Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation

Daniël Karavolos
Amsterdam University of Applied Sciences
Duivendrechtsekade 36-38
1096 AH Amsterdam
The Netherlands
k.d.karavolos@hva.nl

Anders Bouwer
Amsterdam University of Applied Sciences
Duivendrechtsekade 36-38
1096 AH Amsterdam
The Netherlands
a.j.bouwer@hva.nl

Rafael Bidarra
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands
r.bidarra@tudelft.nl

ABSTRACT

A level designer typically creates the levels of a game to cater for a certain set of objectives, or mission. But in procedural content generation, it is common to treat the creation of missions and the generation of levels as two separate concerns. This often leads to generic levels that allow for various missions. However, this also creates a generic impression for the player, because the potential for synergy between the objectives and the level is not utilised. Following up on the mission-space generation concept, as described by Dormans [5], we explore the possibilities of procedurally generating a level from a designer-made mission. We use a generative grammar to transform a mission into a level in a mixed-initiative design setting. We provide two case studies, dungeon levels for a rogue-like game, and platformer levels for a metroidvania game. The generators differ in the way they use the mission to generate the space, but are created with the same tool for content generation based on model transformations. We discuss the differences between the two generation processes and compare it with a parameterized approach.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General—*Games*

General Terms

Game Design, Procedural Content Generation, Generative Grammars, Automated Game Design, Mixed-Initiative Design, Level Generation

1. INTRODUCTION

Modern computer games often contain a large amount of content, which is typically organised in levels, to ensure narrative development, variation, and (re)playability. Generating this content takes a lot of time and effort, even for experienced game designers. To deal with these problems, researchers have been looking at procedural content generation [22, 17] and automated game design [5, 3, 11, 24, 2]. Both focus on automatically generating game content, and, if done correctly, support the human designer by reducing the amount of time involved in the operational development of games.

Research on procedural content generation often differentiates the game content into different objects, for example, a certain data structure, that can be created and/or modified automatically to generate variations that are in some sense similar to the original object. Ideally, but not always, these variations should ensure replayability of the game with varying levels of difficulty [25]. While the quality of game levels generated in such a way has long been high enough to be applicable in popular games, such as Rogue [23], The Sentinel [4], Sid Meier's Civilization [13], Diablo [16], Minecraft [14] and Spelunky [15], the quality (in terms of user experience) is still considered lower than those created by human game designers [21]. Shortcomings of generated levels include low reliability, high predictability, low visual quality and believability, and are, in general, difficult to evaluate in terms of quality.

To ensure a higher level of quality, it is important to consider other game aspects, such as narration or the underlying game mechanics, in the generation of levels. For instance, if the generated level includes no challenge for the player, there is a large chance it will not be seen as a positive experience. This is a topic often addressed by automated game design research, which has the ambition to generate game levels in conjunction with other aspects of the game, such as the goal, mission, narrative, or theme, visual style, and/or the rules of game play [2]. These approaches tend to be explorative, as it is not yet clear how game aspects can best be used in the automatic generation of levels [11, 24].

Instead of taking the perspective that computers should automatically generate the content of the games on their own, we feel it is important to understand how the human designer and the algorithms used to generate content can work together [18]. Therefore, in this work, we take a different approach, by looking at the automatic generation of game content as a collaborative mixed-initiative design process, similar to work by Smith et al. [20]. In their work, a human designer can manipulate a level generator through a set of

parameters, which will then provide different level configurations for the designer to choose from. Other researchers, such as Mawhorter & Mateas [12], instead propose a mixed-initiative approach to generating platformer levels based on the assembly of manually designed level chunks.

In this paper, we describe two case studies in game level design (for a top-down dungeon quest game and a side-view platform game) to explore the underlying design processes in terms of collaboration between human designers and computer algorithms. To this end, we have created explicit models of the design processes involved, using the LudoScope tool [5, 6], which is able to transform conceptual design ideas about missions and space to concrete game levels for actual playable games in interaction with a human designer. In this approach, we use the strength of computers to systematically employ rules for transformations, combine multiple inputs, and generate alternatives, but also acknowledge the skill of the designer in exploring sources of variation, spotting potential problems, recognizing opportunities, and finding creative solutions to problems.

