STUDIES IN COMPUTER SCIENCE NO 2, DOCTORAL DISSERTATION

STEVE DAHLSKOG

PATTERNS AND PROCEDURAL CONTENT GENERATION IN DIGITAL GAMES

Automatic Level Generation for Digital Games Using Game Design Patterns

MALMÖ UNIVERSITY

PATTERNS AND PROCEDURAL CONTENT GENERATION IN DIGITAL GAMES

Studies in Computer Science No 2

ISBN 978-91-7104-684-0 (print)
ISBN 978-91-7104-685-7 (pdf)
Holmbergs, Malmö 2016

STEVE DAHLSKOG

PATTERNS AND PROCEDURAL CONTENT GENERATION IN DIGITAL GAMES

Automatic Level Generation for Digital Games Using Game Design Patterns

Department of Computer Science
Faculty of Technology and Society
Malmö University, 2016

The publication is also electronically available at:
http://dspace.mah.se/handle/2043/20371

Dedicated to my family and to the loving memory of Elfride Åkesson,
Stig Åkesson, Majken Nilsson and Karl-Erik Nilsson.

ABSTRACT

The development of content in digital games, such as game worlds, quests, levels, 3D-models, and textures, is costly and time consuming. To address this, different approaches to automate the process of creating game content, often referred to as procedural content generation (PCG), has been suggested. However, PCG is a complex task and include challenges such as creating content with variation, coherent style, speed, and correctness. The research in the thesis is concerned with generating game content with the aid of game design patterns, both by establishing models and exploring different methods to generate actual game content for different games. The methods include implementations of evolutionary computation, i.e. a set of search-based approaches that searches for instances of game design patterns on different abstraction levels that make up Super Mario Bros. (SMB) levels and a learning algorithm implementation based on a model (n-grams) of patterns from the original SMB-game. The different generators were evaluated with metrics concerned with the expressive range of the generators and with user tests.

PUBLICATIONS

This thesis is based on the following papers.

- Patterns and Procedural Content Generation – Revisiting Mario in World 1 Level 1 [48]
- Patterns as Objectives for Level Generation [49]
- Procedural Content Generation Using Patterns as Objectives [51]
- A Multi-level Level Generator [50]
- A Comparative Evaluation of Procedural Level Generators in The Mario AI Framework [101]
- Linear levels through n-grams [52]
- Patterns, Dungeons and Generators [53]
- Player Experience Evaluation of Level Generators in the Mario AI Framework (submitted)

The following paper is related to the research but not included in the thesis

- The Conceptual Relationship Model: Understanding Patterns and Mechanics in Game Design [163]

The only legitimate use of a computer is to play games.

— Eugene Jarvis, creator of *Defender* and *Robotron: 2084*

ACKNOWLEDGEMENTS

I would like to thank my supervisors Paul Davidsson and Julian Togelius for their valuable help and support on this journey. I'm grateful for how you have approached my troubles and questions, often with a more positive outlook than I myself have had.
I'm also grateful for the work my co-authors have put into the different research projects and papers; I have learnt a lot from you. Thank you Gillian, Britton, Noor, Mark and Staffan.
I would also like to mention some of my colleagues that, on a almost weekly basis, have supported this work. Thank you Olle, Carl Magnus and Jeanette. Thanks also to my colleagues at Malmö University, both past and present.

I would also like to thank my children Max, Sam and Esme for letting me fiddle with this project and for bringing me joy and laughter. I love you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AI	Artificial intelligence
DLC	Downloadable content
DPG	Design Patterns in Games
EC	Evolutionary computation
FPS	First person shooter
HCI	Human computer interaction
MMO	Massively multiplayer online game
NPC	Non-playable character
PCG	Procedural content generation
QA	Quality assurance
RPG	Role-playing game
RTS	Real-time strategy
SBPCG	Search-based procedural content generation
SMB.	Super Mario Bros.

Part I

COMPREHENSIVE SUMMARY

1 INTRODUCTION

Digital games is a large industry generating sales around 47 billions dollars Worldwide in 2014 [216]. In the U.S. four out of five households own a device to play games on and houses 155 million players that generates 15 billions dollars in sales on a annual basis [67]. As a market it provides economic growth all over the World, but mostly in the developed countries, particularly Japan, North America and Europe where the large publishing companies and the large consumer markets are located. Digital game development demands a variety of roles of the workforce: software developers, software testers, visual artists, game designers, sound designers, musicians, writers, administrative staff and sometimes even more specialised roles, depending on the size of the developer studio.

However, the game industry is challenged with a set of problems. The high-end games typically takes 18-36 months to develop and the cost can be over 100 million dollars not counting marketing budgets [178, 4]. A contributing factor to the cost and time consumed while developing games is the complexity of the software components being developed, both with respect to the technical and the entertainment aspects. Typically, a digital game could be divided into two major components; the *game engine* and the *content*. The game engine upholds the rules of the game, provides the user with an interface and projects an image visualising the game state. The content of a game could be described as the things that is contained within a game. Examples of individual pieces of game content ranges from game worlds, objects in the game world to interact with, background stories, tasks for the player to perform, to graphics and music.

