INVOLVING PLAYER EXPERIENCE IN DYNAMICALLY GENERATED MISSIONS AND GAME SPACES

Sander Bakkes and Joris Dormans
Amsterdam University of Applied Sciences
Computer Science Dept., section Game Development / CREATE-IT Applied Research
P.O. Box 1025, NL-1000 BA Amsterdam, The Netherlands
e-mail: {s.c.j.bakkes, 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 role-playing games for which procedural level generation has been successful in the past. The approach outlined by this paper distinguishes between missions and spaces as 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.

INTRODUCTION

In the domain of video games, procedurally generated content is considered to be of great 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, though in a severely limited form.

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 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. Although these algorithms have a proven track-record for the creation of (relatively uncomplicated) rogue-like games, the gameplay their output supports is rather limited.

What is more, Kate Compton and Michael Mateas point out that generating levels for a generally relatively complex action platform game is more difficult as level design is a far more critical aspect of that type of game [2]. 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 rogue-like games. The 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 sufficient. 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. 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.

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 authors and described in more detail elsewhere [3], two different structures emerge that both describe the level. 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 pre-

ceding 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 ultimately 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 a common strategy to gradually train the player in the available moves and techniques. In addition, numerous missions follow a Hollywood-like structure, that can be attributed to Joseph Campbell's monomyth (*cf.* [4]).

The spatial qualities of the Forest Temple are distinct. 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 [5].

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.

Concept. Generative grammars originate in linguistics where they are used as a model to describe sets of linguistic phrases. 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 $\rightarrow$ 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. 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 analyse and classify language. Moreover, grammars are suited for predicting, and generating automatically language phrases. We utilise grammers 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 non-terminal symbols that can directly or indirectly result in the application of the same rule recursively. The rule 'S $\rightarrow$ abS' is an example of a recursive rule and will produce endless strings of ab's. The rule 'S $\rightarrow$ 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 $\rightarrow$ Obstacle + treasure
2. Obstacle $\rightarrow$ key + Obstacle + lock + Obstacle
3. Obstacle $\rightarrow$ monster + Obstacle
4. Obstacle $\rightarrow$ 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.

