Generating Missions and Spaces for Adaptable Play Experiences

Joris Dormans and Sander Bakkes

***Abstract*—This paper investigates strategies to generate levels for action-adventure games. For this genre, level design is more critical than for rule-driven genres such as simulation or rogue-like role-playing games, for which procedural level generation has been successful in the past. The approach outlined by this article 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. Notably, the approach acknowledges that the online generation of levels needs to be tailored strictly to the actual experience of a player. Therefore, the approach incorporates techniques to establish and exploit player models in actual play.**

***Index Terms*—Game AI, game design, generative grammars, real-time generated game environments.**

I. Introduction

In THE domain of video games, procedurally generated content is considered to be of increasing importance to the computer–game development in the present and in the future; both offline, for making the game development process more efficient (design of content such as environments and animations now consume a major part of the development budget for most commercial games), and online, for enabling new types of games based on player-adapted content [1]. In fact, games with procedurally generated content have been around for some time.

The classic example of this type of game is *Rogue* [2], an old *Dungeons & Dragons* [3] style ASCII dungeon-crawling game which levels are generated every time the player starts a new game. Newer games that use procedural techniques include *Diablo* [4], *Torchlight* [5], *Spore* [6], and *MineCraft* [7]. 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. Often these algorithms generate a large sample and rely on evaluation functions to select the level that is the most fit [8]. Others evaluate the level in order to remove areas that turn out to be unreachable [9]. One strategy, for example, 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 [10]. Another strategy involves zoning the dungeon into large tiles, generating dungeon rooms in some of these zones in the next step, and finally connecting the rooms with a network of corridors [11]. To create game space to represent wilderness areas cellular automata are used to generate more organic structures [12].

Although these algorithms have a proven track-record for the creation of rogue-like games, the gameplay their output supports does not necessarily translate to the generation of action-adventure games. Action-adventure games are story-driven games where exploration, puzzle-solving, conceptual and physical challenges make up the majority of the gameplay [13]. Compared to simulation games and role-playing games, action–adventures typically have a relatively simple set of simulation rules and only a few available power-ups. These games usually do not have an elaborate leveling system where character development, expressed in terms of skills and attributes, is an essential part of the gameplay. Lacking these, action–adventure games must rely more on level design as its prime source of gameplay. As a result, a structured learning curve, clever pacing of action, challenges, and puzzles play a more prominent role for the levels in an action–adventure game. A procedure to generate levels for this genre must include a way to incorporate these elements. It is in a similar light that Smith *et al.* [14] point out that generating levels for an action-platform game is more difficult, as level design is also a far more critical aspect of that type of game.

As it turns out, level-design principles, like flow, pacing and structured learning-curves, are difficult to implement with the algorithms commonly encountered in rogue-like games. These algorithms generally cannot express these principles as these principles mostly operate on larger structures than the individual dungeon rooms and corridors the algorithms work with. The level-design principles that we have in mind are principles such as disguised locks and keys that are typically found in action-adventure games [15]. But also principles akin to those described by the level design patterns discussed by Hullett and Whitehead for first-person shooter games [16] which can span entire sections or levels.

In order to generate game levels informed by such principles we need to turn to a method that does operate on the same type of structures these principles operate on. This method is the use of generative grammars, which are very good at operating on

Manuscript received November 01, 2010; revised March 01, 2011, April 15, 2011; accepted April 21, 2011. Date of publication May 05, 2011; date of current version September 14, 2011. This work was partially funded by the SIA project “Smart Systems for Smart Services.”

The authors are with Section Game Development/CREATE-IT Applied Research, Amsterdam University of Applied Sciences (HvA), Amsterdam, The Netherlands (e-mail: j.dormans@hva.nl; s.c.j.bakkes@hva.nl).

Color versions of one or more of the figures in this paper are available online at http://ieeeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2011.2149523

large abstract structures, and fine detail at the same time. Generative grammars also have the advantage of generating levels, according to the grammar, and must be syntactically correct. This means that with the right implementation of the grammar, there is no need to verify its correctness or to select correct levels from a large generated sample. However, even with the use of generative grammars, generating good levels is still very hard. Levels often have a random feel and tend to lack overall structure. To search simply for a single generative grammar to tackle all these problems is not sufficient. As we will argue below, 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 subsequently generate spaces to accommodate these missions. A similar approach was outlined in a previous paper [17]. However, this paper presents the next iteration of the same research. We acknowledge that the online generation of levels needs to be tailored strictly to the actual experience of a player. Therefore, the approach incorporates techniques to establish and exploit player models in actual play.

II. Background: Hierarchy of Challenges

It is common to frame level generation as a spatial problem. However, a game level is not just a map. Levels offer a structured experience by setting up a number of tasks for the player to perform. The way these tasks are structured constitutes an alternative perspective on how a level functions, a perspective that can be leveraged for generation of game levels.

There are a few ways to organise these tasks. Cousins [18] introduces the idea of a hierarchy to order a gameplay experience. In a detailed analysis of the game *Super Mario Sunshine* [19] he divides the experience in five layers, where the top layer constitutes the whole game. The subsequent layers describe the individual missions, mission elements, input elements, and primary elements. The middle layers correspond to player's plans and intentions. They correspond with what cognitive science would call basic level categories for the games actions; these actions are most accessible to players and would typically be the same action players would use when describing the gameplay. The lowest layer represents the actions made possible by the game mechanics and interface; these correspond with the buttons pressed by a player and the resulting actions of the avatar, such as jumps or simple moves. Cousins stresses the importance of the quality of these low level actions as the player spends a lot of time performing them. Using simple metrics Cousins shows that 55% of primary elements in the four minute session of *Super Mario Sunshine* [19] consisted of running forward and changing direction.

The hierarchy of challenges described by Adams and Rollings [13] was directly inspired by Cousins' analysis. However, where Cousins focuses on individual game sessions, with the hierarchy of challenges Adams and Rollings try to capture all possible sessions into one single representation. In this hierarchy all the game's challenges are ordered into a layered structure representing what a player needs to accomplish to complete the game. To build a hierarchy of challenges requires inside knowledge of the level's design. Some challenges can be performed in different order or even simultaneously. At the lowest level in the hierarchy are the atomic challenges; the micro actions the player needs to perform to get ahead. At higher levels in the hierarchy there are goals of individual sections and missions. At the highest level the game's ultimate goal resides. Adams and Rollings discuss the different needs of visibility among the levels: games need to be very clear in their high level and atomic level challenges while they can be less clear for intermediate challenges. They also stress that presenting a player with simultaneous atomic challenges considerably increases the game's difficulty [13].

Creating simultaneous or alternative options in the hierarchy of challenges leads to a potentially more varied gameplay. However, this is more difficult in the higher levels of the hierarchy than it is in the lower levels. The game *Deus Ex* [20] is a good illustration of this. In this game the player has to get through a series of connected missions. Most low level challenges present the player with options: to deal with a guard the player might use stealth or violence. As the player progresses she gets to choose how the game's protagonist develops his skills; she can choose to specialize in different strategies to overcome common challenges. Yet, at the high end of the challenge structure there are only few options and branches. It is only at the very end of the game that the player gets to choose between one of three alternative endings.

