Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games

Joris Dormans
Hogeschool van Amsterdam
Weesperzijde 190
1097DZ Amsterdam, The Netherlands
+31 20 595 1686

j.dormans@hva.nl

ABSTRACT

This paper investigates strategies to generate levels for action adventure games. This genre relies more strongly on well-designed levels than rule-driven genres such as strategy or roleplaying games for which procedural level generation has been successful in the past. The approach outlined by this paper distinguishes between missions and spaces as two separate structures that need to be generated in two individual steps. It discusses the merits of different types of generative grammars for each individual step in the process.

Keywords

Procedural generation; level design; action adventure games.

1. INTRODUCTION

Games with procedurally generated content have been around for some time. The classic example of this type of game is *Rogue*, an old *Dungeons & Dragons* style ASCII dungeon-crawling game which levels are generated every time the player starts a new game. The source code of newer games with procedural content is generally not accessible, so it is hard to determine what type of algorithms are used in the creation of their levels. The source code of older games is available and their level generating strategies are well documented on the internet. The typical approach of these games can be classified as a brute-force random algorithm that is tailored to the purpose of generating level structures that function for the type of game. One strategy is to generate a tile map that is filled with tiles representing solid rock and to 'drill' tunnels and rooms into the map starting from an entrance. Multiple paths can be created by drilling into new directions from previously created locations. The dungeon is then populated with creatures, traps and treasures [1]. Another strategy involves zoning the dungeon into large tiles, generate dungeon rooms in some of these zones in the next step, and finally connecting the rooms with a network of corridors [2]. To create game space to represent wilderness areas cellular automata are used to generate more organic structures [3].

Although these algorithms have a proven track-record for the creation of roguelike games, the gameplay their output supports is rather limited. A typical major component of the gameplay of roguelike games is character building. This type of gameplay, which stems directly from a rather mechanistic interpretation of pen-and-paper roleplaying, resolves for a large part around gathering experience points and magical equipment to improve the main character. As game designer Ernest Adams points out in his satirical 'letter from a dungeon', there seems to be little purpose behind these mechanics, resulting in a shallow representation of character growth as a faint echo of the mythical quest [4]. Gameplay of this type, although forming a viable niche of its own, is well suited for a random dungeon layout and does not require the same level of level-design quality as, for example, an action adventure game from the *Zelda* series in which this style of character development plays only a little part, as is mentioned in an interview by Shigeru Miyamoto, the series main designer [5]. Just as the random encounter table is an appreciated tool to facilitate a particular style, but not all styles, of role-playing in *Dungeons & Dragons* [6].

It is in a similar light that Kate Compton and Michael Mateas point out that generating levels for an action platform game is more difficult as level design is a far more critical aspect of that type of game [7]. Action adventures rely on level design principles that result in enjoyable exploration, flow and narrative structure, too. As it turns out, these principles are difficult to implement with the algorithms commonly encountered in roguelike games. These algorithms generally cannot express these principles as they mostly operate on a larger scale than the scale of individual dungeon rooms and corridors. In order to generate game levels informed by such principles we need to turn to a method that does operate on the scale on which these principles reside. This method is the use of generative grammars.

However, even with the use of generative grammars, generating good levels is still very hard. Levels often have a random feel to it and tend to lack overall structure. To search simply for a single generative grammar to tackle all these problems is not going to work. Well-designed levels generally have two, instead of one structures; a level generally consists of a *mission* and a *space*. This paper suggests that both missions and spaces are best generated separately using types of generative grammars that suit the particular needs of each structure. As outlined in the final sections of this paper, the route presented here is to generate missions first and then generate spaces to accommodate these missions.

2. MISSIONS AND SPACES

In a detailed study of the level design of the Forest Temple level of *The Legend of Zelda: The Twilight Princess*, conducted by the author and described in more detail elsewhere [8], two different structures emerge that both describe the level. First, there is the

