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

There is an increasing demand to improve the procedural generation of 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.

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 and Mateas 2011), (ii) the increased variability of the generated content (Hastings, Guha, and Stanley 2009; Smith *et al*. 2011), and (iii) its use to support player-centered adaptive games (Lopes and Bidarra 2011; Yannakakis and Togelius 2011). 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 *et al*. 2011).

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 *gameplay grammar*, resulting from designer-expressed constraints, generates graphs 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 sequences 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 gameplay 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.

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, Yannakakis, and Togelius (2010) use the self-organization capabilities of cellular automata to generate natural and chaotic infinite cave levels. For platform games, Mawhorter and Mateas (2010) 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 and Brown (2012) 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 and Parberry (2004) 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*. (2009) 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*. (2011) 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' (2010, 2011) work on grammars to generate dungeons shares most similarities with our research. Missions are generated through a graph grammar, representing sequential player tasks. This mission graph is then used by a shape grammar to create a corresponding game space. Recipes are created by designers to instruct the generator how to re-write grammar rules. Our generic approach aims for a wider range of application, to both games and genres, than done before. The main distinct and novel contributions are: (i) tasks can be created from scratch, with associated game content, (ii) parameters to control the generative grammar can be freely specified and manipulated, and (iii) additional spatial relationships, beyond "key-lock" pairs, can be specified and controlled.

Gameplay Grammars

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 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, *i.e.* a gameplay 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.

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, Tutenel and Bidarra 2012), 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 grouped and specified as a compound, where the whole composition of sub-actions is represented by a single name. For example, as seen below, *acquiring a key* can be fulfilled by *killing an enemy* and then *looting the key from its body*. Furthermore, each sub-action can itself be a single 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: *Kill an enemy → Loot key from body*

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

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 and generation time, and not at game time, several options for fulfilling a compound action may be specified. Selecting one 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 global parameter values to evaluate against those conditions. For example, and as shown below for re-writing *Acquire key*, if values for Difficulty and Length are met, then option 1 is chosen, with random selection between option 1.1 and 1.2.

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

Expressing all these design constraints enables designers to author how gameplay should 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, and looting a key from his body), and (b) *semantically connected* action pairs (*e.g.* a key and its lock, like in Dormans' work).

Action co-location is 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, this does not preclude that, as mentioned above, some actions *must* be together in the same space, whatever that space may be. For example, two individual actions targeting the same object instance. This object exists in a single space, and therefore those actions are also required to be in that 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.

Graph Generation

Once a designer has expressed all design constraints, they result in an instance of a gameplay 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 their order. Eventually, this information in the graph will determine a game level layout.

Different sets of designer-specified constraints result in distinct gameplay grammars, which is a first way of controlling generation. Additionally, generation can also be controlled by setting the initial parameter values of a single grammar which, as explained before, will steer the selection of the corresponding 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 sub-graph of linked actions. 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 no conditions are set)
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. Aggregating 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 occur 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. Figure 1(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 *e.g.* their target content, the groups where some actions must occur in the same space, as well as semantically connected action pairs.