The continuous advancement of the hardware platform (like gaming PCs, Playstation 4, Xbox One, etc.) and its capabilities (larger memory, faster pro-

cessors and graphics) will increase development cost as well as development time [79, 115]. Similarly, due to the increased complexity of the game engine, every piece of content will become more time consuming to produce [79]. For instance, a 3D-model in a game needs to be modelled with higher detail to look good in higher resolution. Games with large amounts of content are also in themselves a problem in that they will consume large amount of storage space.

Furthermore, digital game projects are sometimes troubled by the fact that it is hard to find enough staff in time to deliver the different components on time, which is both a lifecycle problem and an organisational problem. The competences needed to solve the various tasks varies during the development lifecycle; different skills are needed in different amounts. For instance, during the concept phase only a small crew of about ten persons is needed in contrast to during development and testing where staff could be counted in hundreds. If a game studio is working on multiple titles, the different projects overlap of each other, may cause additional problems, both during upsizing and downsizing of projects and phases. As if it was not enough that project deliverables are more time consuming and costly to produce, sometimes a fair share of project artefacts are scrapped during the development process due to changes in requirements [35]. The objects produced during game development projects are affected in other aspects as well, and rework will be needed of not just software, but also art assets, as well as video, music and audio recordings that have to be changed or created from scratch.

Approaches to automatically create content, often referred to as Procedural Content Generation (PCG) to limit the impact of these problems exist and have done so for some time. A classic example of PCG dating back to 1984, is the space trading game *Elite* [3]. The main advantage of relying on online generation of the game world (galaxies and star systems) [214], instead of designing it by hand was that it allowed the developers to let the game and not the content to take up almost all of the available inter-

nal memory. Moreover, by introducing PCG the developers also made the player able to explore a huge game world. The implemented system was based on Fibonacci sequences together with an initial seed value picked by the two developers. Since most of the information about the game world was automatically generated in real-time during game play, this approach allowed the players to visit eight complete galaxies in the game. But however powerful the solution was, the developers had to manually pick seed values and search through the content in order to prevent profane names of planets and other problematic aspects of the content. The PCG approach used in *Elite*, demonstrates the advantages of the approach, but also its disadvantages. If PCG is to be used efficiently it has to be controlled in a more efficient matter than by letting a human manually check its output. Nevertheless, PCG allow game developers to save resources during production but it also allow them to make different games. With PCG the content can be endless and adaptive, thus giving the player a never-ending game universe to explore and content that is adjusted to the player's way of playing. PCG allow developers and designers to explore the possibilities of the game and similarly it allows researchers to understand the game design process.

The research presented in this thesis focuses on automatic generation of digital game content using *design patterns* to guide the process to specific solutions. Design patterns is a design method were a pattern describes a solution to a recurring problem in such way that it is possible to reuse in different situations. Design patterns are used together with different algorithms to generate game content. The goal of this approach is similar to other PCG methods, namely; to reduce cost, to contribute to solve staffing problems, and to limit rework of digital game content but still fit into the game development lifecycle. Further, the approach is also suitable for the game design process and attend to content-related issues and not only focus on the automatic generation activity. The approach proposed in the thesis have been applied to area of level generation, especially for 2D representation but also for some 3D representation of game space. In particular, the approach of searching for patterns in game content on different levels of ab-

straction where smaller patterns on a lower level makes up more complex patterns on higher levels, could be used to generate content, not just for games, but for any virtual environment including training scenarios and simulators.

2 BACKGROUND

This chapter presents the context for the thesis; *digital games*, as well as the central knowledge area for the research, namely; *Procedural Content Generation*. From the knowledge area the chapter moves onto the application domains and the related research areas. The chapter will conclude with concepts that are essential for the thesis: *Evolutionary Computation*, *n-grams* and *Design Patterns*.

2.1 DIGITAL GAMES AND THEIR INDUSTRY

Digital games[1] have over the last 50 years (see Fig. 1ff) evolved from simple space shooters (*Spacewar!* [217]) and sport games (*PONG*[2] [16]) to large open world games[3] and massive multiplayer online games (MMO[4]). During this time, digital games have moved from obscurity to a more mainstream position within the entertainment industry. In a recent survey the US Entertainment Software Association [66] found that 51% of U.S. households own a dedicated game console and that 58% of the U.S. citizens play video games. Similarly, 59% of 6-65 year olds in the U.K. "are gamers" according to a BBC-report [173].