In section 2, we discuss related work on graph-based and tile-based transformations, and the Ludoscope tool. Section 3 and 4 describe case study 1 and 2, respectively, in terms of the design process and the targeted game content. In section 5, we discuss the differences and the resulting lessons learned. Section 6 finally presents concluding remarks and recommendations for future research.

2. RELATED WORK

Both of our case studies use a different approach to create game content through a cooperation between designer and algorithm. These approaches are best classified as 'graph-based transformations' and 'tile-based transformations'. In the coming subsections we will first explain how these transformations work and how they have been incorporated in previous work. After this, we will briefly introduce Ludoscope [5, 6, 8], the tool used to formalize the game design flow.

2.1 Graph-based Transformations

Van der Linden et al. [25, 26] proposed a graph grammar method for procedurally generating dungeons, and implemented it for a game called Dwarf Quest [28] based on a graph of player actions. This generator has two stages, graph creation and transforming the graph into game space (or layout solving). The graph is generated by a graph grammar, which contains information about the mission in the level. For example, a level always contains a boss fight and a part leading up to it.

Each node stands for a specific player action, e.g. fighting a monster or looting a chest. Generally there is only one player action per room. However, there are two exceptions. The first is that fighting a monster and picking up its dropped key are two actions, but take place in the same room. The second exception is that there can be filler rooms. To transform the action graph into a level, the algorithm performs some layout solving, to fit the graph into a 2D grid according to the games' constraints. This can introduce filler rooms that are not essential to the gameplay, but which could potentially contain filler actions or enemies.

The grammar of Van der Linden et al. [25] also contains recursive rules, for example a recursive puzzle node can be transformed into a recursive puzzle node and a puzzle. This allows the system to create graphs of arbitrary length, which can be controlled by a parameter that affects the firing probability of this recursive rule. Concurrently, the recursion creates a hierarchy that contains information about the graph, for example which keys and locks are connected. This would allow the generator to insert actions between picking up the key and unlocking the lock, or even create nested key-lock mechanisms.

The algorithm has several parameters for the action graph: length, branching, difficulty, and seeds can be set to fix the layout solving algorithm. However, this is limited control. The designer cannot make minor changes, such as removing/adding a single action, or adding/removing edges between nodes. The algorithm can only be run again, with potentially a completely different layout as result.

In section 3 we will describe an approach that allows the user to change the design at multiple stages in the generation process.

2.2 Tile-based Transformations

Spelunky [15] is a platformer that contains rogue-like elements, such as permadeath and procedurally generated levels. The level generation system is based on the principle of dividing a grid into rooms, selecting rooms that will be part of the player's necessary path, and filling in the details of the rooms based on templates with probabilistic elements [10]. After generating the level geometry, the enemies are placed in a similar fashion. The generator uses a grammar to transform abstract tile types, represented by numbers and letters, into the tiles that can be instantiated by the game engine.

This level generator has several interesting characteristics. First, the symbols used to describe the outline of the level contain information about the connections of the rooms. For example, a '1' means that a room only has connections to rooms left and right, whereas a '2' means that a room is guaranteed to also have upward and downward connections. This allows assumptions about the neighbors of a symbol without actually having to check these neighbors.

Second, many templates contain probabilistic blocks and probabilistic tiles. These probabilistic blocks have their own set of templates, which can again contain various types of probabilistic tiles. After applying the block templates, the remaining tiles can be transformed into air, ground or spikes, creating minor variations in the rooms. Together, these options create a multitude of different level configurations.

Finally, as shown in [10], these transformation steps are split up into separately executable modules. This could potentially be used to enable the designer to change the minor variations without changing the room templates, or change the room template without changing the path, if he/she so desires, but this potential is not employed in Spelunky - the level generation system is fully automatic, without any (need for) interaction with a human designer.

Despite its success, Spelunky's level generator has several limitations. It can only generate a level in a 4x4 grid, and the path to the exit is mostly linear. This precludes some interesting game-flow patterns such as the deliberate generation of multiple paths, or non-linear traversals of the level through a set of lock-and-key pairs.