PCGames 2010, June 18, Monterey, CA, USA

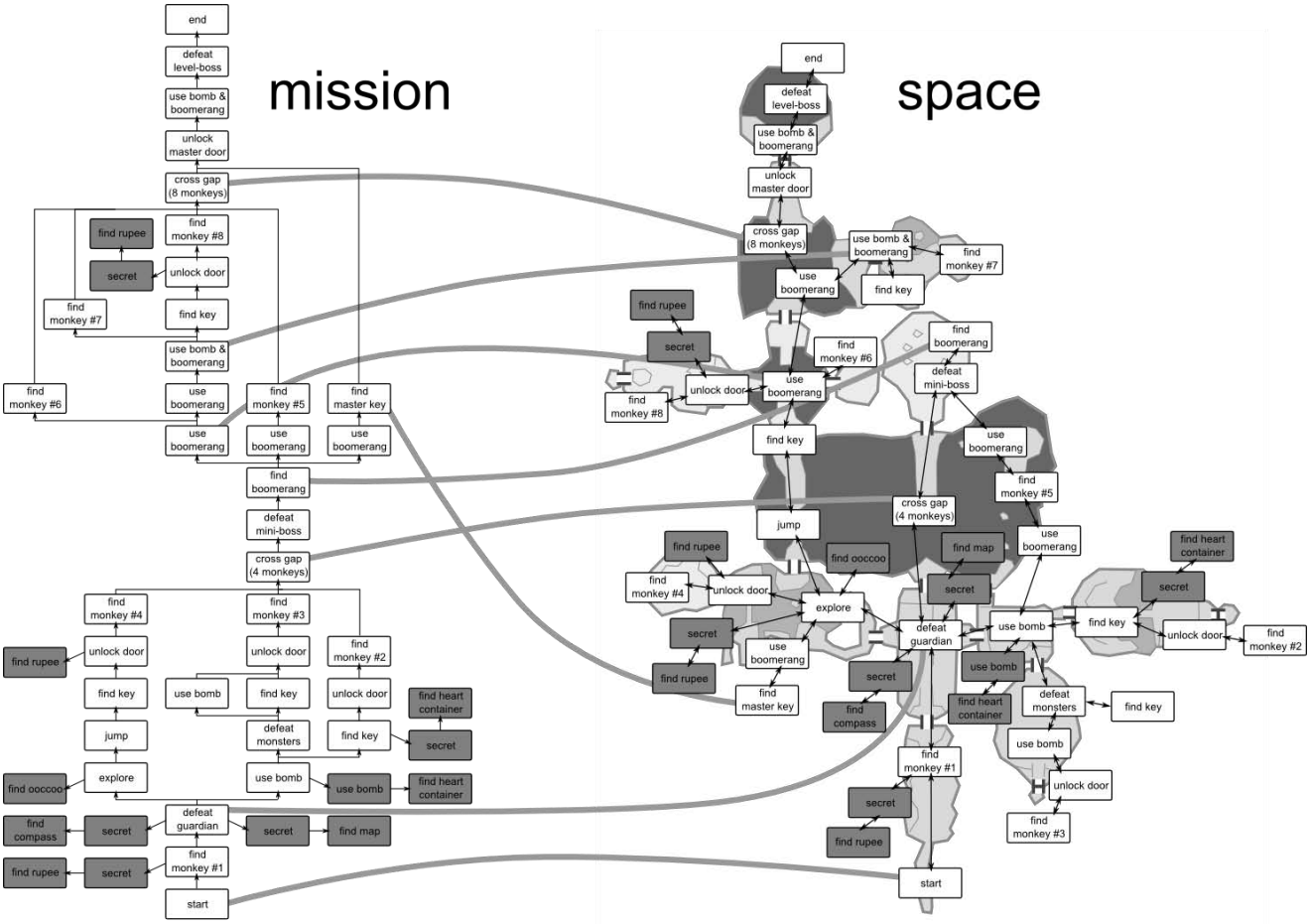

Figure 1. Mission and space in the Forest Temple level of *The Legend of Zelda: The Twilight Princess*

geometrical lay-out of the level: the space. Level space can be abstracted into a network of nodes and edges to represent rooms and their connections. Second, there is the series of tasks to player needs to complete in order to get to the end of the level: the actual mission. The mission can be represented by a directed graph indicating which tasks are made available by the completion of a preceding task. The mission dictates a logical order for the completion of the tasks, which is independent of the geometric lay-out. As can been seen in figure 1, the mission can be mapped to the game space. In this case certain parts of the space and the mission are isomorphic. In particular, in the first section of the level mission and space correspond rather closely. Isomorphisms between mission and space is frequently encountered in many games, but the differences between the two structures are often just as important.

Level space accommodates the mission and the mission is mapped onto the space, but otherwise the two are independent of each other. The same mission can be mapped to many different spaces, and one space can support multiple different missions. The principles that govern the design of both structures also differ. A linear mission, in which all tasks can only be completed in a single, fixed order, can be mapped onto a non-linear spatial configuration. Likewise, a non-linear mission featuring many parallel challenges and alternative options, can be mapped on to a strictly linear space, resulting in the player having to travel back and forth a lot.

Some qualities of a level can be attributed to its mission while others are a function of its space. For example, in *Zelda* levels, and indeed in many Nintendo games, it is common strategy to train the player in the available moves and techniques using a structure that is also found in martial arts training [9]. Following this structure a player first learns a simple technique in isolation (the *kihon* stage), then she repeats the technique in order to perfect it (the *kihon-kata* stage). In practicing martial arts this repetition can be long and tedious; an excellent example of this can be found in the film *Karate Kid* where the hero practices his skills to perfection by performing the same task over and over again ("wax-in, wax-out"). Next, the player learns how different techniques can be combined (the *kata* stage) before her real skills are tested in a boss fight (the *kumite* stage). This structure can be witnessed in the Forest Temple level. In this level Link first learns how to use 'bomblings' to attack creatures and unblock passages (*kihon*), he must repeat this feat a couple of times in order to progress (*kihon-kata*). He also obtains a special boomerang which he learns to use in similar series of relative simple tasks (*kihon*, *kihon-kata*). Towards the end Link must combine bomblings and his boomerang in order to get to the last monkey, which he needs to reach the last rooms in the temple (*kata*), where he must to use the same techniques to defeat the final level-boss (*kumite*).