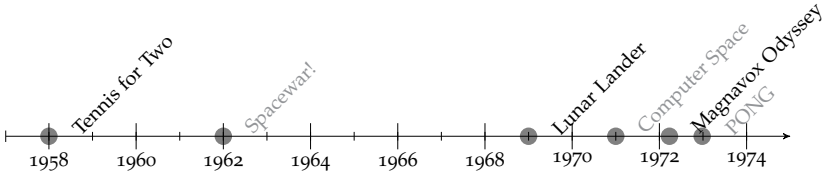


Figure 1: Games timeline 1958-1974

50 years is a rather long period of time, at least in a consumer and/or production perspective, especially if you consider the technical development of digital games.

As a way to understand this period of time and the challenges the game industry are faced with, we will use the notion of (technical) platform[5] generations. A generation typically stretch for 5-7 years, and members (products or platforms) of a generation share functionality and traits. A platform from a previous generation will in some aspects seem dated in relation to the current. At the time of writing, the current generation is the eighth generation[6] and consists of the game consoles *Xbox One*, *Playstation 4* and *Wii U*. A (console) generation can be identified as a *de facto* standard where the following determinants are important; technological innovations, switching costs, installed base and complementary products [80].

Another centrepiece of the digital game hardware advancement is the advancement in "technological innovations based on video graphics capability" [80]. Other technological enhancements of the platform typically means more memory (primary and secondary storage), more storage space in the off-line read only storage (cartridges, optical discs, etc.) and more computational power. However, this development mainly affects the player experience in a positive way. On the developer's side of things, the need for specialised knowledge occurs. After a new console (generation) has been introduced, we can observe the following trends;

- new hardware investment (consumer & developer) [115, p. 57]
- development teams increase in size (developer) [78, p. 366]

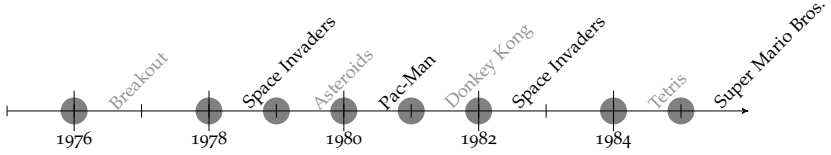


Figure 2: Games timeline 1975-1985

- development time for an average project increase (developer) [78, p. 366]
- increased expectations (consumer) [78]
- average project cost increase (publisher and developer) [78, p.423]
- increased platform complexity (developer) [115]

Other changes are of a more temporarily nature like the steep learning curve developers face when switching to another platform [115, p. 57].

From a certain viewpoint it might seem like a strange effect; how can the change of the platform in a positive way affect the development of a game in a (production-wise) negative way? What drives the higher cost, longer development time and forces the need of larger team size (and thus more complex to manage)? More advanced video graphics forces the need of 3D-graphics models and textures with higher details, which in turn takes longer time to model for the visual artist[7]. The larger available space on storage devices allows larger game worlds, more non-player characters, more buildings, longer levels, more story, more items, more enemies, more quests, etc., which in turn takes longer time to design and develop.

How large is the effect of this change? During the previous half century, for every new game console generation, the project scope and size is about twice of the previous generation's scope and size [78] (see Fig. 5). For instance, the MMO *World of Warcraft* [33], contains more than 12000 quests. The open world action-adventure game *Just Cause 2* [19] which is in a trop-

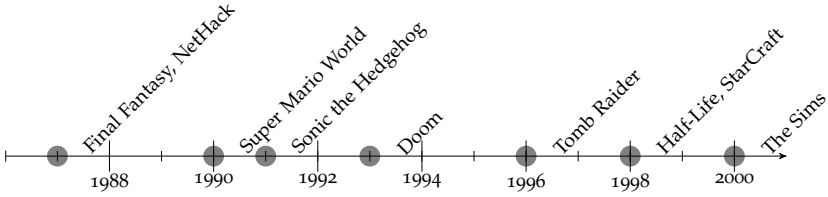


Figure 3: Games timeline 1986-2000

ical island setting has a virtual world with a size of over 1000 km^2, that needs to be filled with interesting things to do and interact with.

If this trend of doubled cost, doubled team size and longer development time continues, it will force continuous change on the game industry and its consumers, either by forcing the developers to change their way of working, changing the product or changing the consumers expectations on the product. One way of prevailing the current state of the game industry and market is to increase the use of tools that support to the development, preferably without causing to much change in other areas. Another possible outcome is that the market only is able to sustain a smaller set of companies, thus limiting the amount of games produced and with this, limiting the artistic expression and diversity. On the other hand, independent[8] game developers and mobile games have over the last years been able to counter this trend.