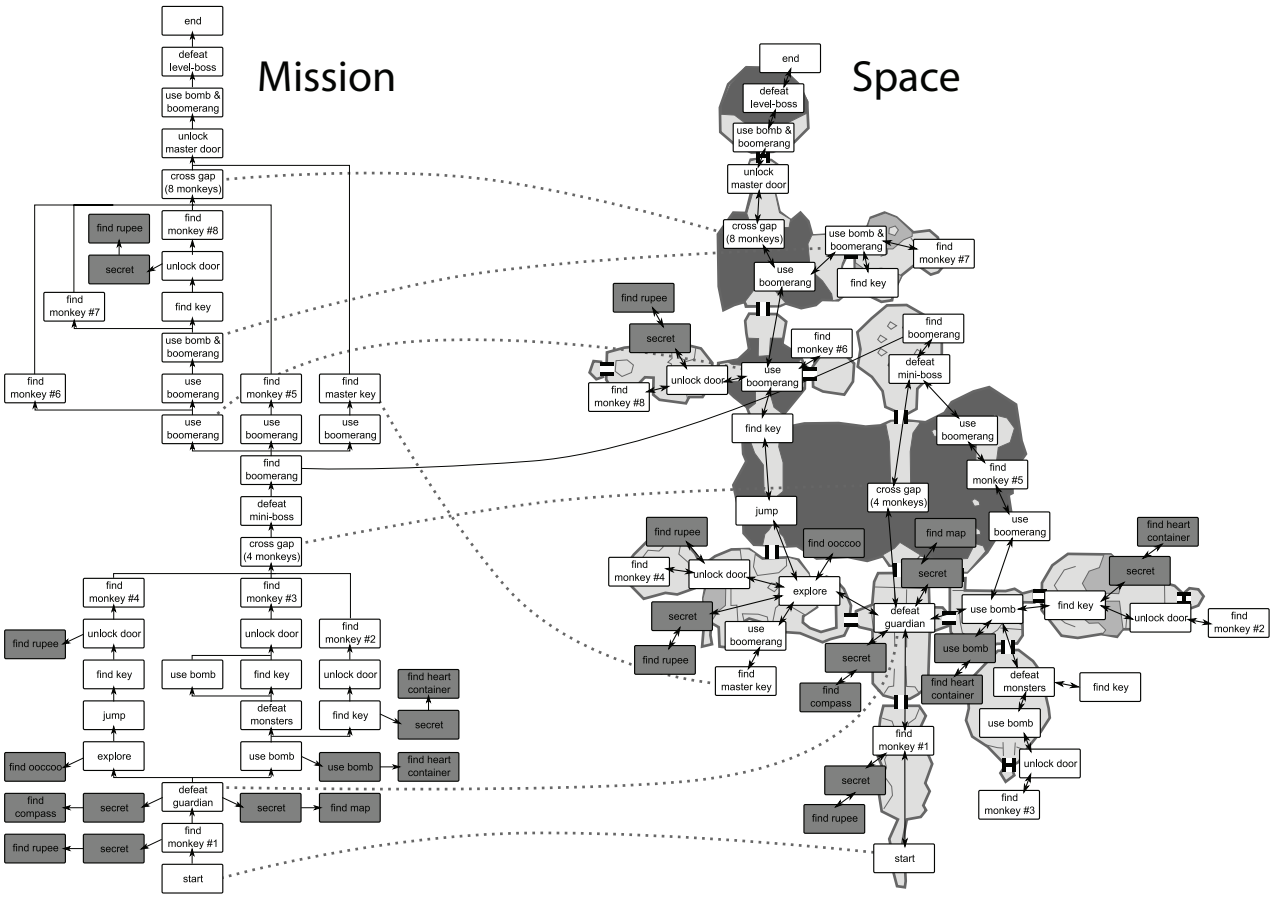

Figure 1: Mission and space in the Forest Temple Level of *The Legend of Zelda: The Twilight Princess*.

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* above.

Advantages 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 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 [6]. 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 Lindermayer Systems (L-Systems), for instance for the procedural generation of city models [7].

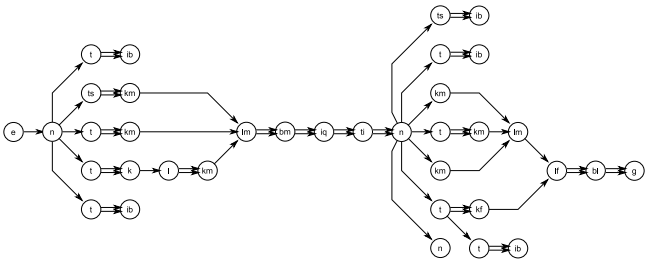

Figure 3: Example of a generated mission, based on the rules given in Figure 2.

GRAPH GRAMMAR TO GENERATE MISSIONS

Graph grammars are a specialized form of generative grammars that produce graphs consisting of edges and nodes, instead of producing strings. In relation with level generation, graph grammars are discussed by David Adams in his 2002 thesis *Automatic Generation of Dungeons for Computer Games* [8]. In a graph grammar, one or several nodes and interconnecting edges can be replaced by a new structure of nodes and edges. For examples we refer the reader to [3].

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 2 proposes several rules to generate a mission structured similarly as the mission of the forest temple. Figure 3 shows sample output of the graph grammar. We 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 that the subordinate can be placed anywhere. This information is very important for the space generation algorithm (see section *Generating Space from Mission*).

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 [9]. 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 4a). 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 wall (effectively closing the connection), a short piece of corridor, or a T-fork (see Figure 4b). The construction depicted in Figure 4c is a possible output of these rules, provided that the start symbol also was a connection, and given that at every iteration a random connection was selected to be replaced.