2.3 Ludoscope

Ludoscope [5, 6, 8] is a mixed-initiative design tool based on the principles of model driven engineering and generative grammars. It promotes the approach of creating a content generator by making a model of the design process. This comes down to breaking down the generation process into small, separately executable steps, called modules. In Ludoscope, the game content - in our case a level - is defined as an expression that is generated by a grammar. This grammar can be based on strings, tile maps, voronoi diagrams, graphs, or shapes. The designer can define the alphabet of the grammar, i.e. the elements of an expression, and a set of transformation rules, i.e. rules that can rewrite parts of an expression. Furthermore, the designer can define a (parallel) sequence of modules that transform the expression based on their grammar. Each of these modules receives input from either another module or some starting expression. Then, the module can either execute a set of rules probabilistically, execute a set of rules according to some user-defined recipe, or allow the designer to make changes to the expression. The output is either sent directly to another module for further processing, sent to a binary check that chooses a module based on some user-defined (logical) condition, or presented to the designer as the final result.

3. CASE STUDY 1: DWARF QUEST

The first case study is Dwarf Quest [28], a dungeon crawler with turn-based combat, created in Unity3D[1]. In this game, the player explores a set of connected rooms in search for loot and new equipment. A room can contain one of the following elements: a trap, loot, monsters, a locked door or bridge, or a lever or a hidden key. The bridges and the locked doors are considered as puzzles, because they require the player to find the correct path to the corresponding lever or key. There are also empty rooms (fillers), which contribute to the maze-like dungeon feeling and facilitate the orthogonality of the map.

In the approach of Van der Linden et al. [25], described in section 2.1, the dungeons are created with one button press, after setting the desired parameter values. Both the creation of the action graph and the conversion to playable levels is done in working memory, without feedback to the user. This is a fine approach for an endless mode at runtime, but we consider the case of using procedural content generation at development time. By separating the creation of the action graph from the conversion to the game level, we can allow the game designer to make changes to the flow of a level without having to consider all the details. Moreover, the designer can store graphs as seeds for game levels, or even use a selection of graphs to create a story line. This increases the influence of the designer, while maintaining the benefits of procedural generation.

[1]http://unity3d.com/

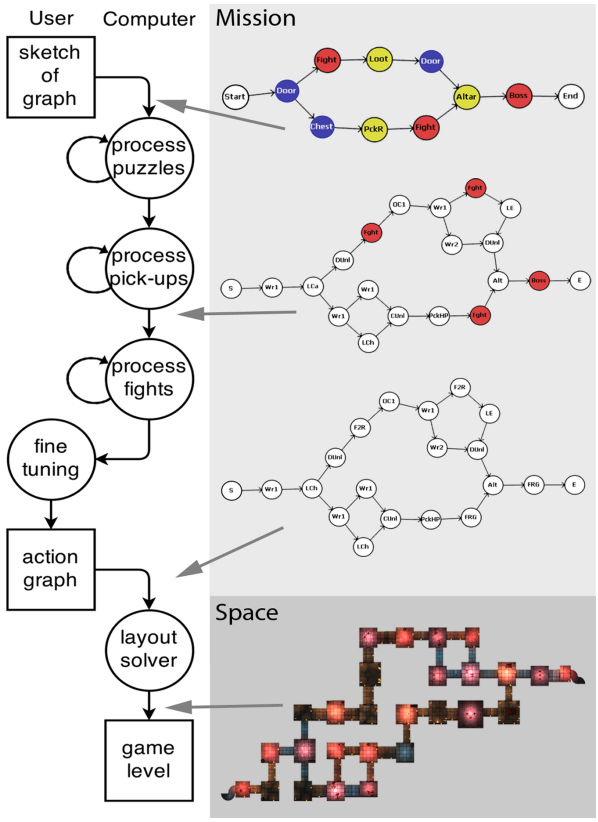

Figure 1: A model of the action graph generation process. Left: The squares represent meaningful products, the circles represent transformation modules. Right: These figures show the data representation during the generation process. The colored nodes in the graphs can be transformed, the white nodes are leaf nodes. The bright rooms in the resulting game level are based on the graph, the dark rooms are fillers created by the layout solver.