At the same time, the mission in the Forest Temple also follows a similar structure that is often found in Hollywood films and that can ultimately be attributed to Joseph Campbell's monomyth (see

[10] & [11]). Following this structure, the player crosses into the realm of adventure (the dungeon) after a confrontation with a threshold guardian. Around halfway or two-thirds into the level the player defeats a mid-level boss and obtains the boomerang signaling the start of the third and final act which ends with the defeat of the level boss. The spatial qualities of the Forest Temple are different. Its basic layout follows a hub-and-spoke layout that provides easy access to many parts of the temple. The boomerang acts as key to many locks that can be encountered right from the beginning. Once it is obtained extra rooms in the temple are unlocked for the player to explore, a structure frequently found in adventure games [12].

3. GENERATIVE GRAMMARS

Generative grammars originate in linguistics where they are used as a model to describe sets of linguistic phrases [13]. In theory, a generative grammar can be created that is able to produce all correct phrases of a language. A generative grammar typically consists of an alphabet and a set of rules. The alphabet is a set of symbols the grammar works with. The rules employ rewrite operations: a rule specifies what symbol can be replaced by what other symbols to form a new string. For example: a rule in a grammar might specify that in a string of symbols, symbol 'S' can be replaced by the symbols 'ab'. This rule would normally be written down as 'S → ab'. Generative grammars typically replace the symbol (or group of symbols) on the left-hand side of the arrow with a symbol or group of symbols on the right-hand side. Therefore, it is common to refer to the symbols to be replaced as the left-hand side of the rule and to refer to the new symbols as the right-hand side. Some symbols in the alphabet can never be replaced because there are no rules that specify their replacement. These symbols are called terminals and the convention is to represent them with lowercase characters. The symbols 'a' en 'b' in the last example are terminals. Non-terminals have rules that specify their replacement and are conventionally represented by uppercase characters. The symbol 'S' from the previous rules is an example. For a grammar that describes natural language sentences, terminal symbols might be words, whereas non-terminal symbols represent functional word groups, such as noun-phrases and verb-phrases. The denominator 'S' is often used for a grammar's start symbol. A generative grammar needs at least one symbol to replace; it cannot start from nothing. Therefore, a complete generative grammar also specifies a start symbol.

Grammars like these are used in computer science to create language and code parsers; they are designed to understand and recognize language. However, grammars are also suited to generate language. It is easy to see that simple rules can produce quite interesting result especially when the rules allow for recursion: when the rules produce non-terminal symbols that can directly or indirectly result in the application of the same rule recursively. The rule 'S → abS' is an example of a recursive rule and will produce endless strings of ab's. The rule 'S → aSb' is another example and generates a string of a's followed by an equal number of b's. Generative grammars developed for natural languages are capable of capturing concepts that transcend the level of individual words, such as argument construction and rhetoric, which suggests that generative grammars developed for games should be able to capture higher level design principles that lead to interesting levels at both micro and macro scopes.

Generative grammars can be used to describe games when the alphabet of the grammar consists of a series of symbols to represent game specific concepts, and the rules define sensible ways in which these concepts can be combined to create well-formed levels. A grammar that describes the possible levels of an adventure game, for example, might include the terminal symbols 'key', 'lock', 'room', 'monster', 'treasure'. While the rules for that grammar might include:

1. Dungeon → Obstacle + treasure
2. Obstacle → key + Obstacle + lock + Obstacle
3. Obstacle → monster + Obstacle
4. Obstacle → room

In this case, when multiple rules specify possible replacements for the same non-terminal symbol, only one rule will be selected. This can be done (pseudo-)randomly. The rules can generate a wide variety of strings including:

1. key + monster + room + lock + monster + room + treasure
2. key + monster + key + room + lock + monster + room + lock + room + treasure
3. room + treasure
4. monster + monster + monster + monster + room + treasure

The strings produced by the grammar discussed above are not all suited for a game level. Especially string 3 is far too short even in the limited example above. The problem is not with generative grammars as such but the quality of the rules used in the example. In fact generative grammar can easily counter these problems by creating rules that capture level design principles better, such as:

1. Dungeon → Obstacle + Obstacle + Obstacle + Obstacle + treasure
2. Dungeon → Threshold Guardian + Obstacle + Mini-Boss + reward + Obstacle + Level-Boss + treasure.

Where rule 1 incorporates the idea that a dungeon needs to have a minimal length to be interesting at all, and rule 2 directly incorporates a three act story structure like the one described for Forest Temple level of *Zelda: The Twilight Princess* above.

Generative grammars can be used in different ways to produce content for games. Game experts and designers can produce a grammar to generate content for a particular game. Drafting such a grammar would by no means be an easy task, but the initial effort vastly outweighs the ease by which new content can be generated or adjusted. Furthermore, grammars and procedurally content can be used to aid the designer by automating some, but not all, design tasks. This approach was taken by Epic Games for the generation of buildings and large urban landscapes. It proved to be very versatile as it allowed designers to quickly regenerate previous sections with the same constraints but with new rule sets without having to redo a whole section by hand [14]. Finally, it would be possible to grow grammars using evolutionary algorithms that select successful content from a test environment. The grammars presented in this paper were all drafted using the first method. Evolutionary grammars, although a tantalizing concept, are beyond the scope of the material presented here.