III. Missions and Spaces

From a detailed study of the level design of the Forest Temple level of *The Legend of Zelda: The Twilight Princess* [21] two different structures emerge that both describe the level (see Fig. 1). First, there is the 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 the 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 Fig. 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 nonlinear spatial configuration. Likewise, a nonlinear mission featuring many parallel challenges and alternative options, can be mapped

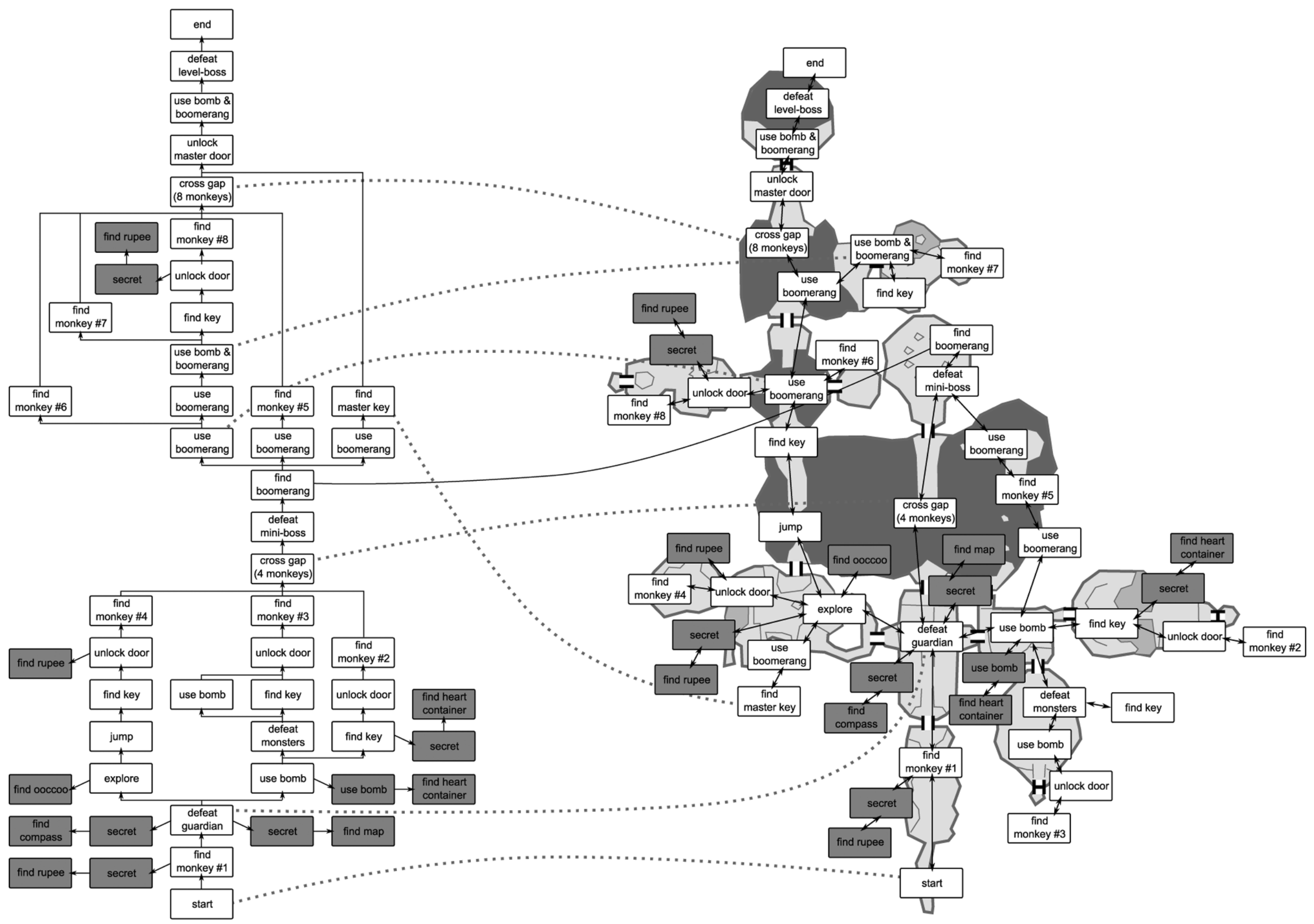

Fig. 1. Mission and space in the Forest Temple Level of *The Legend of Zelda: The Twilight Princess* [21].

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 [22], 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 [23]. Following this structure a player first learns a simple technique in isolation (the *kihon* stage), then he 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 on, wax off"). Next, the player learns how different techniques can be combined (the *kata* stage) before his 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 a 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*).

The way these individual moves are learned and are combined into more advanced moves also recalls Cook's work on "skill atoms" [24]. He frames a player's journey through a game as a chain of learning new skills the player needs to negotiate the game environment. On the atomic level there is a cycle in which the player performs an action, such as pressing a button on the controller, which has an effect simulated in the game. The feedback of that effect hopefully will lead to an update to the player's mental model of the game. Skills need to be practiced in order to be mastered and are combined into more advanced skills requiring the mastery of a few prerequisite skills [24].

This structure, that follows a well-defined recipe, and which is repeated in many dungeons in the same game, and same series can be summed up as follows: The player enters a dungeon and works his way through a short series of challenges ending with an enemy that guards the entrance to the main dungeon. This series usually sets the tone for the rest of the dungeon. After the confrontation with the guardian, the player can explore the dungeon in multiple directions. Around halfway or two-thirds into the level the player defeats a midlevel boss and obtains and item that he will need in order to progress beyond certain obstacles encountered during the first half. Finally, the player confronts and defeats the level boss with the aid of the acquired item. It is this types of structures that constitute design principles that procedural level generation must incorporate in order to successfully generate levels suitable for action–adventure games.

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 [15].

IV. Generative Grammars

Before detailing our approach to dynamically generated missions and game spaces, we introduce the concept of generative grammars, and provide a discussion on the advantages and applications of generative grammars.

A. Concept

Generative grammars originate in linguistics where they are used as a model to describe sets of linguistic phrases [25]. 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" and "b" in the last example are terminals. Nonterminals 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 nonterminal 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 [26]; they are designed to analyze and classify language. Moreover, grammars are suited for predicting [27], and automatically generating language phrases [28]. We utilize grammars for this latter purpose. It is easy to see that simple rules can produce quite interesting results especially when the rules allow for recursion: when the rules produce nonterminal 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 the generation of stories [29], 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," and "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 nonterminal 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. For instance, string 3 is too short (and uninteresting) even in the limited example above. The problem stems not from generative grammars as such, but from quality of the rules that are 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* [21] above.

B. Advantages, Disadvantages, and Applications

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 procedural 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 highly versatile as it allowed designers to rapidly regenerate previous sections with the same constraints but with new rule sets without having to redo a whole section by hand [30]. 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.

A disadvantage of generative grammars, though inherently controllable, is that the ultimate result of the generation process may initially be difficult to foresee for game developers. Also, the construction of generative grammars may be relatively hard and time intensive. Knowledge of generative grammars is not common among game designers, yet it is required to make the most of this approach. Additional playtesting may be required in early stages of the design process to ensure an adequate grammar is adopted. Using generative grammars requires a considerable investment initially. However, once this investment is made, the expected return is high.

Relevant applications of generative grammars can also be found in with Lindenmayer Systems (L-Systems) [31]. 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 [32]. Within the search-based PCG framework, L-systems form a well-functioning way of adapting content that is based on grammar rewriting (cf., e.g., [33] and [34]). 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 [35]. 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.

C. Graph Grammars

Graph grammars are discussed in relation with level generation by David Adams in his 2002 thesis *Automatic Generation of Dungeons for Computer Games* [36]. Graph grammars are a specialized form of generative grammars that produce graphs consisting of edges and nodes, instead of producing strings. In a graph grammar one or several nodes and interconnecting edges can be replaced by a new structure of nodes and edges (see Figs. 2 and 3; [37]). 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 Fig. 3). Next, all edges between the selected nodes are removed (step 3). The numbered nodes are then replaced by their equivalents (i.e., 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). We note that graph grammars can have operations that

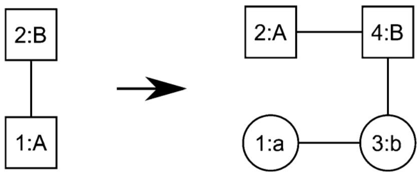

Fig. 2. Example of a graph grammar rule. The numbers are used to identify corresponding nodes on the left and right hand side. The squares indicate nonterminal nodes, whereas the circles indicated terminal nodes; [37].

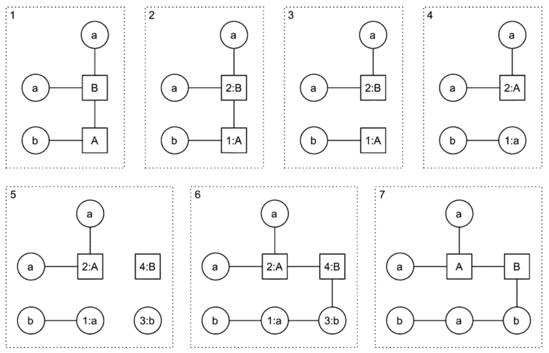

Fig. 3. Replacement operations according the rules from Fig. 2; [37].