From a software architecture standpoint the approach of utilising tools is already supported in the game industry, since most digital game development use *Separations of Concern* [56, 176] to facilitate development. A common approach is to organise the source code structure or the software architecture into a "game engine" (software components) separated from the art assets (data) [93]. In essence, the game engine could be viewed as a software framework with the sole purpose of supporting the creation and development of digital games. Core functionality of a game engine is typically; a render engine (2D and/or 3D graphics), animation, a physics engine (handling collision detection and response), memory management and sound. If needed other tasks could be handled by the game engine cover; artificial

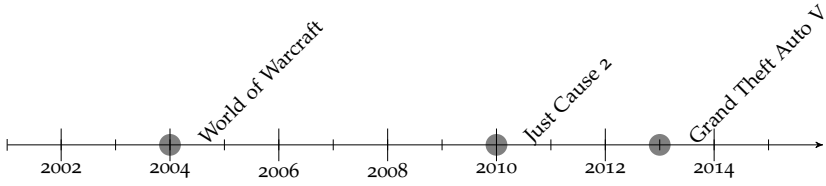


Figure 4: Games timeline 2001-2015

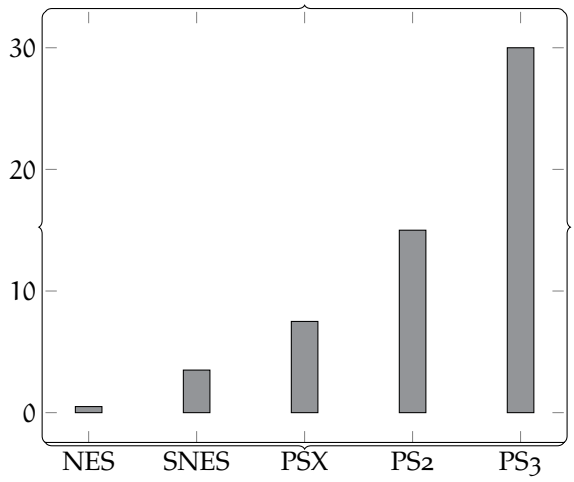


Figure 5: Average project cost (in millions of $) per console type (from [78, p. 423]).

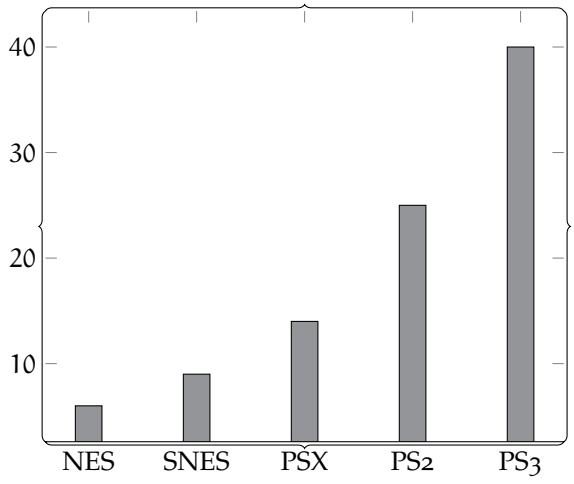


Figure 6: Average console development team size (from [78]).

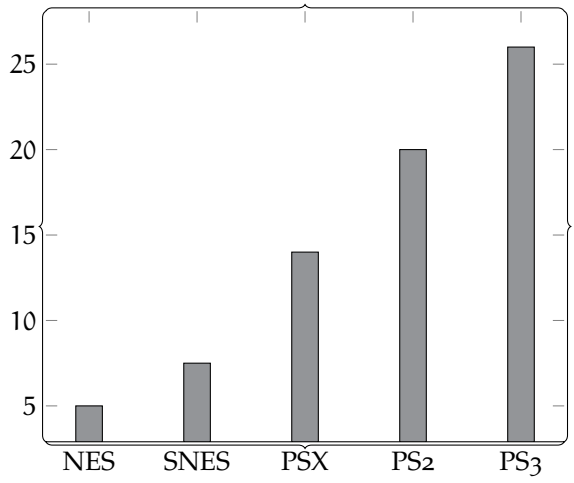


Figure 7: Average development time in months by platform (from [78, p. 366]).

intelligence, scripting and networking. Advantages of using a game engine could be software reuse and parallel development. Major reuse gain include actual code reuse by using the code and components in different games or supporting multiple target platforms and localisation markets. The modification of art assets from source data files to runtime use in digital games, is usually referred to as the content pipeline. Software tools are used by art assets creators (level designers, 2D- and 3D-artists, etc.) but software components can be used to support automatic (co-)creation and reuse of those assets. One suggested method is to apply a (semi)-automatic process called *Procedural Content Generation* (PCG).

2.2 PROCEDURAL CONTENT GENERATION

This section of the thesis begins by explaining the name Procedural Content Generation. It will then go on to give a set of examples of content in more detail, and exemplify how PCG has been used in the game industry together with some related examples from the research arena. Finally, before

transitioning into the next section, it will review the literature concerning PCG and point to research challenges in the area.

As previously stated (see chapter 1), a digital game can be divided into two components; the *game engine* and the *content*. Procedural Content Generation has over the last 10 years been the topic of numerous research papers and it has been used in commercial as well as small independent game productions even longer. However, the name Procedural Content Generation is a bit unclear and has been, argued to be an unsuitable name [43], especially when considering other research areas which also use algorithms to generate artefacts, like procedurally generated kitchen designs [76]. However, "procedural" in PCG was coined as a term in Computer Graphics where *procedural* denotes a computational process-derived artefact [179, 43].