Relevant applications of generative grammars can also be found in with Lindenmayer Systems (L-Systems). Lindenmayer was a biologist who used grammars to describe the growth of plants, but L-Systems have been applied to generate many different spatial outputs [15]. L-Systems are used today in games to generate trees and other natural structures. L-Systems have been extended for

the procedural generation of city models [16]. This extension serves to create looped networks of roads, where original L-Systems only generate tree-structures. The extension allows a street that is generated close to a previously generated street to intersect the latter, and thus create a loop back to the previously generated structure.

4. GRAPH GRAMMAR TO GENERATE MISSIONS

Graph grammars are discussed in relation with level generation by David Adams in his 2002 Bachelors thesis *Automatic Generation of Dungeons for Computer Games* [17]. Graph grammars are a specialized form of generative grammars that does not produce strings but graphs consisting of edges and nodes. In a graph grammar one or several nodes and interconnecting edges can be replaced by a new structure of nodes and edges (see figures 2 & 3; [18]). After a group of nodes have been selected for replacement as described by a particular rule, the selected nodes are numbered according to the left-hand side of the rule (step 2 in figure 3). Next, all edges between the selected nodes are removed (step 3). The numbered nodes are then replaced by their equivalents (nodes with the same number) on the right-hand side of the rule (step 4). Then any nodes on the right-hand side that do not have an equivalent on the left-hand side are added to the graph (step 5). Finally, the edges connecting the new nodes are put into the graph as specified by the right-hand side of the rule (step 6) and the numbers are removed (step 7). Note that graph grammars can have operations that allow existing nodes to be removed, these operations are not used in this paper.

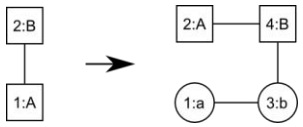

Figure 2. An example of a graph grammar rule

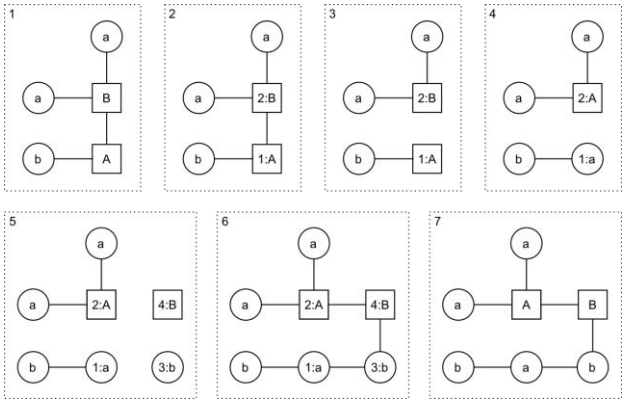

Figure 3. The replacement operations according the rules from figure 2.