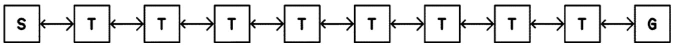

Fig. 4. Sample linear mission with a start, goal and eight tasks to be completed.

allow existing nodes to be removed. However, for the application of graph grammars in level generation this may not be a desirable functionality.

For our purpose, graph grammars are well suited to generate missions as missions are best expressed as nonlinear graphs.

V. Generating Missions

In its most simple form a mission consists of start, a goal, and a number of tasks in between. This structure can be easily represented by a directed graph containing a single line or a simple string (see Fig. 4). Obviously, such a mission is very easy to generate but it would not be very interesting. We could vary the tasks, some tasks might involve fighting enemies, while other would involve solving puzzles. With good pacing, this leads to interesting games with procedural levels, such as in the currently popular running games (e.g., *Canabalt* [38]). Indeed, linear missions are not a feature of action adventure games. Therefore, for a game where exploration is important, we need to reorder these task in a nonlinear fashion.

Graph grammar rules can be used to transform our linear mission into a different structure. Consider for example the transformative rule represented in Fig. 5. Simply applying this rule a number of times could transform the linear mission in Fig. 4 into the mission of Fig. 6. Although this structure still will not make a very good level, an interesting property of applying this rule is illustrated: the number of tasks is not affected, the rule only reorganizes their connections. This illustrates that, when the rules are drafted right, these transformations are quite controllable; no matter how often a rule is applied the number of

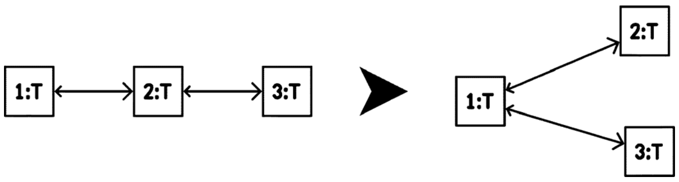

Fig. 5. Reorganize tasks rule. As in Fig. 2, the numbers are used to identify corresponding nodes on the left and right hand side.

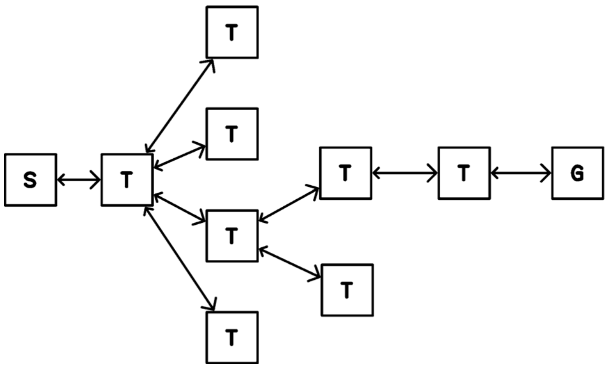

Fig. 6. Sample mission reorganized.

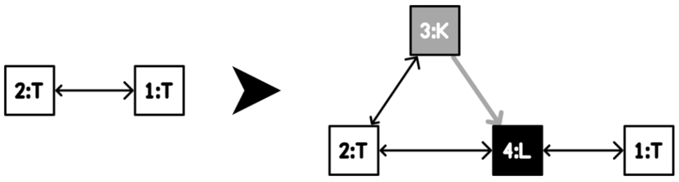

Fig. 7. Rule to add a key and lock. The colors of the nodes are only used to help identify the node types, the gray lines indicate which key opens what lock.

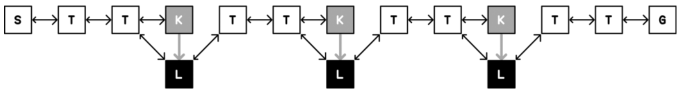

Fig. 8. Mission with keys and locks.

tasks will stay the same. However, the rule also leads to a structure where all tasks no longer need to be completed. There is a short path leading directly to the goal, if the player takes this direct route, only four tasks will be encountered.

A better way to introduce multiple branches through the mission is to introduce locks and keys. Locks and keys, in many different guises are very common in adventure games. Zelda games are filled with lock and key problems, however not all locks and keys are immediately recognizable as such. As mentioned above, the boomerang found in the Forest Temple level acts as a key to multiple locks the player needs to unlock in order to proceed. To add locks and keys we can simply introduce the rule depicted in Fig. 7. which could transform our simple mission into a construction as found in Fig. 8. The gray connections in Figs. 7 and 8 indicate that the keys are required to unlock the doors. The black connections indicated how a player might move through the mission.

Obviously, locks and keys are best not found at the same time. A few simple rules that allow us to move locks and keys throughout the mission solves this problem (see Figs. 9, 10,

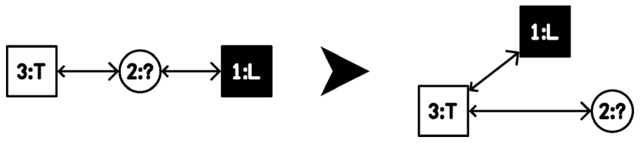

Fig. 9. Rule to move a lock forward. The circular node marked with a question mark indicates any node. Applying this rule will not change the type of this node, only its location and connections in the graph.

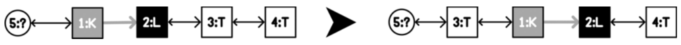

Fig. 10. Rule to move a key backwards by relocating a task from behind the door to the position right in front of the key.

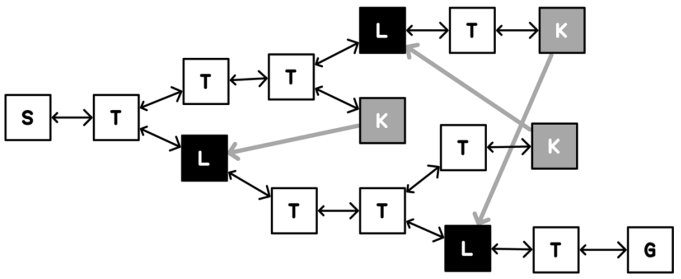

Fig. 11. Reorganized mission with keys and locks.

and 11). In fact, by making sure that locks tend to move forward and keys tend to move backwards, the criteria that players must encounter the door before they encounter the key is codified in these rules (recall that the criteria reflects conventional level-design wisdom). The question mark identifying the node with number 2 in Fig. 9 indicates that this can be any node. Note that the rule in Fig. 10 can never relocate the last task behind a door, this way the rule cannot create doors that would lead to nowhere.

By adding rules that allow doors to move forward past other doors, duplicating keys, or create parallel tasks many different missions can be generated. If the transformation rules are drafted correctly, these will always generate missions that can be solved. By adding extra rules to include quest items that function as keys to new locks and tasks which, when completed, prepare the player for tasks that follow, the level design principles that drove the design of the Forest Temple level can be easily codified. The number of missions that can be generated with these rules are infinite. Also, we note that missions can be extended at run-time, while the player is playing the game. This is implemented by maintaining unused nonterminals in the mission structure, that may be filled in at a later time, alongside the concerning space generation.

The greatest advantage of the proposed technique is that it is highly controllable. The number of tasks, and the number of locks and keys generated is an immediate result of number of times certain rules are applied. Depending on how the rules are constructed, a designer might specify he wants to generate a dungeon that includes three locks, and duplicate one key. What is more, it is fairly easy to give human designers direct control over the selection of rules, so that the construction of the level might not only be fully automated, it might also be collaboration between the computer and a human designer (see [39] and [40]

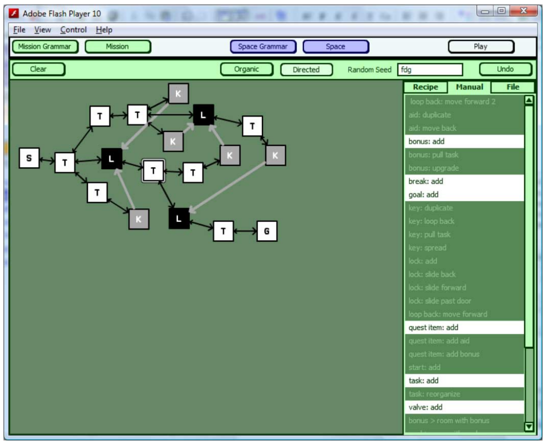

Fig. 12. Level generation prototype developed for this research. The designer selected a task in the middle of the mission structure. The applicable rules are displayed on the right.

for such a "mixed-initiative" approach). In the prototype we developed for this research such collaboration is already possible: a designer is able to select nodes and manually choose to apply rules that are applicable for that node (see Fig. 12).