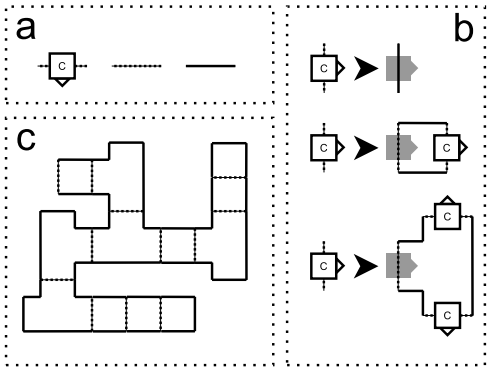

Figure 4: Shape grammar a) alphabet, b) rules, and c) output.

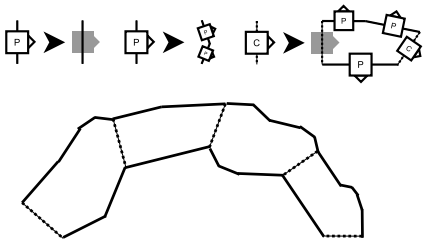

Figure 5: Recursive shape rules and its output.

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 Figure 5 are recursive and the shapes these rules produces will have a more natural (fractal) feel.

GENERATING SPACE FROM MISSION

In order to use a shape grammar to generate a space from a generated mission a few adjustments need to be made to the shape grammar. The terminal symbols in

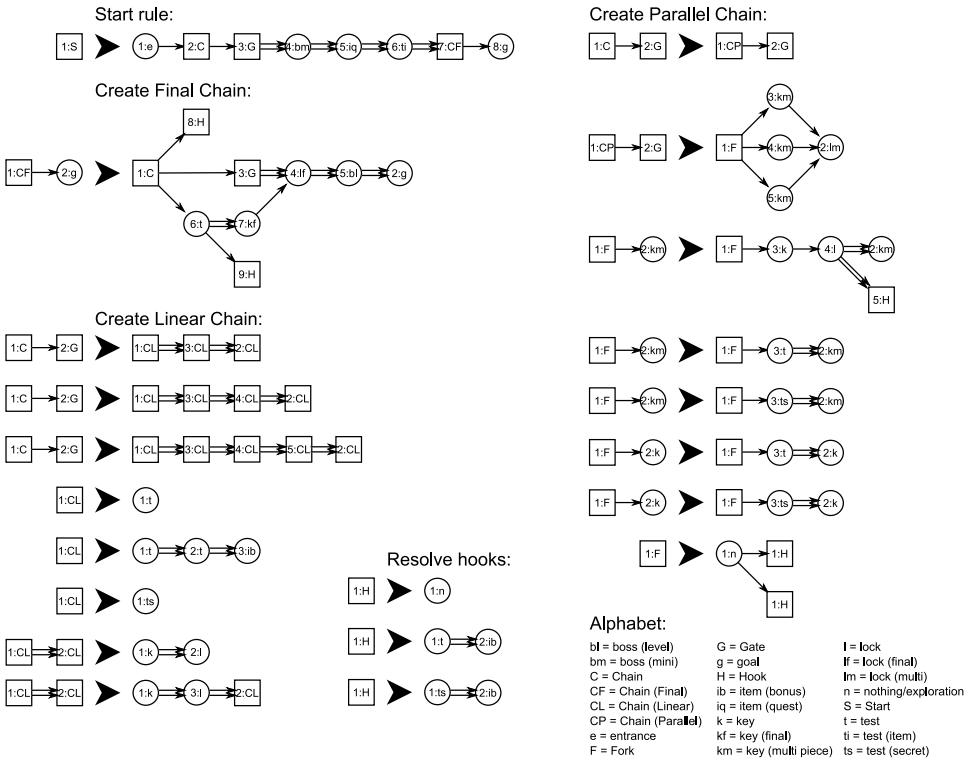

Figure 2: Example of rules to generate a mission.