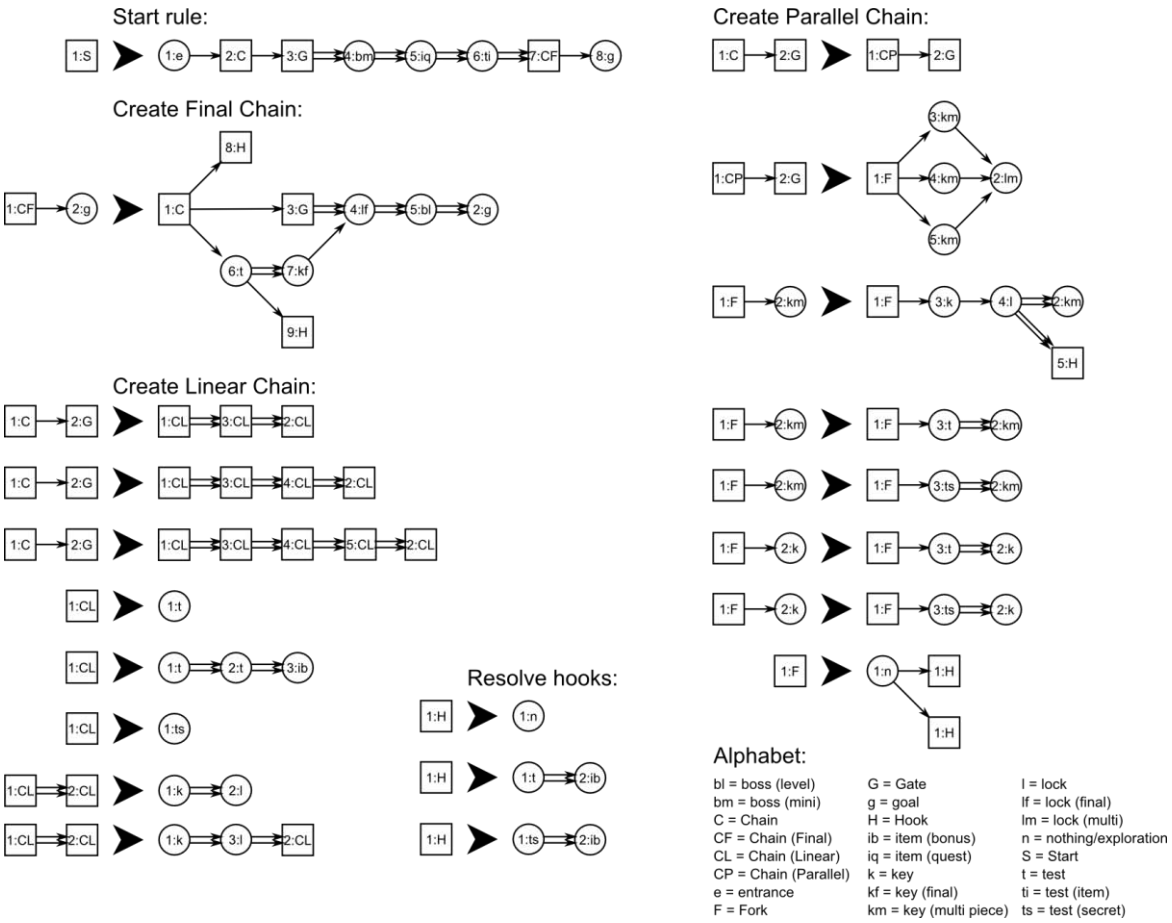

Figure 4. Rules to generate a mission

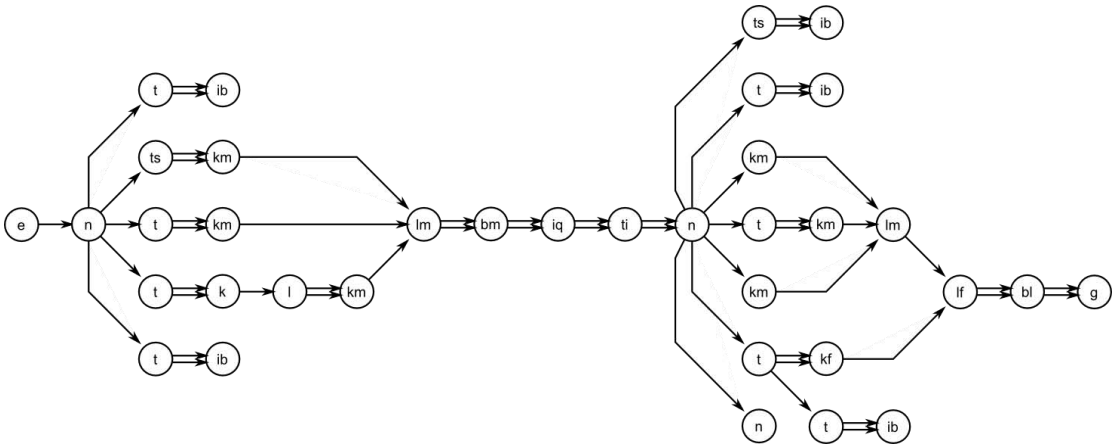

Figure 5. A generated mission (from the rules in figure 4)

Graph grammars are well suited to generate missions as missions are best expressed as nonlinear graphs. It would need an alphabet that consists of different tasks, including challenges and rewards. Figure 4 shows some rules to generate a mission structured similarly as the mission of the forest temple. Figure 5 shows sample output of the graph grammar. Note that this grammar includes two types of edges, represented by single arrows and double arrows; different types of edges is a feature that can be found in other graph grammars. In this case, the double edges indicate a tight coupling between the subordinate node and its super-ordinate: this means that the subordinate must be placed behind the superordinate in the generated space. It is specific to the implementation described in this paper. A normal edge represents a loose coupling and indicates the subordinate can be placed anywhere. This information is very important for the space generation algorithm (see section 6 below).

5. SHAPE GRAMMAR TO GENERATE SPACE

Shape grammars are most useful to generate space. Shape grammars have been around since the early 1970s after they were first described by George Stiny and James Gips [19]. Shape grammars shapes are replaced by new shapes following rewrite rules similar to those of generative grammar and graph grammar. Special markers are used to identify starting points and to help orientate (and sometimes scale) the new shapes.

For example, imagine a shape grammar, which alphabet consists of three symbols: 'a wall', 'open space' and a 'connection' (see figure 6a). In this grammar only the 'connection' is a non-terminal symbol, which has a square marker with a triangle indicating its orientation. The grey marker on the right-hand side of a shape grammar rule as represented here, indicates where the original shape was and what its orientation was. We can design rules that determine that a connection can be replaced by a short piece of corridor, a T-fork or a wall, effectively closing the connection (see figure 6b). It should be apparent that the construction depicted in figure 6c is a possible output of these rules, provided that the start symbol was also a connection, and given that at every iteration a random connection was selected to be replaced.

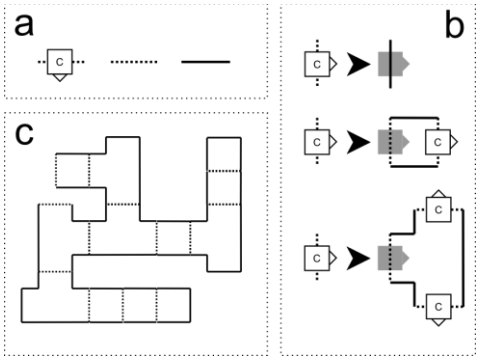

Figure 6. Shape grammar a) alphabet, b) rules and c) output

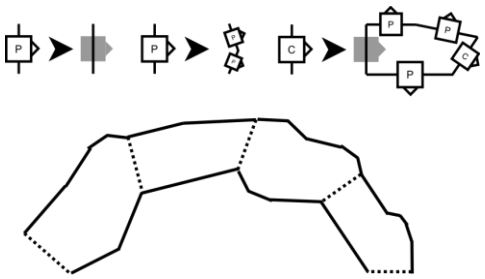

Figure 7. Recursive shape rules and output

Shape grammars, like any generative grammar can include recursion. Recursion is a good way to introduce more variation in the resulting shapes. For example, the rules in figure 7 are recursive and the shapes these rules produces will have a more natural (fractal) feel. In this case the implementation of the grammar should allow the right-hand side to be resized to match the size of the growing shape.