VI. Generating Spaces

Once a mission structure is generated there are several strategies to build spaces to accommodate the mission. In a previous paper, we described a method that uses shape grammars to define spatial parts which are used to build up the space not unlike a jig-saw puzzle [17]. Although this approach works, it has difficulty generating spaces for missions which allow multiple paths to converge at the same target. To deal with this problem we take advantage of the spatial nature of a 2-D representation of a graph. Our prototype can generate an organic layout for the mission graph by simulating all nodes in the network as nodes with connections functioning as springs with some basic algorithms to reduce the number of overlapping connections (see Fig. 13). As it stands there is a lot of opportunity to improve this algorithm, but for now this has little priority. Within the context of action adventure games there are many solutions to deal with crossing connections should the algorithm not be able to solve them all. Using teleporters is one such solution that might be applicable in certain game contexts, stacking multiple 2-D levels and creating portals or stairs between them might constitute another solution.

Subsequently, simple replacement rules allow us to replace tasks with rooms of various sizes, place keys inside them and have locked doors connect them (see Fig. 14). From this a spatial structure can be generated that follows the same outline and consists of planes and edges (see Fig. 15).

The next step involves using shape grammar to flesh out this basic shape and generate more detail. Shape grammars have been around since the early 1970s after they were first described by Stiny and Gips [41]. 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.

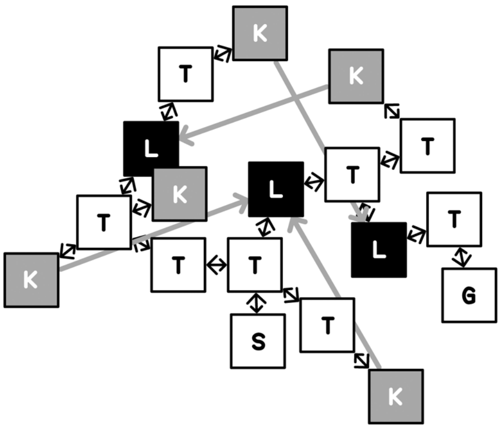

Fig. 13. Mission in an organic layout.

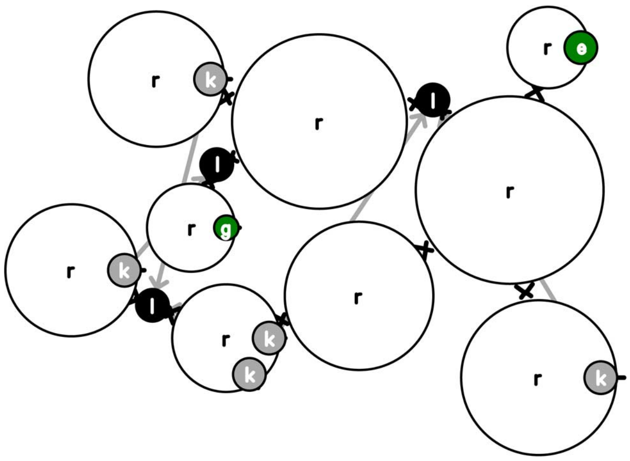

Fig. 14. Tasks replaced with rooms of various sizes.

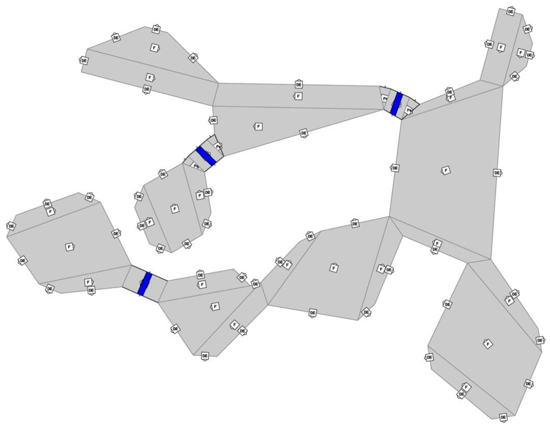

Fig. 15. Mission structure from Fig. 14 translated to a spatial construction.

For example, imagine a shape grammar, whose alphabet consists of three symbols: "a wall," "open space," and a "connection" [see Fig. 16(a)]. In this grammar only the "connection" is a nonterminal symbol, which has a square marker with a triangle indicating its orientation. The gray 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 wall (effectively closing the connection), a short piece of corridor, or

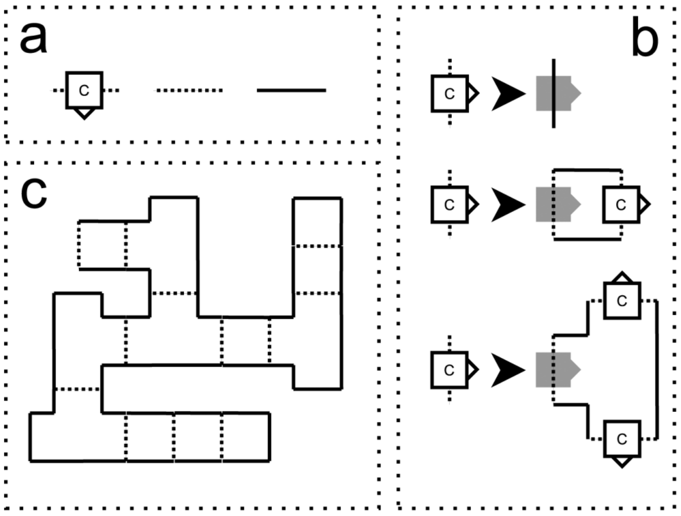

Fig. 16. Shape grammar (a) alphabet, (b) rules, and (c) output.

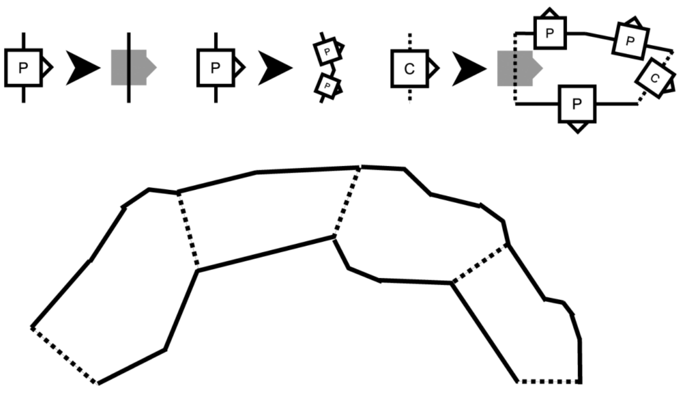

Fig. 17. Recursive shape rules and output.

a T-fork [see Fig. 16(b)]. The construction depicted in Fig. 16(c) 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.

Shape grammars, like any generative grammar can include recursion. Recursion is a good way to introduce more variation in the resulting shapes. An interesting possibility is that hereby it provides shapes to grow in a certain direction. 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. For instance, the rules in Fig. 17 are recursive and the shapes these rules produces will have a more natural (fractal) feel.

Rules like those in Fig. 17 are used to add detail to the spatial construction from Fig. 15, after which the generated space is analyzed to identify natural paths and possible locations to place barriers, etc. (see Fig. 18). The shape grammar rules used to generate the space in Fig. 18 result in a natural-looking cave. This need not be the case. With slightly different rules the procedure can generate rooms that look much more artificial (see Fig. 19). In this case the effect would have benefited from aligning the mission structure to a grid before translating it into a spatial construction.