the mission need to function as building instructions for the shape grammar. To achieve this, each rule in the shape grammar is associated with a terminal symbol form in the mission grammar. The prototype that implements the shape grammar first finds the next symbol in the mission, looks for rules that implement that symbol, selects one pseudo-randomly based on their relative weight, then looks for possible locations where the rule could be applied, and finally selects one location pseudo-randomly based on their relative fitness (one location might be more suitable than another).[1] 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, e.g., tests or locks as specified by the mission. The shape grammar is further extended by 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 prototype application supporting this research,

[1]The prototype in which we incorporate the ideas proposed in this paper, is available online, at http://www.jorisdormans.nl/dungeoncrawl

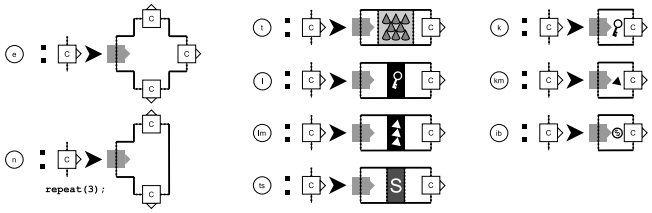

Figure 6: Shape grammar rules to generate missions.

rules can have commands associated with them. These commands are executed either before or after the application of a rule. These commands, among other functions, facilitate dynamic rule weights and progressive difficulty.

Once the complete mission is accounted for, the shape grammar will continue to iterate until all non-terminals are replaced with terminal symbols using a set of rules designed to finalize the space (or, alternatively, to grow some additional branches). Figure 6 lists several rules for a shape grammar constructed in this way. Figure 7 illustrates a few iterations in the construction of a level based on the first part of the mission presented in Figure 3 above.

In practise, it is not be very difficult to generate maps that can accommodate multiple missions. Missions can be blended, with the generator alternating between missions when selecting the next task to accommodate on

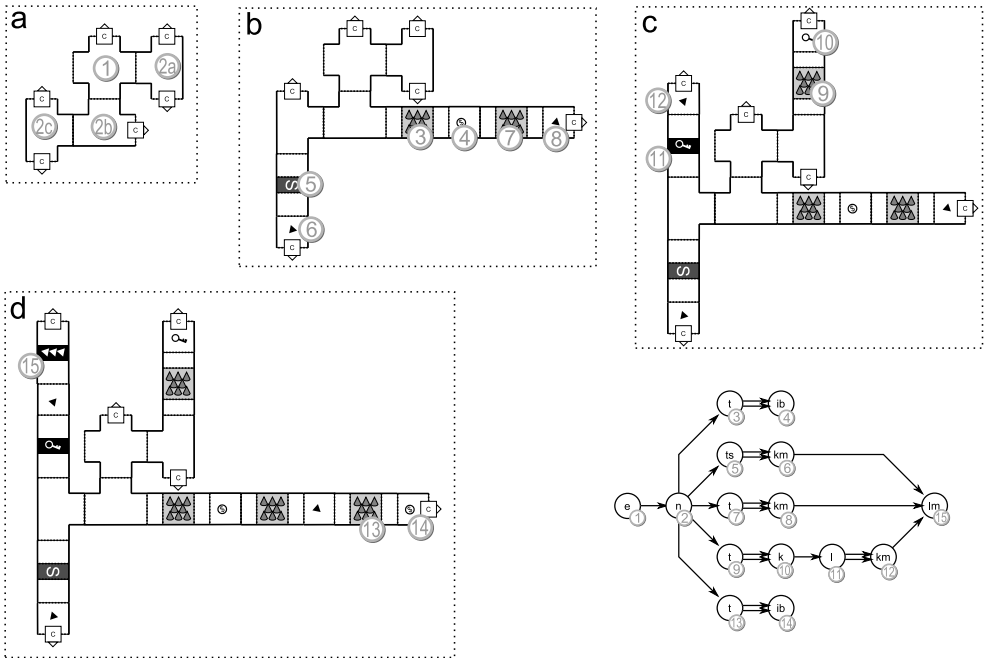

Figure 7: Space generation using the rules from Figure 6 and part of the mission from Figure 3 (depicted in the lower-right), over four iterations.

the map. Alternatively, a second mission is used as building instructions after the first mission has been completely accounted for.