3.1 Generation Process

Figure 1 shows a model of the generation process and how the representation looks during certain stages of the design process. In this case study we have chosen to have the system ask the designer for a sketch of an action graph and expand that into a detailed graph. This detailed graph is shown to the designer, allowing him to make manual changes. To transform this graph into a level for Dwarf Quest, this graph is fed to the layout solver of Van der Linden et al. [25]. In more detail, we can discern the following steps:

1. The first module allows transformations by the user. A generative grammar needs an initial phrase (or in this case graph) to transform. The generator provides an example for an initial graph: Start -> Puzzle -> Boss -> End. The designer can modify this graph by adding, removing, and connecting non-terminal nodes. The nodes describe abstract player tasks. Possibilities are: battles (red), boss battles (red), puzzles with a locked door (blue), puzzles with an open bridge (blue), traps (blue), various item pickups (yellow), and an altar for

buying items (yellow). A valid level needs at least the terminal nodes 'Start' and 'End'.

2. The computer will then process the puzzles, pickups and battles, respectively. This order is important, because puzzles can contain pickups and battles. The available transformation rules depend on a difficulty parameter, which is manipulable by the designer, even between executing modules.

3. The result of the processing steps is a detailed graph, which contains only terminal nodes. These nodes loosely correspond to rooms in the dungeon. The user can make changes with the 'FineTuning' module.

4. The quest is saved as a text file, which is parsed and interpreted by a script in Unity. The game engine converts each node into a room with the required elements. Exceptions are the nodes concerning fighting and looting enemies for keys–these actions must take place in the same room.

5. Finally, the layout solver defines the location of each room, adds paths and empty rooms to fulfil some game-specific constraints, e.g., the rooms must fit nicely in a grid and can have at most four exits.

We could perform all automatic graph transformations in one module. However, we have chosen to increase the influence of the designer by separating the application of the rewrite rules for the puzzles, the loot and the monster encounters. This way the designer can, for example, change the loot and the fights without changing the puzzles.

Note that, since Ludoscope produces a text-file, it is not possible to connect nodes through a data structure. However, we do need to connect certain game elements in the game to make them function properly. Therefore, we have added ID tags in the form of members to the symbols for keys and locks, and fighting and looting, e.g. `LootKeyFromChest(keylockID = 4)` and `UnlockDoor(keylockID = 4)`. This extra information can be used by the parser in the game engine to connect the nodes.

4. CASE STUDY 2: TICKTICK++

The prototype TickTick++ is based on the game TickTick [9], a small 2D side-view platformer game with a custom game engine based on Monogame[2]. In TickTick++ the player must reach the end of a level within a given time limit. The player is obstructed by obstacles and enemies. To overcome these challenges, the player must find three power-ups. Each of these power-ups can be considered as a key that unlocks a new part of the level.

4.1 Generation Process

The main difference between the platform level generator and the dungeon generator is the data type of the grammar. The quest grammar is based on graphs, whereas the platform grammar is based on tiles. This is mainly because the game engine has a level generator based on a tile parser. Transforming a graph into a level for a side-view platformer would either require creating templates in game code, such as in case study 1, or an elaborate process of transforming the graph into a grid of tiles. The advantage of using tiles is that it can be parsed as a level by the game engine without further transformations.

The disadvantage of using tiles is that it is not an intuitive representation for specifying a sequence of actions if the level is not a line. Indeed, if the designer provides the system a sketch with multiple paths, there is no way to specify in which order the power-ups should be picked up. We could make the designer responsible, by transforming a sketch that not only contains the start and the exit, but also the specific power-ups and their obstacles. However, this would require little more than expanding paths and applying templates. We are interested in using the generator to explore the design space by influencing the lock-and-key mechanism, similar to case study 1. Therefore, we let the designer give input in two modules: one is used to specify the approximate locations of the start, the exit and the power-ups, the second asks the designer for a list that specifies the pickup order of the power-ups. The generator will combine this information to produce an outline with procedural generated locks.