The particular implementation of using generative grammars to outline and generate game levels outlined here generates mis-

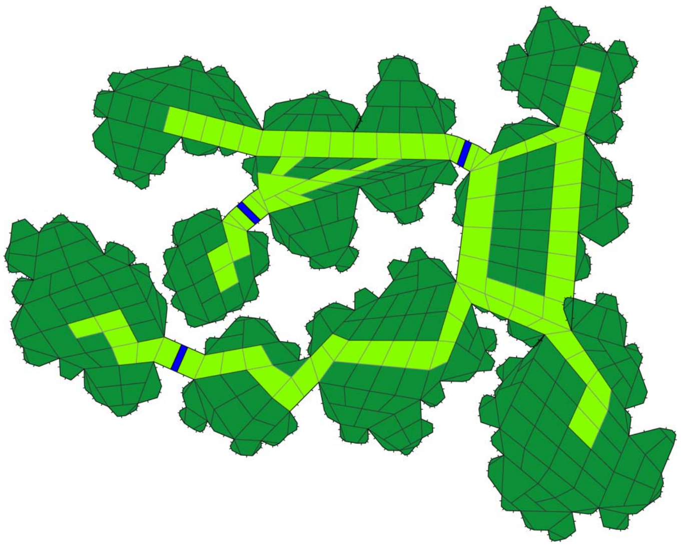

Fig. 18. Space grown with shape grammar and analyzed for natural paths connecting the rooms (natural appearance).

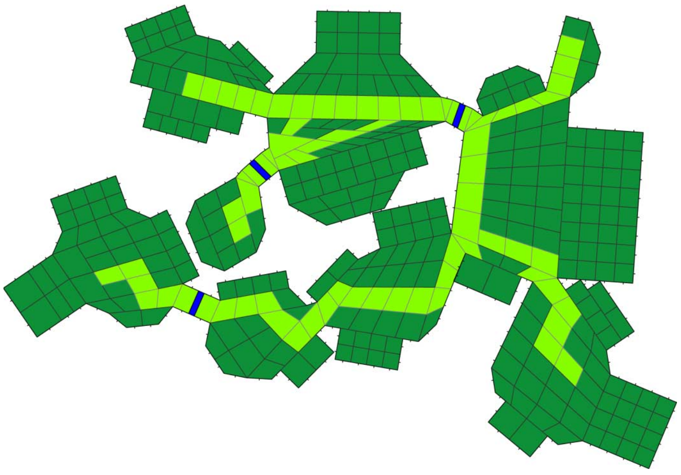

Fig. 19. Space grown with shape grammar and analyzed for natural paths connecting the rooms (artificial appearance).

sions first and spaces afterwards, but this is not the only possible implementation. It is conceivable to set up a procedure that generates spaces first (probably best represented by a graph at this stage), and then generate a mission to match that space. This would probably lead to a process that involves three steps: generating a functional description of a space, generating a mission to match that space, and finally generating a detailed map that conforms to both the functional description and the mission.

As already pointed out the presented techniques are highly controllable. The different application of grammars which results in difference types of dungeons (Figs. 18 and 19) illustrate this again. In addition, the process outlined up until now can be executed very fast, and the results invariably lead to fairly interesting levels. There is no need to generate multiple levels and select the most suitable level in order to get results of reasonable quality. This high speed and high controllability makes the technique particularly suited for level generation and adaptation during play based on player profiles; it can be used to generate and morph levels based on the player's preferences as expressed by his behavior during play. To this end, techniques for player modeling need to be incorporated.

VII. Establishing Player Models

Player modeling is an important research area in game playing. It concerns establishing models of the player (discussed in this section), and exploiting the models in actual play (discussed in the section that follows). In general, a player model is an abstracted description of a player or of a player's behavior in a game. In the case the models concern not the human player, but instead an opponent player, we speak of "opponent modeling." The goal of opponent modeling is to raise the playing strength of the (computer-controlled) player by allowing it to adapt to its opponent and exploit his weaknesses [42]–[45]. On the other hand, the goal of player modeling generally is to steer the game towards a predictably high player satisfaction [46], on the basis of modeled behaviour of the human player.

Player modeling is of increasing importance in modern video games [47]. Houlette [48] discussed the challenges of player modeling in video-game environments, and suggested some possible implementations of player modeling. A challenge for player modeling in video games is that models of the player have to be established: 1) in game environments that generally are relatively realistic and relatively complex; 2) with typically little time for observation; and 3) often with only partial observability of the environment. Once player models are established, classification of the player has to be performed in real time. Other computations, such as rendering the game graphics, have to be performed simultaneously. Researchers estimate that generally only 20% of all computing resources are available to the game AI [49]. Of these 20%, a large portion will be spent on rudimentary AI behavior, such as manoeuvring game characters within the game environment. This implies that only computationally inexpensive approaches to player modeling are suitable for incorporation in the game AI.

For the domain of modern video games, we deem three approaches applicable to player modeling, namely: 1) action modeling; 2) preference modeling; and 3) player profiling. In the later described proposal for exploiting the generated player models, we utilize the preference modeling approach.

A. Action Modeling

In video games, a common approach to establishing models of the player is by modeling the *actions* of the player [50], for instance, by using n-grams [51]. It has been shown that it is possible to create a model of the actions that players tend to take in particular game situations [48]. Although such models are generally quite useful, the actual action that a player takes in a particular game situation will depend not only on the situation but also on the player's overall game preferences (e.g., adopting a particular strategy). For instance, in one play of the game a player may want to win by purely military means whereas in another they may decide to aim for cultural dominance. In general, a player's preference is not easily determined on the basis of only the current game situation, and hence purely action-based models tend to be of limited applicability [50]. What is more, the interest of game developers generally goes out to determining a player's actual game experience, which, by implication, is even less easily determined solely on the basis of action-based models.

B. Preference Modeling

An alternative, more advanced approach is to model the *preferences* of players, instead of the actions resulting from those preferences. This preference-based approach is viable [50], [52] and identifies the model of a player by analyzing the player's choices in predefined game situation. In the preference-based approach, player modeling can be seen as a classification problem, where a player is classified as one of a number of available models based on data that is collected during the game. Behavior of related game AI is established based on the classification of the player. Modeling of preferences may be viewed as similar to approaches that regard known player models: 1) as stereotypes; and 2) as an abstraction of observations [53]. As a means to generalize over observed game actions, a preference-based approach may be of interest to game developers.

C. Player Profiling

A recent development with regard to player modeling, is to establish automatically psychologically verified *player profiles*. Player profiling has gathered substantial research interest [1], [54], [55]. Van Lankveld [54] states that the major differences between player modeling (by means of action modeling, or preference modeling) and player profiling, lie in the features that are modeled. That is, player modeling generally attempts to model the player's playing style (e.g., playing defensively), while player profiling attempts to model traits of the player's personality (e.g., extraversion). The models produced by player profiling are readily applicable in any situation where conventional personality models can be used. In addition, player profiling is supported by a large body of psychological knowledge.

VIII. Exploiting Player Models

In the context of generative grammers, we distinguish three approaches for exploiting player models, namely for: 1) space adaptation; 2) mission adaptation; and 3) difficulty scaling.

Our ongoing work concerns the implementation of these approaches in the generative framework. How we propose to implement the approaches is discussed in Section IX. The adaptation process is tied in with the generative grammars in order to transform the gameplay in a controlled manner.

A. Space Adaptation

A natural starting point for exploiting player models, is to allow the space in which the game is played to grow in response to the actual behavior of the player. First, after observing the player for a select period of time, features within the established player model may indicate that it is recommendable to transform (gradually) the game surroundings. For instance, transform from open to confined spaces, from linear to more organic environments, and from easily maneuverable corridors to intricate mazes. Secondly, variations in gameplay may be provided by, in addition, allowing events that take place within certain game spaces (e.g., particular rooms) to respond to the player's previous behavior. For instance, if the player already has encountered and fought numerous opponent characters, the rules that generate more opponents decrease in weight while rules that generate obstacles of a different type increase in weight. Inversely, when the player model indicates that the player has a preference for combating opponents (e.g., because he chases every opponent he encounters), the adaptive game may confront him with more and tougher opponents.