ESTABLISHING PLAYER MODELS

The generation techniques discussed in this paper can also be employed to generate (or adapt) levels during play, allowing for the opportunity to *let behaviour of the player impact the generation*. To this end, techniques for player modelling need to be incorporated.

Player modelling 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 behaviour in a game. In the case the models concern not the human player, but instead an opponent player, we speak of 'opponent modelling'. The goal of opponent modelling is to raise the playing strength of the (computer-controlled) player by allowing it to adapt to its opponent and exploit his weaknesses [10]. On the other hand, the goal of player modelling generally is to steer the game towards a predictably high player satisfaction [11], on the basis of modelled behaviour of the human player.

Player modelling is of increasing importance in modern video games [12]. Houlette [13] discussed the challenges of player modelling in video-game environments, and suggested some possible implementations of player modelling. A challenge for player modelling 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. As by approximation only twenty percent of computing resources are available to the game AI [14], only computationally inexpensive approaches to player modelling are suitable for incorporation in the game.

For the domain of modern video games, we deem three approaches applicable to player modelling, namely (1) action modelling, (2) preference modelling, and (3) player profiling.

Action modelling. In video games, a common approach to establishing models of the player is by modelling the *actions* of the player [15]. It has been shown that it is possible to create a model of the actions that players tend to take in particular game situations [13]. 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 [15]. What is more, the interest of game developers generally goes out to de-

termining a player's actual game experience, which, by implication, is even less easily determined solely on the basis of action-based models.

Preference modelling. 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 [15] and identifies the model of a player by analysing the player's choices in predefined game situation. In the preference-based approach, player modelling 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. Behaviour of related game AI is established based on the classification of the player. Modelling of preferences may be viewed as similar to approaches that regard known player models (1) as stereotypes, and (2) as an abstraction of observations [16]. As a means to generalise over observed game actions, a preference-based approach may be of interest to game developers.

Player profiling. A recent development with regard to player modelling, is to establish automatically psychologically verified *player profiles*. Player profiling has gathered substantial research interest (*cf.*, e.g., [1]). Van Lankveld [17] states that the major differences between player modelling (by means of action modelling, or preference modelling) and player profiling, lie in the features that are modelled. That is, player modelling 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.

EXPLOITING PLAYER MODELS

Here, we propose three approaches for exploiting player models in the context of generative grammars, namely for (1) space adaptation, (2) mission adaptation, and (3) difficulty scaling.

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 behaviour of the player. Firstly, after observing the player for some time, features within the established player model may indicate that it is interesting to transform (gradually) the game surroundings. For instance, from open to confined spaces, from linear to more organic environments, and from easily maneuverable corridors to intricate mazes. Secondly, varied gameplay may be provided by also allowing events that take place within certain game spaces (e.g., particular rooms) to respond to the player's previous behaviour. For instance, if the player already has encountered and fought many monsters, the rules that would generate more monsters might decrease in weight while rules that would generate obstacles of a different type might increase in weight. Or, inversely, when the player model indicate that the player enjoys combating monsters (for example because he goes after every monster he can find), the game may confront him with more and tougher creatures. The possibilities of a feedback loop between the actual performance of the player and the online generation of the game, are many.

Mission adaptation. An interesting alternative to space adaptation, is to allow the game's mission to grow in response to the actual behaviour of the player. A strategy in this regard, which we regard as promising, 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 his attention to certain parts of the story [18].

Difficulty scaling. Player models may potentially be applied to adapt automatically the challenge that a game poses to the skills of a human player. This is called difficulty scaling [19], or alternatively, challenge balancing [20]. 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 behaviour, 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 [21] used coevolutionary algorithms to train game characters that best fit the challenge level of a human

player. Hunicke and Chapman [22] 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.* [19] 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 [23] 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. Generally, we 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. For instance, the grammars could 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 frustratingly many powerful opponents.