Figure 2 shows a model of the generation process, with examples of how the representation looks during the generation process. The generation process contains the following steps.

1. The first input module asks the user to provide a sketch of the level, this includes the player's spawn location, the exit, the three power-ups and the paths that connect these elements. The second input module asks the user to specify a string with the order in which power-ups should be picked up.

2. The generator creates an outline based on the two inputs. The list of power-ups is used as a recipe, to execute rules in a specific order [5]. These rules transform the sketch and use global variables to mark the generated paths with the availability of the power-ups. This is done by adding a script to those rules, which is executed after the rule is executed. Thus, the execution of one rule triggers one or more other rules to fire. Each power-up rule transforms the power-up symbol into two temporary symbols; this allows each power-up to use the same transformation rules. The player's path is generated between these two symbols, after which both symbols are transformed into either a path or the power-up. Finally, the system marks the power-up as available by changing a global variable.

3. The combination of user input and transformations for winding paths can create ambiguous paths. Ambiguity is very difficult to solve with transformation rules. To prevent undesirable or impossible solutions, the generator offers the designer the option to make changes to the outline.

4. Each tile is split into 7x7 tiles to move from sketch to level layout.

5. The locks are placed in the level at a location where the related power-up is marked as available. The order in which these locks are placed should be the inverse

[2]www.monogame.net

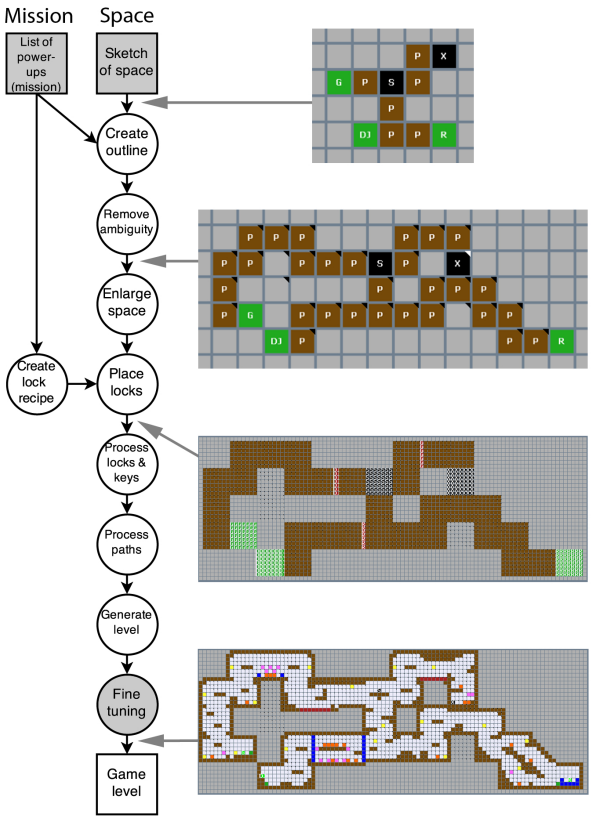

Figure 2: Model of TickTick++ level generation process. Left: The squares represent meaningful products, the circles represent transformation modules. The grey shapes represent the designer's steps. Right: These figures show the data representation during the generation process.

of the order in which the power-ups should be picked up, because the last power-up has the least space in the level. To achieve this, a module that receives input from the designer's list feeds the lock placement module a recipe, a list of instructions similar to the second step, but inversed.

6. After lock placement, the start, the exit and the power-up tiles are transformed into rooms, using templates of 7x7 tiles. The path tiles are transformed into level segments based on templates of various sizes, to take advantage of special shapes in the winding paths. This is where the basic tasks of the player (jumping, fighting/avoiding enemies) are added to the level.

7. The game for which this model was created uses characters that are 2x2 tiles. Therefore, we need to generate a game-specific version of this level. This is done in a separate module by splitting each tile into 2x2 tiles of the same type, and transforming all the non-level geometry back into 1 tile.