Notable definitions describe PCG as "creating game content automatically, through algorithmic means" [228] or that PCG "is the application of computers to generate game content, distinguish interesting instances among the ones generated, and select entertaining instances on behalf of the players" [96].

2.2.1 *Game Content*

Game content could be expressed as the things that is contained within a game, like the game world to move around in and where event takes place, game objects to interact with, game tokens to move (units), a background story for the player or the game world, tasks and assignments for the players to complete, but also graphics with variation, sound and music based on events in the game. The following examples are chosen because of their relevance to the work presented in this thesis.

QUEST is a task a player character or a group of player characters have to complete to gain a reward. The reward can be in-game items, supporting Non-Player Characters (NPCs), new skills, access to new areas or in-game currency. Quests can be linked together in a sequence or a chain to advance a plot or story the game has. Some quests or quest

chains may have prerequisites that must be fulfilled before the player can start any or some of the quests. Usually the game has a set of quests, so called side-quests, tied to the completion of the game and a set of quests that are not required to complete to win the game. Quests are usually part of role-playing games (RPGs) or similar types of games. An RPG is a game in which the player assume a role of a character. A central part of the game is character development both in relation to the narrative and the skills the character has. Quests in other types of games than RPGs are sometimes called missions (cf. *Grand Theft Auto V* [178]).

DUNGEON A single level or a set of levels in an RPG that is set in a underground complex, castle, cave or ruin. Sometimes several dungeons are connected via an overworld that acts like a hub, often with cities and a wilderness were other types of adventures or quests takes place. The cities may function as a replenishment zone where the player can trade found objects or treasures for better equipment, weapons, magic spells or healing potions. A dungeon often contains both enemies, hidden passages, locked doors and puzzles. A common theme is to let the dungeon be a labyrinth or at least have a non-trivial layout allowing the player to explore unknown game space. The overworld often has the function of being the world where the game takes place and is sometimes referred to as world map, especially when the scale is different from the scale in the dungeon.

LEVEL Most games are divided into sections of game space with a discrete change in difficulty [143]. Levels may represent locations or have objectives for the player to fulfil. In some games like *Super Mario Bros.* [158] the objective is simply to move the avatar from the starting point to the end of the level. When the objective is completed the player usually continues to the next level, perhaps via a certain place in the level. If the player fails the level, the player usually have to start from the beginning of the level or at certain predefined points (check points). The concept of levels is realised differently in different types of games. In RPGs, Real-time Strategy games (RTS) and multiplayer games a level

is often referred to as a map. An RTS game is a kind of strategy game where the game progress in realtime and not in turns. The player manoeuvre units to control areas of the game space and to destroy the opponents units and resources. RTS games often include base building, resource gathering, technological advancement and building new units. Other common names for levels are; area, zone and stage. Sometimes levels are grouped together by the same gameplay theme with the same type of enemies or obstacles. These grouped levels are often called a world. For older games the dividing of the game space was a necessity due to memory constraints. In classic arcade games like *Donkey Kong* [157] the levels only consisted of a single screen, often called a board.

Influential games with PCG are *Rogue*, *Elite*, *Civilization*, and *Diablo*. They have all spawned sequels and in some cases spin-offs and boardgames. There are however, several prominent examples of different usage of PCG in independent and commercial games (see Fig. 8 and 9):

AGE OF WONDERS: SHADOW MAGIC is a turn-based strategy game with a random map generator that allows the player to play new stand-alone scenarios [233].

ALIEN SWARM is a top-down shooter that incorporates two PCG components; the *AI Director* that dynamically generates swarms of opponents based on a set of parameters and the *TileGen* which allow the

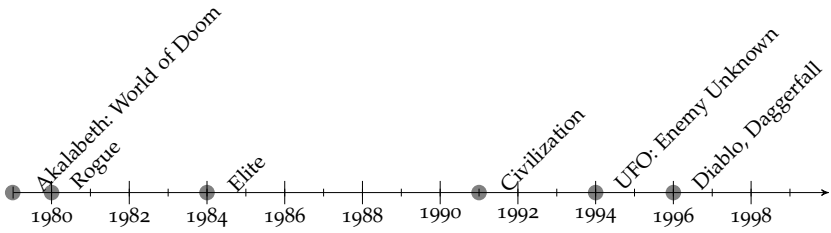


Figure 8: PCG Games timeline 1979-1999

user to build a set of rooms and define rules on how these rooms can be connected in order to generate a level [238].

BORDERLANDS I and II utilise the *Gearbot* to create most of the randomised objects (weapons, enemies, armour stats and some of the orientation of stairs and cover) in the game [90, 91].

CIVILIZATION is a turn-based macro-scale strategy game with a world map generator [147]. Civilization is an influential game franchise that has inspired other games but also resulted in some spin-off games like *Sid Meier's Colonization* [148] and *Sid Meier's Alpha Centauri* [72] both having world map generation.