6. GENERATING SPACE FROM MISSION

In order to use a shape grammar to generate a space from a generated mission a few adjustments were made to the shape grammar. The terminal symbols in the mission need to function as building instructions for the shape grammar. To achieve this, each rule in the shape grammar was associated with a terminal symbol form in the mission grammar. The prototype that implements the

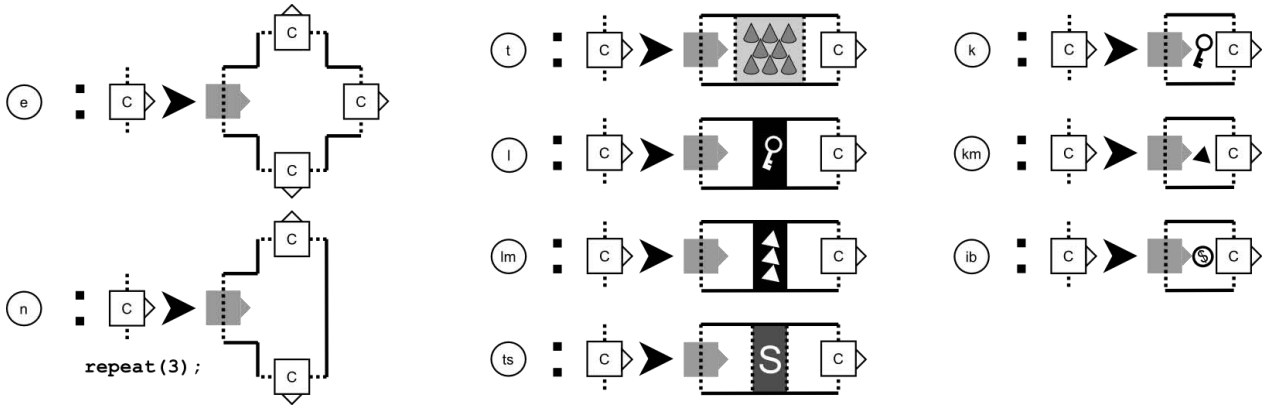

Figure 8. Shape grammar rules to generate missions

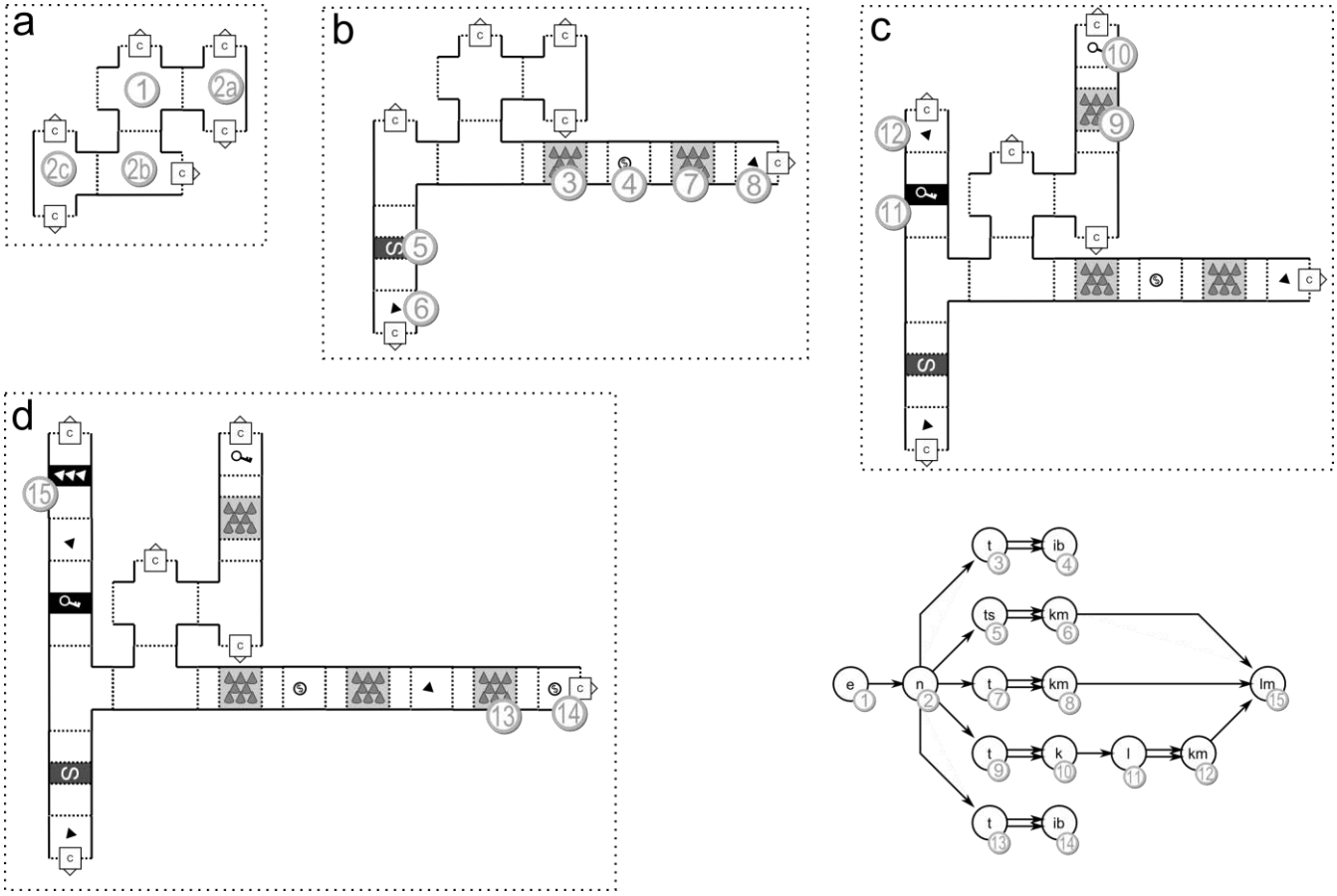

Figure 9. Space generation using the rules from figure 8 and part of the mission from figure 5.

shape grammar first finds the next symbol in the mission, looks for rules that implement that symbol, selects one at random based on their relative weight, then looks for possible locations where the rule could be applied, and finally selects one location randomly based on their relative fitness (one location might be more suitable than another). The algorithm stores a reference to the mission symbol for which each element was generated, allowing the algorithm to implement the tight coupling as dictated by the mission. This prevents the algorithm from placing keys and items at random locations instead of behind tests or locks as specified by the mission. The shape grammar is further extended with some dynamic parameters that influence the rule selection. These parameters are used to create progressive difficulty or to shift between different 'registers'. For example the grammar can increase the chance of selecting rules with more difficult obstacles with every step, and switch from a register that causes it to build many traps to a register that causes it to include many monsters.

In the test application supporting this research, rules can have commands associated with them. These commands are executed either before or after the application of a rule. These commands

facilitate dynamic rule weights and progressive difficulty among other things.

Another addition to the shape grammar was inspired by the automatic creation of road intersection in city modeling L-Systems (see section 3). In order to ensure that the growing space actually reconnects to previously generated parts, a step was added to the algorithm. This step is executed after a rule has been placed in the space, and looks for two connections that are in close proximity and in the correct alignment to be connected, and connect the two spaces. To prevent short circuiting the mission, by accidently connecting the final room to a room near the entrance, all open connections in the generated structure can be closed off after or before the implementation of a particular rule. The commands associated with a rule were used to implement this type of logic.

Once the complete mission is accounted for, the shape grammar reverts to a normal implementation, and will continue to iterate until all non-terminals are replaced with terminal symbols using a set of rules designed to finalize the space (or perhaps to grow some additional branches). Figure 8 lists some rules for a shape grammar constructed in this way. Figure 9 illustrates a few iterations in the construction of a level based on the first part of the mission presented in Figure 5 above.