8. The resulting level is presented to the designer, who can make final changes.

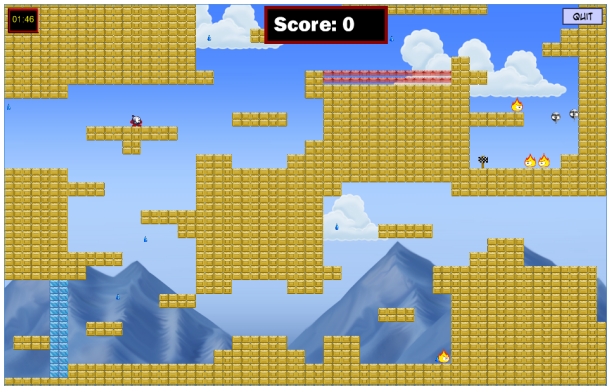

Figure 3: In-game view of a part of the resulting level of Figure 2.

Figure 3 shows a part of the resulting level of the example in figure 2, involving different jump heights, obstacles, pickups and enemies.

5. DISCUSSION

In the previous sections we have described two cases of transforming sketches into game levels. Table 1 shows properties of the two models. We would argue that the cycle of sketching, transforming, fine-tuning, combined with a model-driven approach is a viable way to generate levels in development time. It allows for parameterization, but has multiple input options and re-executable steps as extra options to influence the generation process.

The main advantage of the described model-driven, mixed-initiative approach compared to parameterized grammar-based systems is that the designer can make manual changes at any time in the generation process without changing the content created in previous steps. Moreover, the user does not just give input by tweaking numbers, but by manipulating the representation of the content directly. The generator of case study 1 asks for user input before and after the generation of the quest structure, which gives the user extensive influence on the mission before it is transformed into level space. The generator of case study 2 starts by asking the user for an initial level layout, later asking the user to check the validity of the created outline and again to make final tweaks.

User input modules are not the only input possibilities. The generation process can be executed step by step, allowing the user to re-execute a module if it initially does not give the desired output. Similarly, in the generator of case study 1, the difficulty can easily be adjusted without affecting the abstract action graph by re-executing the process with a different difficulty parameter, starting with the first module after the input module. This will change the selection of monsters as enemies and affect the branching of the puzzles, but not the type or the order of the tasks.

Another quality of the model of case study 1 is the clear separation between mission and space generation. It allows the designer to experiment with the flow of the level without having to consider the details of level geometry. Given that

Table 1: The difference between the models of the case studies.

Case Study	Dwarf Quest	TickTick++
User input	Sketch of mission structure	Sketch of space and list of objectives
Output	Unity	MonoGame
Representation	Graphs	Tiles
Process	Linear	Incoming branches
Mission/Space	Mission first, then space	Space and mission in parallel
Key-lock placement	Based on mission	Based on space
Parameterization	Difficulty level	-
Context	Key-lock mechanisms	Key-lock mechanisms, power-ups

the game objects are there, it is easy to expand the possibilities of the action graph, for example with more elaborate lock-and-key constructions. Also, this clear separation of mission and space allows for the generation of a large number of levels, possibly with varying difficulty, based on the same action graph. One can simply feed the detailed graph to the layout solver multiple times to create different levels.

Though systems with a lot of interacting components, such as these, offer a lot of flexibility in terms of design, an issue that remains is debugging. A bug in the resulting level could originate from the grammar, the model of the design process, or from game code. It might be difficult to separate the actual cause from the effects. So it should be noted that, independent of having a mixed-initiative process, modular generators require extensive testing of the individual components.

An advantage of parameterized algorithms is that they typically communicate with the game engine through data types stored in working memory, allowing elaborate connections between elements through data structures. In Van der Linden et al.'s quest generator, for example, a hierarchy is stored, which allows locks and keys to be connected through their parent nodes [25]. Striving for general compatibility, Ludoscope, however, outputs text files. So, in Ludoscope connections between symbols require a textual representation. As a solution, we have demonstrated the use of adding members to symbols to pass along contextual information.

The generator of case study 2 also uses members, not to connect symbols, but to differentiate between symbols of the same type. Specifically, to mark available power-ups in each path tile, allowing the system to detect where locks can be placed. This could also be used to create more variation in the paths, for example to increase the probability of a room with more enemies if the player has a gun. Members are a desirable solution if the templates are the same in principle, but have some special cases. This reduces the number of rules necessary, thereby optimizing the rule base.