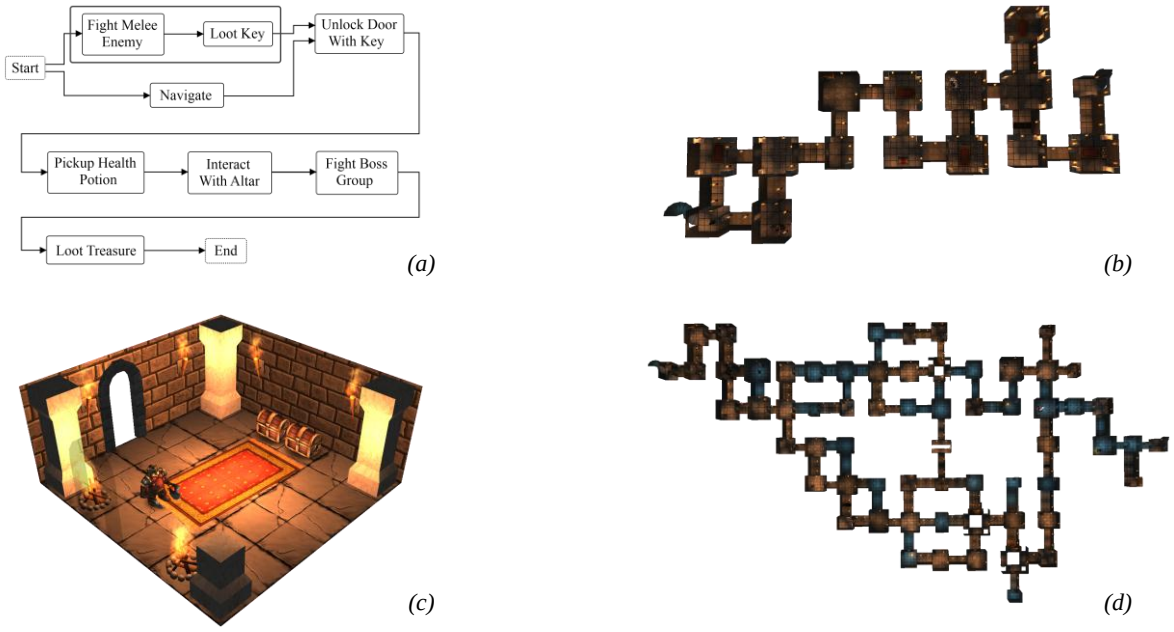

Fig. 1. (a) Graph of player actions for an example generated dungeon, (b) Dwarf Quest dungeon layout, generated for (a), (c) 'Loot Treasure' room, generated for (a), (d) another (unrelated) example of a dungeon layout (colors are only decorative).

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, once created, player actions and related design constraints can be 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 and 'portable' 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 level 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 *et al*. 2010).

For our case study, we used Dwarf Quest[1], a typical dungeon crawler game in which 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 a gameplay grammar, into dungeon levels. Several Dwarf Quest features were essential for constraining this dedicated generator. First, the rooms and hallways have to be orthogonally placed on a 2D grid, 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. Full details on the algorithms of the dedicated generator are outside the scope of this paper and can be found elsewhere (Van der Linden 2013).

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

1. *space assignment* converts nodes of the action graph into rooms and edges into hallways;
2. *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);
3. *layout solving* converts the planar graph into an orthogonal graph mapped onto the 2D grid map;
4. *layout post-processing* still needs to optimize the resulting layout. As with other orthogonal planar graph drawing techniques, excessively long edges (*i.e.* hallways) are a side-effect. Long hallways are then compressed, and rooms added into the ones which cannot be further compressed.