B. *Mission Adaptation*

A promising alternative to space adaptation, is to allow the game's mission to grow in response to observed behavior of the player. A strategy in this regard, is to generate a mission that still has some nonterminals in its structure before constructing the space. The subsequent replacement of these nonterminals occurs during play, and is directly or indirectly informed by the performance of the player. For instance, obtaining a certain in-game achievement by the player may trigger a dynamically generated parallel mission to be inserted at a nonterminal node.

In turn, the space in which the mission takes place may grow in response to the changes in the mission, or may already have accommodated all resulting 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 his attention to certain parts of the story [56].

C. *Difficulty Scaling*

Player models, aside from forming an input for space and mission adaptation, can in addition be applied for adapting automatically the challenge that a game poses to the skills of a human player. This is called difficulty scaling [57], or alternatively, challenge balancing [58]. When applied to game dynamics, difficulty scaling aims usually at achieving a "balanced game," i.e., a game wherein the human player is neither challenged too little, nor challenged too much.

In most games, the only implemented means of difficulty scaling is typically provided by a *difficulty setting*, i.e., a discrete parameter that determines how difficult the game will be. The purpose of a difficulty setting is to allow both novice and experienced players to enjoy the appropriate challenge that the game offers. Usually the parameter affects plain in-game properties of the game opponents, such as their physical strength. Only in exceptional cases the parameter influences the strategy of the opponents. Consequently, even on a "hard" difficulty setting, opponents may exhibit inferior behavior, despite their, for instance, high physical strength. Because the challenge provided by a game is typically multifaceted, it is tough for the player to estimate reliably the particular difficulty level that is appropriate for himself. Furthermore, generally only a limited set of discrete difficulty settings is available (e.g., easy, normal, and hard). This entails that the available difficulty settings are not fine-tuned to be appropriate for each player.

In recent years, researchers have developed advanced techniques for difficulty scaling of games. Demasi and Cruz [59] used coevolutionary algorithms to train game characters that best fit the challenge level of a human player. Hunicke and Chapman [60] explored difficulty scaling by controlling the game environment (i.e., controlling the number of weapons and power-ups available to a player). Spronck *et al.* [57] investigated three methods to adapt the difficulty of a game by adjusting automatically weights assigned to possible game strategies. In related work, Yannakakis and Hallam [61] provided a qualitative and quantitative method for measuring player entertainment in real time.

The proposed mission and space grammars combined with player models offer a straightforward means to difficulty scaling. That is, generally game developers may prefer to use the grammars for the purpose of generating the most challenging game environment possible. Analogously, the grammars can be applied for the purpose of obtaining (and maintaining) a predefined target in the provided challenge, via outclassing knowledge on the effect of executing grammar rules on the level's difficulty (cf. [62] and [63]). Finally, one may also scale difficulty spatially: e.g., providing a shorter, more difficult path, and longer, easier paths. This can allow e.g., speed running, risk versus reward tradeoffs in speed versus health etc. Grammars can exploit information of the established player models, to generate dynamically an easily maneuverable environment with appropriately few powerful opponents, instead of a complex environment with overwhelmingly many powerful opponents. The proposed implementation of the previously described is discussed next.

IX. Adaptable Play Experiences With Grammars

Here we discuss the proposed grammar-based implementation of adaptable play experiences. We cover the topics of: 1) grammars for space adaptation; 2) grammars for mission adaptation; and 3) grammars for difficulty scaling. We note that the adaptation process is tied in with the generative grammars in order to transform the gameplay in a *controlled* manner.

A. *Grammars for Space Adaptation*

Adapting the game's space in response to behavior of the player is a relatively straightforward procedure. It consists of two steps, namely: 1) classifying the player behavior (i.e., determining which player model best explains the observed behavior); and 2) dependent on the classification, applying an adaptation grammar to gradually transpose the rules of the shape grammar into distinct, other rules. In this subsection we focus on the second step, considering the paper's focus on generative grammars, and refer the reader to Section VII for literature on the topic of modeling the player behavior.

Concretely, our adaptation grammars define how a certain rule given in the space grammar should be transposed into another rule. For example, for the rules illustrated in Fig. 16(b), the adaptation grammar will define how the rule to generate rectangular shaped rooms should gradually transform into a rule to generate organically shaped rooms. This process is illustrated in Fig. 20. Indeed, a rule is gradually (instead of instantaneously) transformed with the intention to maintain player immersion, which may predictably be lost in case of abrupt changes in the generation process.

An extension left for future work, is transposing already generated spaces dynamically into spaces of different appearance

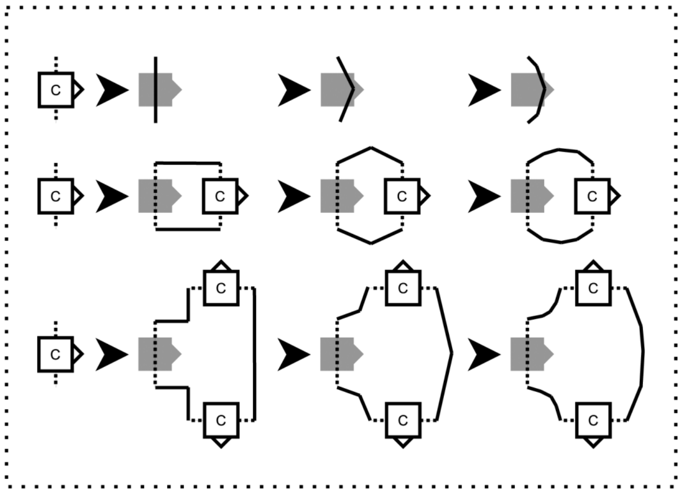

Fig. 20. Set of shape grammar rules used in the space adaptation process. The adaptation rules result in a gradual transformation of the rules to generate spaces. Execution of a set of adaptation rules depends strictly on the classification of the player's behavior.

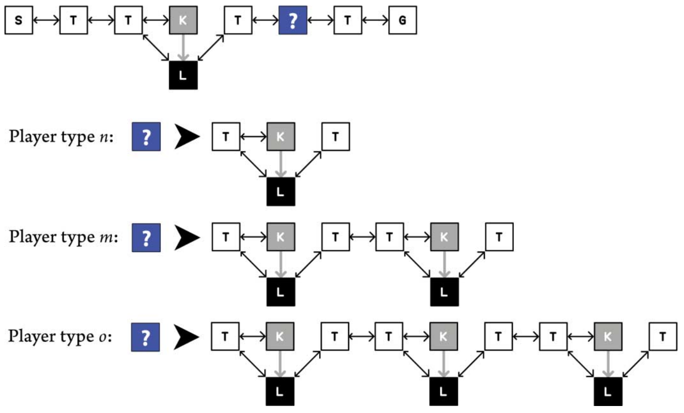

Fig. 21. Set of mission grammar rules used in the mission adaptation process. In the figure, the adaptation rule for either "player type *n*," "*m*," or "*o*" is applied to the adaptive nonterminal node "*?*," resulting in a player-dependent mission structure.

and property (cf. the video game *Ōkami* [64].) For instance, an already generated space of abstract, rectangular appearance may transpose into an open, organically appearing space while the player is observing the process.

B. Grammars for Mission Adaptation

Adapting the game's mission in response to behavior of the player is relatively challenging, as the game's mission needs to be tightly controlled by the game designer. Our proposed implementation of mission adaptation is analogous to that of space adaptation (see Subsection IX.A), in that first the player's behavior is classified, and dependent on the classification an adaptation grammar is applied.

Concretely, our adaptation grammars for mission adaptation define how nonterminal nodes in the predefined mission structure should be filled adaptively, i.e., dependent on behavior of the player. In the proposed implementation, the game designer envisions the general mission of a game level, and subsequently defines the mission rules to replace the nonterminals. Each adaptation rule matches a certain classification of the player behavior, and hence, dependent on the player classification, the resulting mission structure is adapted. This process is illustrated in Fig. 21. In the figure, the adaptation rule for either "player type *n*," "*m*," or "*o*" is applied to the adaptive nonterminal node "*?*," resulting in a player-dependent mission structure.