In Spelunky, context is added to symbols in another way. Path tiles with a different number of exits are treated as different symbols, even though they all encode path rooms [10]. In that case, having a different number of exits is not a special case of a path tile, but requires a significantly different template. This is a useful alternative, and could be used in conjunction with the above mentioned members to solve even more complex layout problems.

We have taken two different approaches to the interacting concepts of mission and space while generating a level. In case study 1, the mission is gradually transformed into a spatial representation. Whereas in case study 2, the mission is a list of instructions that guides the transformation of the space. In the case of platformers, the second approach has the advantage that the tile grid gives an easily recognizable shape of the emerging level.

6. CONCLUSIONS

In this paper, we have explored the process of generating content in a mixed-initiative design process. In particular, we have described two examples of how game levels can be generated in a mixed-initiative process with Ludoscope, a tool based on generative grammars. We have described several ways to guide the search space of generative grammars. The design process can be split in multiple re-executable modules, each with their own set of transformation rules. This limits the number of rules that are considered at a certain time step, and is one of the ways to add order to the execution of rules. Instead of deriving all applicable rules from an expression and choosing one to execute, a recipe can tell the system in which order the rules of a module should be executed. Adding members to symbols can reduce the number of considered left-hand sides of a transformation rule. Post-application scripts can be used to let rules trigger the execution of other rules.

The separation of the mission and the space of a level is a useful construct for designing levels at a high level of abstraction, which combines well with model transformations. It separates the reasoning about the flow of a level from the reasoning about how these abstract concepts translate to game assets. However, the creation of templates that convert the concepts to actual game assets still requires quite some manual labor. For example, if the jump height of the player is changed during the development process of a platformer, the current grammar-based generator requires manual validity checking of the templates. Perhaps the creation of these templates could be parameterized to allow automatic adjustments based on the game mechanics.

The translation from the abstract concept, or sketch, to the actual game level benefits from having an extra layer of abstraction, such as the final graph in case study 1 or the outline in case study 2. This layer should be based on encapsulated chunks of information, such as the rooms in a dungeon. In case study 2 we have tried to take advantage of the layout of the environment to place special level chunks. However, this caused many exceptions that required very

specific rules. Even when generating platformers, using the notion of rooms, like Spelunky does, is beneficial, because it takes advantage of the pattern matching capabilities of the computer.

Dormans [5] suggested that separating the mission and space structures would facilitate the reuse of a space for multiple levels. Based on our experience with the two case studies described, we agree, but it can also work the other way around. A mission can be used as seed to create multiple spaces. Designing this way would allow a designer to guarantee a certain flow in the overall game, but still promote replayability by having procedurally generated levels.

We have created a model of the design process by grouping our model transformations into modules, based on their function in the design process. These re-executable modules are an interesting way to influence the design process. The current system doesn't change information of previous transformation steps. However, when a module is re-executed, desired elements of the previous result might still be lost. There should be a way to select what the computer can and cannot transform. Moreover, the designer should be able to re-evaluate this choice every time the module makes transformations.

The two case studies presented both involved interaction between automated generation and human designers, but not with players. An interesting avenue for further research would be to include live adaptation of the game level design process, based on a player's behaviour or performance in the game. Work in this direction has been done by Van Rozen and Dormans [27]. Incorporating this would allow game designers to intermittently playtest and modify their game extensively, but it could also allow personalizing the game based on the player's behaviour [7, 1].

Another direction for further work includes adding methods for visualization of higher-level features of (sequences of) generated game levels that facilitate a human designer to reason about and fine tune the aspects of the generated levels at the level of gameplay. For example, we can imagine system that summarizes rooms in terms of difficulty, based on its content, to allow the designer to see the flow of difficulty of a level or to make suggestions to increase playability.

