

B.8 3D World Layout

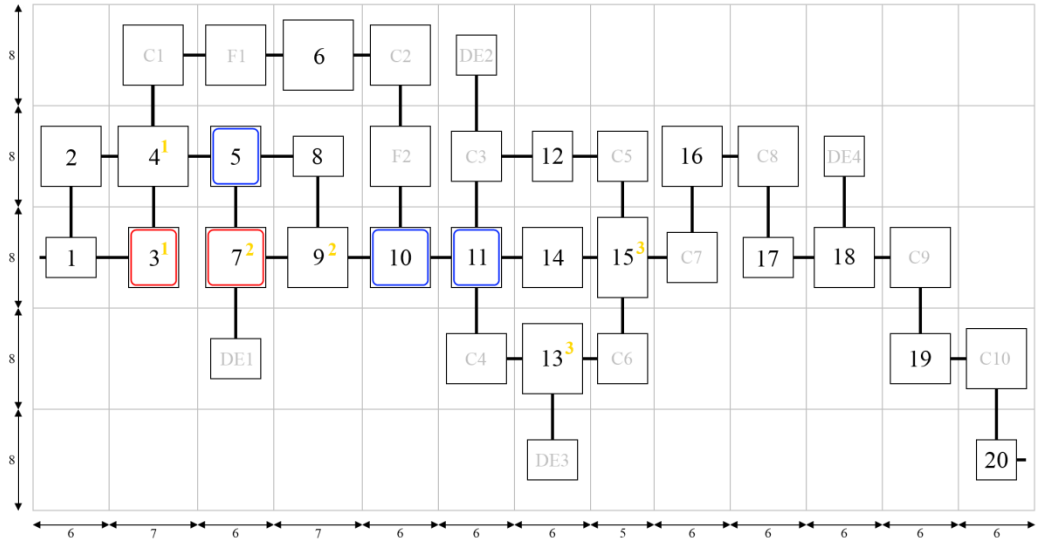


Fig. 33. Using the sizes from the matched room-configurations to determine the 3D world layout. Parcel sizes (rectangular spaces for the rooms) are calculated by: the longest room length of all rooms (parcel length), and the longest room width per grid column (parcel width).

B.9 Geometry Generation

Fig. 34. The 3D world layout of the dungeon, here realized with Dwarf Quest geometry.

(a)

(b)

Fig. 35. Examples of generated Dwarf Quest room geometry. (a) The 'Cross Bridge' room with ID 15, here displayed with theme 'Banner Room'. The central gap in the floor blocks player progress; a bridge moves into place when the player switches a lever. (b) An example 'Fight Boss' room, here displayed with theme 'Torture Room'. It is similar to the room with ID 18, but has a different theme for demonstration purposes.