DAGGERFALL from Bethesda used pseudo-randomisation for several different types of content in this RPG game, both before shipment and during runtime. The randomised content consists of dungeons, NPCs, rumours told by NPCs, layout of cities, magical items, etc. [27].

DIABLO by Blizzard procedurally generates different dungeon levels for every play-through which includes items and monsters (placement, type and amount) [34].

DWARF FORTRESS is a management game where the player controls a settlement of dwarves [6]. During the game world generation phase the player can adjust a set of parameters like size and length of history. A midpoint displacement algorithm works as world generator and it includes elevation, rainfall and mineral distribution. Areas are divided into biomes which decides its population. Erosion and rivers are added to the world and the history of world is generated including how civilisations, races and populations spread over the world.

ELITE [3] uses Fibonacci sequences [214] and a look-up table, allowing the player the possibility to visit 2^{48} galaxies. Due to concern for the users, this was scaled down to only eight galaxies with 256 planets to explore. However powerful the solution is, the developers had to manually search through the content in order to prevent profane names

of planets and other problematic, but more game mechanic oriented, aspects of the content.

FAR CRY 2 Ubisoft Montreal worked together with World Machine [242] to generate the terrain and solved the generation of enemies and events with PCG [236].

.KKRIEGER was a First-person shooter game (FPS) developed by a German demogroup called *.theprodukkt* at a 96 kB game competition. Due to the limitation of storage space, procedural methods are used to handle textures and 3D-models [69]. An FPS is a projectile weapon-based game in a first-person perspective for the player.

MINECRAFT is an open world game in a procedurally generated world divided into biomes including deserts and jungles. The world is generated as the player moves in it and the generation is based on a single seed that initiates three 2D Perlin noise [171] heightmaps (overall elevation, terrain roughness, and local detail) that set the shape of the world [150].

LEFT 4 DEAD I and II are both cooperative survival horror FPS games set in a urban environment firmly haunted by a zombie outbreak. The game uses an *AI-director* that orchestrates dynamic spawn points for both enemies and items. Similarly, the *AI-director* varies the music to create a dramatic setting, as well as modifies layout, textures, objects and animations for greater variation [235].

ROGUE is a survival dungeon crawl RPG that takes place in procedurally generated dungeons [232]. The game was highly influential and emanated in several spin-off games, which resulted in a new genre games called *rogue-like* which usually have the following characteristics: procedurally generated levels, turn-based gameplay and *permadeath* (or permanent death) of the player-character.

SPEEDTREE is a commercial middleware system for vegetation generation which has been used in over 1,000 games [105]. The developers of

SpeedTree: *Interactive Data Visualization* have developed a special version to be used for film productions as well.

SPORE is a god game where the player takes control over a creature and makes choices for the creature and potentially its civilisation [140]. The game uses procedural content to handle how creatures moves since the player's different choices affect how the creature looks.

UFO: ENEMY UNKNOWN is a science fiction strategy game that uses a generator to create UFO crash sites or UFO landings [153]. The generator places different prefabricated terrain sections from a library covering different environments (urban, desert, temperate climate or farmland) to match the landing site where the player lands.

In games where the game world is procedurally generated the players are encouraged to explore and in games where the levels are procedurally generated the players are encouraged to play the game again. Moss [152] argues that games like *No Man's Sky* [83] with their vast content has "a singular purpose: to make the player feel lonely". As previously mentioned, PCG may be used to save time and resources but PCG can also allow new types of games likes Dwarf Fortress and Minecraft which generates huge amounts of content for the user to interact with, but also provide aesthetic values for the player when observing the world. Both Dwarf Fortress and Minecraft are unique games and would not been done in their special way without the aid of PCG.

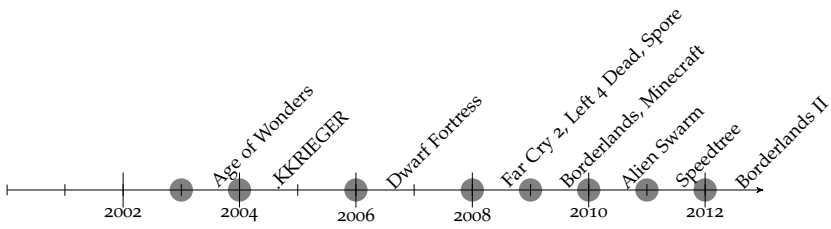


Figure 9: PCG Games timeline 2000-2012

Correspondingly, researchers in academia have explored different usage of PCG for other uses. As well as in the game industry the examples are expressing different usage. Due to the amount of studies performed, only a handful are mentioned here to show the variety in the research.

SPEEDROCK by Dart et al. [54] is a lightweight tool using a 3D L-system to generate rocks for games and virtual worlds not unlike the commercial PCG middleware SpeedTree for vegetation generation.

STARCRAFT MAPS GENERATION in this approach a multi-objective evolutionary search [225, 224] is used to generate maps for the RTS game *StarCraft* [32].