CONCLUSIONS

The levels of action adventure games, and numerous other 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 behaviour. 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 opportunities for gaming and interactive storytelling that hitherto have hardly been examined.

REFERENCES

[1] Pedersen, C., Togelius, J., Yannakakis, G.: Modeling Player Experience for Content Creation. IEEE Transactions on Computational Intelligence and AI in Games **1**(2) (2009) 121–133

[2] Compton, K., Mateas, M.: Procedural level design for platform games. In: Proceedings of the AIIDE. (2006)

[3] Dormans, J.: Adventures in level design: generating missions and spaces for action adventure games. In: Proc. of the 2010 Works. on Proc. Content Generation in Games. (2010) 1–8

[4] Vogler, C.: The writer's journey: Mythic structure for writers. Michael Wiese Productions (1998)

[5] Ashmore, C., Nitsche, M.: The Quest in a Generated World. In: Proc. 2007 DiGRA Conference: Situated Play. (2007) 503–509

[6] Golding, J.: Building blocks: Artist driven procedural buildings (2010) Presentation at GDC 10, San Fran., CA.

[7] Parish, Y., Müller, P.: Procedural modeling of cities. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. (2001) 301–308

[8] Adams, D.: Automatic Generation of Dungeons for Computer Games. University of Sheffield Dissertation (2002)

[9] Stiny, G., Gips, J.: Shape grammars and the generative specification of painting and sculpture. Information processing **71** (1972) 1460–1465

[10] Carmel, D., Markovitch, S.: Learning models of opponent's strategy in game playing. In: Proceedings of AAAI Fall Symposium on Games: Planning and Learning. (1993) 140–147

[11] Van den Herik, H.J., Donkers, H.H.L.M., Spronck, P.H.M.: Opponent modelling and commercial games. In: Proceedings of the CIG'05. (2005) 15–25

[12] Fürnkranz, J.: Recent advances in machine learning and game playing. ÖGAI-Journal **26**(2) (2007) 19–28

[13] Houlette, R.: Player modeling for adaptive games. In: AI Game Programming Wisdom 2. (2004) 557–566

[14] Millington, I.: Artificial Intelligence for Games. Morgan Kaufmann Publishers, San Francisco, California, USA (2006)

[15] Donkers, H.H.L.M., Spronck, P.H.M.: Preference-based player modeling. In Rabin, S., ed.: AI Game Programming Wisdom 3. (2006) 647–659

[16] Denzinger, J., Hamdan, J.: Improving modeling of other agents using tentative stereotypes and compactification of observations. In: Proceedings of the IAT 2004. (2004) 106–112

[17] Lankveld, G., Schreurs, S., Spronck, P.: Psychologically verified player modelling. In: Proceedings of the GAMEON'2009. (2009) 12–19

[18] Ryan, M.: Narrative as virtual reality: Immersion and interactivity in literature and electronic media. Johns Hopkins University Press Baltimore, MD, USA (2001)

[19] Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., Postma, E.O.: Difficulty scaling of game AI. In: Proceedings of the GAMEON'2004. (2004) 33–37

[20] Olesen, J.K., Yannakakis, G.N., Hallam, J.: Real-time challenge balance in an RTS game using rtNEAT. In: Proceedings of the CIG'08. (2008) 87–94

[21] Demasi, P., Cruz, A.J.de.O.: Online coevolution for action games. International Journal of Intelligent Games and Simulation **2**(3) (2002) 80–88

[22] Hunicke, R., Chapman, V.: AI for dynamic difficulty adjustment in games. In: AAAI Workshop on Challenges in Game Artificial Intelligence. (2004) 91–96

[23] Yannakakis, G.N., Hallam, J.: Towards optimizing entertainment in computer games. Applied Artificial Intelligence **21**(10) (2007) 933–971