[1](Wild Card Games) http://www.dwarfquestgame.com/

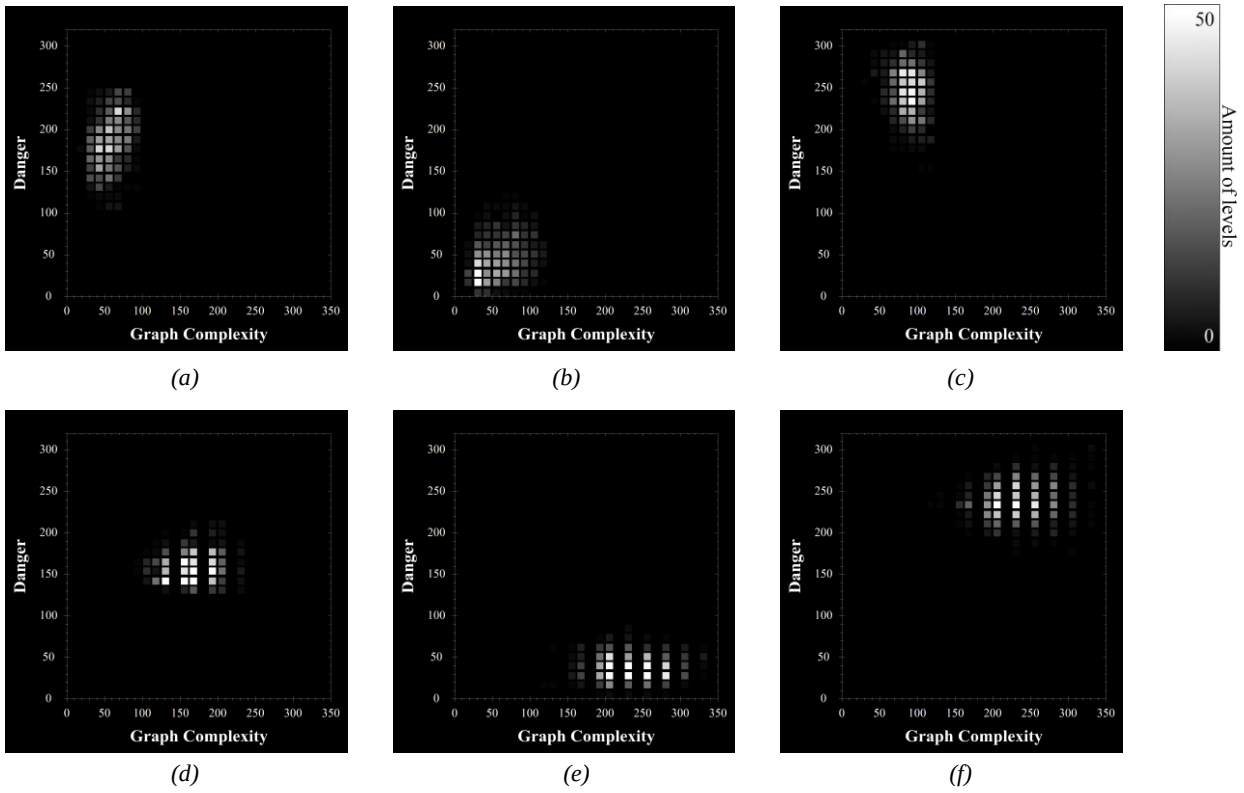

Fig. 2. Histograms for graph complexity and danger, measured for 1000 generated dungeons. Graph complexity is the number of sub-graphs in the final level layout. Danger is the total amount of damage points a level can inflict. Results are displayed for: (a) a grammar without parameters. Another grammar 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 times for the levels generated in (b) and (f) are 1 and 2 seconds, respectively.

Finally, the geometry of rooms, hallways, objects and NPCs 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, so that *e.g.* a lever actually lowers a closed bridge. Figure 1 displays examples of generated dungeons (b, d), and one of their rooms (c).

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 generation 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 and Whitehead (2010).

The generative space can be shaped by our control tools, *i.e*. the gameplay 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 and Whitehead 2010). 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 face. Previously used for molecular complexity in chemistry (Bertz and Sommer 1997), for our purposes, graph complexity is the number of subgraphs of the final 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 and Verbrugge 2013), 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. For our purposes, it is calculated by summing the average amount of damage dealt to the player, for all damage-dealing components.

The histograms in figure 2 show the measurements performed on dungeons from two different grammars. For each histogram, 1000 dungeons were generated. Figure 2(a) plots generated levels from a first grammar, featuring no parameters. This grammar yields a dungeon with a simple structure, each challenge 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 allowing for 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.

Figures 2(b) through 2(f) 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 figures 2(b) – 2(f), different input parameter values were used to generate levels. This resulted in the following dungeon features: (2b) a simple structure and minor danger, (2c) a simple structure and very high danger, (2d) a medium complex structure and medium danger, (2e) a complex structure and low danger, and (2f) 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 figures 2b through 2f always with abundant 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 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 some player model (Lopes *et al.* 2012).

Conclusions and Future Work

We proposed an approach that enables designers to exercise fine-grained control over the procedural generation of game levels by means of a gameplay vocabulary. With our approach, procedurally generated levels can be *designed* by specifying a *gameplay grammar*, expressed in terms of 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 precisely steer generation into distinct sets of desired *Dwarf Quest* level features. This control opens up a variety of possibilities of new game design applications. 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. 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, once specified by the designers, can be adjusted between generation sessions. As such, the performance of the player in a single dungeon may determine the parameter values for 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 it can also capture both gameplay and aesthetic features.

In short, gameplay grammar-based level generation is already quite expressive and powerful to significantly improve the design of procedurally generated levels.

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

References

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.

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, J. 2011. Level Design as Model Transformation: a Strategy for Automated Content Generation. *Proceedings of the 2011 Workshop on Procedural Content Generation in Games*: 2:1-2:8.

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.

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.

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

Van der Linden, R. 2013. Designing Procedurally Generated Levels. MSc Thesis, Delft University of Technology.

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

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

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, 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.

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, 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.

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

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*.

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.

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