QUEST GENERATION Doran and Parberry [60] constructed a quest generator after a structural analysis of MMORPGs. *Skyrim* [26], the sequel to Daggerfall, has a special type of generated quests called *Radiant Quests*.

2.2.2 *PCG Research*

Togelius et al. [228] suggested a taxonomy for PCG, which denotes different dimensions of PCG approaches. The five different dimensions are as follows; 1) *online* versus *offline*, 2) *necessary* versus *optional* content, 3) *random seeds* versus *parameter vectors*, 4) *stochastic* versus *deterministic* generation, and finally 5) *constructive* versus *generate-and-test*.

In the case of *online* versus *offline*, it should be noted that, this covers the aspect of *when* a PCG system generates content, during the running of the actual game (online) and before the game is shipped (offline). The *necessary* versus *optional* dimension express the need for *correctness* of the content and how important the content is to the player experience. The necessary content must be correct due to its function in the game, if it is not correct the player will not be able to continue through the game. Regarding the *random seeds* versus *parameter vectors* dimension, it illustrates, whether the algorithm uses a single random number or if it utilises a multidimensional vector to generate content. As for the *stochastic* versus *deterministic*

dimension it demonstrates whether the output of the algorithm is different or the same every time with the same parameter setup. Concerning the last dimension, *constructive* versus *generate-and-test* it describes if the algorithm has an internal control function evaluating the generated content. In this aspect, the *constructive* can generate content in one step and it is certain that it is good enough whereas the *generate-and-test* continuously generate and evaluate until the content is good enough.

Togelius et al. [228] argue that search-based PCG (SBPCG) is a special case of the *generate-and-test* approach since it does not just accept or reject content, but it grades the content with the aid of a *fitness function* and produce new content with higher fitness from prior candidates. SBPCG is an approach applying stochastic search and optimisation algorithms to generate content. The taxonomy was initially suggested in a prior paper [226] but was more in depth explained and contained a survey of SBPCG research [228].

These five dimensions have been extended with two more; *generic* versus *adaptive* and *automatic generation*[9] versus *mixed authorship* [188, 231]. The first extension covers the aspect of whether the content is generated for all, or if the content is generated for every individual player.

An example of an adaptive approach in PCG was made by Shaker et al. [189] which used a model based on an empirical data set, a mechanism that adapts level design parameters to each player's playing style and an evaluation with both algorithmic and human players. The domain used is the Super Mario Bros. the clone *Infinite Mario Bros* [172].

The other extension demonstrates possible ways for humans to interact with the generating process. Traditionally human input to guide the generation process was focused to the design of the PCG system, but with the *mixed authorship* the designer or player can co-author the content and interaction may happen on multiple points in time. During the last couple of years different variants of sketching-like interface and turn-taking PCG tools [197, 245, 129, 130, 132, 174, 128]. Smith et al. [203, 208] analysed platform game levels and constructed the *Tanagra*, a mixed-initiative

level generator. *Tanagra* allows the designer to interact with the PCG tool to create levels by letting the designer draw parts of a level which is then complemented by a constraint solver algorithm that generate content that fills in the gaps. The geometry is generated with the aid of patterns on two different levels; *single beat* [204] and a slightly higher level which acts as composite patterns of the previously mentioned *single beat* patterns. Related to the human interaction with the generating process is the problem of maintain an understanding of the content generated. The problem is inherent to PCG due to the vast amount of content a generator can create. Smith and Whitehead [202] suggested the concept of *Expressivity* for level generators to analyse the space of levels that can be created.

An example of PCG without the mixed authorship approach dimension of having a designer interacting with the tool is the Occupancy-Regulated Extension (ORE) generator. Instead, it was initially designed to capture the aspect of human creativity to produce interesting level designs by merging humanly pieces of pre-made sections to form new levels [139]. It was submitted to the Level Generation Track of the IEEE CIS-sponsored 2010 Mario AI Championship [190] a competition where generators competed in IMB levels.

Conversely, the quality of a specific game content is intimately tied to the game and its context. A perceived high-quality content in one game (or game genre) might be considered low quality in another game (or game genre). Aspects of game content could further be discussed in how varied (high variation or low variation) or how repetitive (often repeated or unique) it is.

Smith [201], analyses the role of PCG in games from a game design perspective with the aid of the MDA-framework (Mechanics, Dynamics and Aesthetics). The MDA-framework [103] is a formal approach to understand games through three connected aspects: the mechanics (connected to the rules of the game), the dynamics (how the rules functions in action with other rules) and the aesthetics (which covers the player's experience of the

game). Smith [201] argues for several "nuances" that unpack the common replayability argument of PCG.

Among these three dynamics motivate different kinds of replayability than previously demonstrated in the research discourse; 1) reacting in a surprising environment, 2) building generator strategies, and 3) practising in different environments. The first one leads to replayability in a manor of playing new content on each occasion while the second leads to replayability due to building new experiences around the different abilities of the generator. Lastly the third facilitate replayability by using know mechanics in new scenarios.