Indeed, one needs to realize that a multitude of player types may exist in reality, but that for reasons of feasibility, game developers may need to generalize the player's behavior in a limited set of player types. For instance, player type *n* exihibts aggressive behavior, player type *m* exhibits defensive behavior, and player type *o* uses foremost long-range weapons.

C. Grammars for Difficulty Scaling

Following the methods for space and mission adaptation described above, game designers can directly infuse expert knowledge on the game's difficulty into the concerning grammars. That is, terminating rules for adaptation of space and mission generation can be labelled as resulting in, for instance, an "easy," "normal," or "hard" level, where a numeric label of "0" indicates "very easy," and a numeric label of "100" indicates "very hard." The labelling in turn is used to probabilistically match adaptations to the classified player type, e.g., adaptations that result in a relatively "easy" level will never be applied for expert players, rarely for novice players, and often for beginners.

Indeed, this leaves intact a certain discretisation in difficulty levels, one that may not be applicable to every player, or one that game designers may want to infer dynamically depending on the performance of the player. To accommodate dynamically scaling the difficulty level while the game is being played, we define a grammar consisting of two sets of difficulty scaling rules. As an acknowledgement that difficulty is a multifaceted construct, we note that the grammar concerns a set of rules, instead of one rule to increase the difficulty. In our implementation of grammars for difficulty scaling, we implement one *rule set* for increasing the difficulty, and one *rule set* for decreasing the difficulty. Each rule in the set refers directly to one aspect of difficulty, and may be selectively applied to match the facets of the player's performance. For controllability, the decision of which adaptation rule to apply for which player type lies firmly in the hand of the game developer.

One may correctly note that difficulty arises from the combination and ordering of pieces in a level, rather than the existence of individual pieces themselves. In the current framework, a mechanism to precisely control the ordering of pieces according to difficulty is not included. However, as rules are selected on the basis of relative probability, and the selection is linked to the player level, game designers are able to loosely control the execution of "easy" rules for beginners, and "hard" rules for expert players. The combined interplay of difficulties working their way up the grammar, can herewith be controlled to some extent.

X. Prototype

We would like to highlight that the prototype in which we incorporate the ideas proposed in this paper, is available online, at http://www.jorisdormans.nl/missionspacegenerator.

TABLE I
GENERATIVE TECHNIQUES INCORPORATED IN THE PROTOTYPE

Generative technique	Incorporated
Generative grammars for mission adapation	Yes
Generative grammars for space adaptation	Yes
Establishing player models	No
Exploiting player models	No
Adaptable play experiences (Framework)	Yes
Adaptable play experiences (Playing modelling)	No

Table I lists which generative techniques have already been incorporated in the prototype. All described ideas regarding generative grammars and its application to: 1) generating missions; and 2) generating spaces, have been implemented in the prototype. A particular feature of the prototype is that it allows level designers to define and organize a level's mission via a highly controllable organic layout (see Fig. 13), on the basis of which a translated spacial construction is generated (see Figs. 15, 18, and 19). The ideas regarding establishing and exploiting player models are currently being implemented in the prototype, following the proposal discussed in Sections VII–IX. The framework for player-adaptive space and mission generation is already incorporated in the prototype. The use of player models in the player-adaptive framework is not yet incorporated.

XI. CONCLUSION

The levels of action adventure games, and numerous other games, are double structures consisting of both spaces and missions. When generating levels for this genre procedurally, it is best to break down the generation process in two steps; one for generating the game's space, and one for generating the game's mission. Generative graph grammars are suited to generate missions. They are capable of generating nonlinear 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 its 2-D graph representation can be used to construct a basic shape which can be then fleshed out using a shape grammar. What is more, the process can be easily controlled, the selection of different rule-sets can alter the appearance of a level drastically while there are many opportunities for a human designer to hand-pick rules during the process.

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 coherent experience. Furthermore, these techniques can be used to generate levels on the fly, allowing the game to respond to the player behavior. By means of establishing and exploiting player models, generative grammars can be used to scale dynamically the difficulty level to the player, and adapt the game space and game mission online, while the game is still being played, to ensure the game experience is tailored to the individual player. This opens up new opportunities for games and interactive storytelling.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their constructive comments that helped to improve the article considerably.

REFERENCES

[1] C. Pedersen, J. Togelius, and G. Yannakakis, "Modeling player experience for content creation," *IEEE Trans. Comput. Intell. AI Game.*, vol. 1, no. 2, pp. 121–133, Jun. 2009.
[2] M. Toy, G. Wichman, K. Arnold, and J. Lane, Rogue, 1980.
[3] G. Gygax and D. Arneson, Dungeons & Dragons. TSR, Wizards of Coast, 1974.
[4] E. Schaefer, D. Brevik, M. Schaefer, E. Sexton, and K. William, Diablo. Blizzard North, 1996.
[5] T. Baldree, Torchlight. Runic Games, 2009.
[6] W. Wright, Spore. Maxis, 2008.
[7] M. Persson and J. Bergensten, Minecraft. Mojang, 2009.
[8] J. Togelius, M. Preuss, and G. N. Yannakakis, "Towards multiobjective procedural map generation," in *Proc. Int. Conf. Found. Digital Games*, Monterey, CA, 2010.
[9] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proc. Int. Conf. Found. Digital Games*, Monterey, CA, 2010.
[10] M. Anderson, Dungeon-Building Algorithm. On Roguebasin. [Online]. Available: http://roguebasin.roguelikedevelopment.org/index.php?title=Dungeon-Building_Algorithm
[11] Grid Based Dungeon Generator. On Roguebasin. [Online]. Available: http://roguebasin.roguelikedevelopment.org/index.php?title=Grid_Based_Dungeon_Generator
[12] J. Babcock, Cellular Automata Method for Generating Random Cave-Like Levels. On Roguebasin. [Online]. Available: http://roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels
[13] E. Adams and A. Rollings, *Fundamentals of Game Design*, ser. Game Design and Development. Upper Saddle River, NJ: Prentice-Hall, Sep. 2006.
[14] G. Smith, M. Treanor, J. Whitehead, and M. Mareas, "Rhythm-based level generation for 2d platformers," in *Proc. Int. Conf. Found. Digital Games*, Orlando, FL, 2009, pp. 175–182.
[15] C. Ashmore and M. Nitsche, "The quest in a generated world," in *Proc. Digital Games Res. Assoc. Conf., Situated Play*, 2007, pp. 503–509.
[16] K. Hullett and J. Whitehead, "Design patterns in FPS levels," in *Proc. Int. Conf. Found. Digital Games*, Monterey, CA, 2010, pp. 78–85.
[17] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," in *Proc. Found. Digital Games Conf.*, 2010.
[18] B. Cousins, "Elementary game design," in *Develop Magazine*. Hertford, U.K.: Intent Media, 2004, vol. 10.
[19] Y. Koizumi and K. Usui, Super Mario Sunshine. Nintendo, 2002.
[20] W. Spector and H. Smith, Deus Ex. Nintendo, 2000.
[21] E. Aonuma, The Legend of Zelda: Twilight Princess. Nintendo, 2006.
[22] S. Miyamoto and T. Tezuka, Zelda Series. Nintendo, 1986.
[23] C. Kohler, Power-Up: How Japanese Video Games Gave the World an Extra Life. Bradygames, 2005.
[24] D. Cook, The Chemistry of Game Design. Gamasutra, 2007 [Online]. Available: http://www.gamasutra.com/view/feature/1524/the_chemistry_of_game_design.php
[25] N. Chomsky, Language and Mind (Extended Edition). 1972.
[26] C. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press, 2000.
[27] A. Van den Bosch, "Scalable classification-based word prediction and confusible correction," *Traitement Automatique des Langues*, vol. 46, no. 2, pp. 39–63, 2006.
[28] G. Kempen, *Natural Language Generation*. Dordrecht, The Netherlands: Martinus Nijhoff Publishers, 1987.
[29] R. de Beaugrande, "The story of grammars and the grammar of stories," *J. Pragmatics*, vol. 6, pp. 383–422, 1982.
[30] J. Golding, "Building blocks: Artist driven procedural buildings," presented at the GDC, San Francisco, CA, 2010.