In theory it should not be very difficult to generate maps that can accommodate multiple missions. Missions could be blended, with the generator alternating between missions when selecting the next task to accommodate on the map. Alternatively, a second mission is used as building instructions after the first mission has been completely accounted for.

7. INVOLVING PLAYER PERFORMANCE

The generation techniques discussed in this paper can also be employed to (partly) generate levels during play, allowing for the opportunity to let the actual performance of the player impact this generation. A good strategy would be to generate a mission before a level starts, ensuring the level will have an interesting overall structure, while the space grows in response to the players movements. As this generation of the game world occurs during play and could involve dynamic weights for the different space rules, this allows for the actual performance of the player to inform the construction of the world. For example, if the player already has encountered and fought many monsters, the rules that would generate more monsters might decrease weight while rules that would generate obstacles of a different type might increase in weight. This would ensure varied gameplay. Or, when the player performance indicates she enjoys these fights (for example because she goes after every monster she can find), we might throw more, and tougher, monsters at her. A feedback loop between the actual performance the player and the generation of the game offers are many opportunities.

A lighter variant of this approach leaves a few non-terminals in the generated space to be replaced during play. Such non-terminals could specify that there is an obstacle or a reward in a particular dungeon room, without specifying what the nature of the obstacle or the reward is until the player triggers the replacement of the non-terminal by entering the room or opening the container. This allows the game to dynamically alter both the challenges and the rewards in reaction to the players performance and status.

Another, more difficult possibility, is to generate the mission on the fly. The best strategy would be to generate a mission that still has some non-terminals in its structure before constructing the space. The replacement of these non-terminals should occur during play, and should be informed by the performance of the player directly or indirectly. The space could either grow in response to the changes in the mission, or already have accommodated all possibilities. This could quite literally lead to an implementation of an interactive structure that Marie-Laure Ryan calls a fractal story where a story keeps offering more and more detail as the player turns her attention to certain parts of the story [20].