Our work resonates with work by Smith et al. [19], who argue that the field of procedural content generation would benefit from more insight into the goals, knowledge and tools involved in the process of game design. We have created two level generators in a mixed-initiative design tool, as case studies to demonstrate the use of model transformations in a mixed-initiative design process. We have also discussed the effects of separately generating mission and space on the level generation process, and pointed out directions for future work. In our view, this work contributes to a better understanding of how characteristics of a game influence the design of appropriate levels, in a mixed-initiative process that takes advantage of both algorithmic tools and human designer skills.

7. ACKNOWLEDGMENTS

We would like to thank Dylan Nagel for giving us full access to Dwarf Quest's source code, Roland van der Linden for his guidance in reusing and extending his grammar-based dungeon generator, and Nick Degens for reading and commenting on earlier drafts of this paper.

This research has been carried out in the context of the RAAK research project 'Automated Game Design', which is financially supported by the Stichting Innovatie Associatie (SIA) in The Netherlands.

8. REFERENCES

[1] S. Bakkes, C. T. Tan, and Y. Pisan. Personalised gaming: a motivation and overview of literature. In *Proceedings of the 8th Australasian Conference on Interactive Entertainment: Playing the System*, page 4. ACM, 2012.

[2] M. Cook, S. Colton, and J. Gow. Automating game design in three dimensions. In *Proceedings of the AISB Symposium on AI and Games*, 2014.

[3] M. Cook, S. Colton, and A. Pease. Aesthetic considerations for automated platformer design. In *AIIDE*, 2012.

[4] G. Crammond. The Sentinel. Firebird, 1986.

[5] J. Dormans. Level design as model transformation: A strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, page 2. ACM, 2011.

[6] J. Dormans. *Engineering emergence: Applied theory for game design*. PhD thesis, University of Amsterdam, 2012.

[7] J. Dormans and S. C. J. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games. Special Issue on Procedural Content Generation*, 3(3):216–228, 2011.

[8] J. Dormans and S. Leijnen. Combinatorial and exploratory creativity in procedural content generation. In *Proceedings of the 4th International Workshop on Procedural Content Generation in Games*, 2013.

[9] A. Egges, J. D. Fokker, and M. H. Overmars. *Learning C# by Programming Games*. Springer, 2013.

[10] D. Kazemi. Spelunky generator lessons. http://tinysubversions.com/spelunkyGen.

[11] M. Kerssemakers, J. Tuxen, J. Togelius, and G. N. Yannakakis. A procedural procedural level generator generator. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 335–341. IEEE, 2012.

[12] P. Mawhorter and M. Mateas. Procedural level generation using occupancy-regulated extension. In *2010 IEEE Conference on Computational Intelligence and Games*, Copenhagen, Denmark, 2010. IEEE, IEEE.

[13] MicroProse. Sid Meier's Civilization. MicroProse and Koei, 1991.

[14] Mojang. Minecraft, 2011.

[15] MossMouth. Spelunky, 2013.

[16] B. North. Diablo. Blizzard Entertainment, Ubisoft and Electronic Arts, 1997.

[17] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes. A survey on procedural modeling for virtual worlds. *Computer Graphics Forum*, 33(6):31–50, 2014. doi: 10.1111/cgf.12276.

[18] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2):352–363, April 2011.

[19] G. Smith. Understanding procedural content generation: a design-centric analysis of the role of PCG in games. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 917–926. ACM, 2014.

[20] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):201–215, September 2011.

[21] J. Togelius, N. Shaker, and M. J. Nelson. Introduction. In N. Shaker, J. Togelius, and M. J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[22] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[23] M. Toy, G. Wichman, K. Arnold, and J. Lane. Rogue, 1980.

[24] M. Treanor, B. Blackford, M. Mateas, and I. Bogost. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 11. ACM, 2012.

[25] R. van der Linden, R. Lopes, and R. Bidarra. Designing procedurally generated levels. *Artificial Intelligence in the Game Design Process 2: Papers from the 2013 AIIDE Workshop*, pages 41–47, 2013.

[26] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, mar 2014. doi: 10.1109/TCIAIG.2013.2290371.

[27] R. Van Rozen and J. Dormans. Adapting game mechanics with micromachinations. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, 2014*. ACM, 2014.

[28] Wild Card Games. Dwarf Quest, 2013.