[31] A. Lindenmayer, "Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs," *J. Theor. Biol.*, vol. 18, no. 3, pp. 300–315, 1968.
[32] M. Mozgovoy*, Algorithms, Languages, Automata, and Compilers: A Practical Approach*. Sudbury, MA: Jones & Bartlett, 2009.
[33] G. Ochoa, "On genetic algorithms and Lindenmayer systems," in *Parallel Problem Solving From NaturePPSN V*. Berlin, Germany: Springer-Verlag, 1998, pp. 335–344.
[34] D. Ashlock, K. Bryden, and S. Gent, "Creating spatially constrained virtual plants using L-systems," *Smart Eng. Syst. Design: Neural Netw., Evol. Program. Artif. Life*, pp. 185–192, 2005.
[35] Y. Parish and P. Müller, "Procedural modeling of cities," in *Proc. 28th Annu. Conf. Comput. Graphics Interact. Tech.*, Los Angeles, CA, 2001, pp. 301–308.
[36] D. Adams, "Automatic generation of dungeons for computer games," Ph.D. dissertation, Univ. Sheffield, Sheffield, U.K., 2002.
[37] J. Rekers and A. Schurr, "A graph grammar approach to graphical parsing," 1995, p. 195, vl.
[38] A. Saltsman, Canabalt. 2009 [Online]. Available: http://adamatomic.com/canabalt/
[39] M. M. G. Smith and J. Whitehead, "Tanagra: A mixed-initiative level design tool," in *Proc. Found. Digital Games Conf.*, Monterey, CA, 2010, pp. 209–216.
[40] R. Smelik, T. Turenel, K. J. de Kraker, and R. Bidarra, "Inegrating procedural generation and manual editing of virtual worlds," in *Proc. Found. Digital Games Conf.*, Monterey, CA, 2010.
[41] G. Stiny and J. Gips, "Shape grammars and the generative specification of painting and sculpture," *Inf. Process.*, vol. 71, pp. 1460–1465, 1972.
[42] D. Carmel and S. Markovitch, "Learning models of opponent's strategy in game playing," in *Proc. AAAI Fall Symp. Games: Plan., and Learn.*, Raleigh, NC, 1993, pp. 140–147.
[43] H. Iida, J. W. H. M. Uiterwijk, H. J. Van den Herik, and I. S. Herschberg, "Potential applications of opponent-model search. Part 1: The domain of applicability," *Int. Comput. Chess Assoc. J.*, vol. 16, no. 4, pp. 201–208, 1993.
[44] H. H. L. M. Donkers, J. W. H. M. Uiterwijk, and H. J. Van den Herik, "Probabilistic opponent-model search," *Inf. Sci.*, vol. 135, no. 3–4, pp. 123–149, 2001.
[45] H. H. L. M. Donkers, "Nosce Hostem—Searching with opponent models," Ph.D. dissertation, Faculty Humanities Sci., Maastricht Univ., Maastricht, The Netherlands, 2003.
[46] H. J. Van den Herik, H. H. L. M. Donkers, and P. H. M. Spronck, "Opponent modelling and commercial games," in *Proc. IEEE Symp. Comput. Intell. Games*, G. Kendall and S. Lucas, Eds., New York, 2005, pp. 15–25.
[47] J. Fürnkranz, "Recent advances in machine learning and game playing," *ÖGAI J.*, vol. 26, no. 2, pp. 19–28, 2007.
[48] R. Houlette, "Player modeling for adaptive games," in *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, 2004, pp. 557–566.
[49] I. Millington*, Artificial Intelligence for Games*. San Francisco, CA: Morgan Kaufmann, 2006.
[50] H. H. L. M. Donkers and P. H. M. Spronck, "Preference-based player modeling," in *AI Game Programming Wisdom 3*, S. Rabin, Ed. Hingham, MA: Charles River Media, 2006, pp. 647–659.
[51] F. D. Laramée, "Using n-gram statistical models to predict player behavior," in *AI Game Programming Wisdom*, S. Rabin, Ed. Hingham, MA: Charles River Media, 2002, pp. 596–601.
[52] G. Yannakakis, M. Maragoudakis, and J. Hallam, "Preference learning for cognitive modeling: A case study on entertainment preferences," *IEEE Trans. Syst. Man Cybern. A, Syst. Humans*, vol. 39, no. 6, pp. 1165–1175, Jun. 2009.
[53] J. Denzinger and J. Hamdan, "Improving modeling of other agents using tentative stereotypes and compactification of observations," in *Proc. IEEE/WIC/ACM Int. Conf. Intell. Agent Technol.*, J. Liu and N. Cercone, Eds., New York, 2004, pp. 106–112.
[54] G. Lankveld, S. Schreurs, and P. Spronck, "Psychologically verified player modelling," in *Proc. 10th Int. Conf. Intell. Games Simulation*, L. Breitlauch, Ed., Ghent, Belgium, 2009, pp. 12–19.
[55] G. Yannakakis and J. Hallam, "Real-time game adaptation for optimizing player satisfaction," *IEEE Trans. Comput. Intell. AI Games*, vol. 1, no. 2, pp. 121–133, Jun. 2009.
[56] M. Ryan*, Narrative as Virtual Reality: Immersion and Interactivity in Literature and Electronic Media*. Baltimore, MD: Johns Hopkins Univ. Press, 2001.
[57] P. H. M. Spronck, I. G. Sprinkhuizen-Kuyper, and E. O. Postma, "Difficulty scaling of game AI," in *Proc. 5th Int. Conf. Intell. Games Simulation*, A. E. Rhalibi and D. Van Welden, Eds., Ghent, Belgium, 2004, pp. 33–37.
[58] J. K. Olesen, G. N. Yannakakis, and J. Hallam, "Real-time challenge balance in an RTS game using rtNEAT," in *Proc. IEEE Symp. Comput. Intell. Games*, P. Hingston and L. Barone, Eds., New York, 2008, pp. 87–94.
[59] P. Demasi and A. J. de. O. Cruz, "Online coevolution for action games," *Int. J. Intell. Games Simulation*, vol. 2, no. 3, pp. 80–88, 2002.
[60] R. Hunicke and V. Chapman, "AI for dynamic difficulty adjustment in games," in *Proc. AAAI Workshop Challenges Game Artif. Intell.*, Menlo Park, CA, 2004, pp. 91–96.
[61] G. N. Yannakakis and J. Hallam, "Towards optimizing entertainment in computer games," *Appl. Artif. Intell.*, vol. 21, no. 10, pp. 933–971, 2007.
[62] S. C. J. Bakkes, P. H. M. Spronck, and H. J. Van den Herik, "Rapid and reliable adaptation of video game AI," *IEEE Trans. Comput. Intell. AI Games*, vol. 1, no. 2, pp. 93–104, 2009.
[63] S. C. J. Bakkes, P. H. M. Spronck, and H. J. Van den Herik, "Opponent modelling for case-based adaptive game AI," *Entertain. Comput.*, vol. 1, no. 1, pp. 27–37, 2009.
[64] H. Kamiya, Ōkami. Clover Studio, 2006.

Joris Dormans is currently working towards the Ph.D. in game design.

He is currently a Lecturer of Game Development at the Computer Science Department, Amsterdam University of Applied Sciences (HvA), Amsterdam, The Netherlands. He also works as a freelance and independent Game Designer. His designs and research focuses on emergent gameplay, automatic level design, and formal description of game rules. As a designer, he worked on two published board games and several (serious) online games.

Sander Bakkes received the Ph.D. degree in artificial intelligence in video games.

He is currently a Postdoctoral Researcher and Teacher at the Amsterdam University of Applied Sciences (HvA), Amsterdam, The Netherlands. In previous work, he investigated the rapid adaptation of character behavior in complex video-game environments. At present, his research is focussed on incorporating player-modeling techniques in procedurally generated environments, for the purpose of personalizing game experiences, and increasing player satisfaction.