8. CONCLUSIONS

The levels of action adventure games are double structures consisting of both a space and a mission. When generating levels for this genre procedurally, it is best to break down the generation process in two steps. Generative graph grammars are suited to generate missions. They are capable of generating non-linear structures which for games of exploration are preferred over linear structures. At the same time they can also capture the larger structures required for a well-formed game experience. Once a mission is generated an extended form of shape grammar can be used to grow a space that can accommodate the generated mission. This requires some modifications to the common implementation of shape grammars. The most important modification is the association of a rule in the shape grammar with a terminal symbol in the grammar used to generate the mission.

Breaking down the process into these two steps allows us to capitalize on the strengths of each type of grammar. With a well-designed set of rules and the clever use of recursion, this method can be employed to generate interesting and varied levels that are fun to explore and offer a complete experience. Furthermore, these techniques can be used to generate levels on the fly, allowing the game to respond to the player performance. This opens up opportunities for gaming and interactive storytelling that hitherto have hardly been examined.

Although the principles behind this strategy for procedural content generation are independent of an implementation for a particular game, the grammars themselves are not. Mission and space grammars must be build with a clear vision of what the final game will be like. Furthermore, the quality of the grammars is going to be a critical factor for the quality of the game, their creation requires involvement of expert game designers or the use of evolutionary algorithms not described here. Nevertheless, using mission and space grammars are an efficient way of generating a high variety of quality levels for action adventure games.

9. ACKNOWLEDGMENTS

I would like to thank Jacob Brunekreef, Stéphane Bura, Ethan Kennerly, Wilko Oskam and Remko Scha, for reading and commenting on earlier drafts of this paper.

10. REFERENCES

[1] Anderson, Mike. Not dated. “Dungeon-Building Algorithm. On RogueBasin”. DOI= http://roguebasin.roguelikedevelopment.org/index.php?title=Dungeon-Building_Algorithm

[2] Author unknow. Not dated. “Grid Based Dungeon Generator. On RogueBasin”. DOI= http://roguebasin.roguelikedevelopment.org/index.php?title=Grid_Based_Dungeon_Generator

[3] Babcock, Jim. Not dated. “Cellular Automata Method for Generating Random Cave-Like Levels”. On RogueBasin. DOI= http://roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels

[4] Adams, Ernest. 2000. “A letter from a dungeon”. Gamasutra. DOI=http://www.gamasutra.com/view/feature/3424/the_designers_notebook_a_letter_.php

[5] DeMaria, Rusel and Johnny L. Wilson. 2004. High Score! The illustrated history of electronic games. McGraw-Hill/Osborne, Emeryville, CA, 240.

[6] Dormans, Joris. 2006. “On the Role of the Die: A brief ludologic study of pen-and-paper roleplaying games and their rules”. Game Studies, vol 6-1, December 2006. DOI=http://gamestudies.org/0601/articles/dormans

[7] Compton, Kate and Mateas Michael. 2006. “Procedural Level Design for Platform Games”. Proceedings of the American Association for Artificial Intelligence Conference, 2006, 109.

[8] Dormans, Joris. “Mission Space: Elemental Morphology for Level Design”. Submitted to the IEEE-CIG-10 conference, Copenhagen.

[9] Kohler, Chris. 2005. Power Up, How Japanese Video Games Gave the World an Extra Life. Brady Games, Indianapolis, IN.

[10] Campbell, Joseph. 1949. The Hero With A Thousand Faces. Princeton University Press, Princeton, NJ.

[11] Vogler, Christopher. 2007. The Writer’s Journey: Mythic Structure for Writers, Third Edition. Michael Weis Productions, Studio City, CA.

[12] Ashmore, Calvin & Nitsche, Michael. 2007. “The Quest in a Generated World”. Proceedings of the DiGRA 2007 Conference, 506.

[13] Chomsky, Noam. 1972. Language And Mind, Enlarged Edition. Harcourt Brace Jovanovich Inc, New York, NY.

[14] Golding, James. 2010. “Building Blocks: Artist Driven Procedural Buildings”. Presentation at GDC 10, San Francisco, CA.

[15] Mozgovoy, Maxim. 2010. Algorithms, Languages, Automata and Compilers: A Practical Approach. Jones and Barlett Publishers, LCC. Sudbury, MA.

[16] Parish, Yoav & Müller, Pascal. 2001. “Procedural Modeling of Cities”. Procedeedings of th ACM SIGGRAPH 2001 Conference.

[17] Adams, David. 2002. Automatic Generation of Dungeons for Computer Games. Bachelor thesis, University of Sheffield, UK. DOI= http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2002/pdf/u9da.pdf

[18] Rekers, J. 1995. “A Graph Grammar Approach to Graphical Parsing”. Proceedings of the 11th International IEEE Symposium on Visual Languages.

[19] Stiny, George & Gips, James. 1972. “Shape Grammars and the Generative Specification of Painting and Sculpture”. Proceedings of Information Processing 71.

[20] Ryan, Marie-Laure. 2001. Narrative as Virtual Reality, Immersion and Interactivity in Literature and Electronic Media. The Hohns Hopkins University Press, Baltimore, MD, 337