Another design-centric approach to understanding PCG is Khaled et al. [117] who discussed PCG as design metaphors to bridge the gap between PCG research in game AI and more design centric areas as human computer interaction (HCI). The design metaphors for PCG in games are the following: *tool*, *material*, *designer*, and *expert* [117]. Contemplating the different roles PCG can take, might extend the use of PCG since most research in PCG falls under the label *tool*.

PCG as a *tool* means that the approach explains PCG as a device or instrument that is manipulated to fulfil a specific goal like changing the environment or extending the user's abilities. However, given the use of a PCG system the different research artefacts could fulfil more than the role of the *tool*.

*Material*s are dynamic and reconfigurable substances that can be modified by the designer. In effect, this means that both the output of a PCG system and the PCG system itself could be viewed as *material* when the designer can manipulate them, for instances by changing parameters[10] or by selecting specific areas and letting *Speedtree* [105] generate vegetation for that part of the game world or the generator that generates generators by Kerssemakers et al. [116].

PCG as a *designer* denotes a role where the PCG system takes on a design task in contrast to the two previous roles, where a designer must be present. A PCG system working as a *designer* solves both design as well as meta-

design activities. Examples of where a PCG system solves tasks as *designer* include both generating games [221, 38, 37, 44] as well as caricatures of games [198] in order to understand how design works [117].

The final role, *expert*, covers two different types of experts; player and domain *expert*. The first *expert* demonstrates analysis, interpretation and adaptation suggestions related to player experience. The second *expert*, provides analysis and interpretation from specific domain knowledge. The domain knowledge could be used by a *designer* to change the design artefact. Examples of player *expert*s [222, 168, 169] and domain *expert*s [10] may look similar but when the first provides input on experiences as fun, challenge and frustration, the second one provides knowledge on how to generate levels that induce the desired experience for the player.

Togelius et al. [229] suggested a set of long-term goals and research challenges for PCG. The long-term goals are *Multi-level Multi-content PCG*, *PCG-based Game Design* and *Generating Complete Games*.

For *Multi-level Multi-content PCG* Togelius et al. [229] envisioned as a system that can generate multiple types of quality content at multiple levels of granularity in a coherent fashion. *PCG-based Game Design* on the other hand, is a game or a whole game genre in which PCG is the central mechanic without which the game could not exist at all. For the goal of *Generating Complete Games* to be completed, PCG would have to generate a whole game from scratch, including the rules and game engine.

From these long-term goals it was argued that work addressing any of the accompanying nine more concrete research challenges would contribute to progress towards fulfilling the long-term goals of PCG. Amongst these, two research challenges are related to the research in the thesis; *Representing Style* and *General Content Generators*, together with one of the actionable items, *Competent Mario Levels*. *Competent Mario Levels* refers to creating generators with the ability to generate varied, interesting playable, entertaining and good-looking levels. In PCG, representing style, refers to activity of defining a generative model that follows a particular designer's recognised style or a particular school of design thinking. Examples of physical world

of design styles are for instance, *Art Deco* and its predecessor *Art Nouveau* which are clearly separable. Arguably the same would be true for game content (*cf. Super Mario Bros.* vs. *Sonic the Hedgehog* [158, 212]).

2.3 APPLICATION DOMAINS

Most of research in this thesis is centred on content for the classic 2D platform Super Mario game series with total sales over 290 million copies since the release of the first game in 1985 [158]. In addition to its influential design and great impact on the market it has also been used frequently in research in different forms [116, 186, 168, 169, 227, 187, 189], but mostly tied to the Mario AI Benchmark (*cf.* [223, 190, 113]) which, in turn, is based on the clone *Infinite Mario Bros* [172].

The game *Super Mario Bros.* [158] is divided into eight "worlds" that contains four levels each. Every world contains three levels that stretch from 148–377 tiles long and 14 tiles high, that functions as part of the *Mushroom Kingdom* and one level that functions as a "Boss" fight level in a castle where the hero Mario is faced with a different opponent from the other levels; i. e. *Bowser* (or one of his impostors placed to guard the initial seven castles).

The player guides Mario from a position on the left through levels that scrolls to the right. In a level Mario can walk, jump and run. Mario's opposition consists of moving enemies, gaps to jump over and fixed blocks that sometimes breaks or contain coins or power-ups. The power-ups grow Mario to the double size, grant him an extra life, or let him shoot fireballs.

Another popular game genre is the fantasy Roleplaying Game (RPG) style of games. Besides procedurally generated quests [59] and the combined generation of game space and quests [62, 61] the game space artefact (dungeons) have been the interest of researchers [239, 192, 154, 12, 13, 142, 237, 14, 63, 109, 94, 199]. This is perhaps due to the fact that procedurally generated dungeons have been present in popular games since the 1970s and onwards *cf.* [84, 232, 34].

2.4 RELATED RESEARCH FIELDS

The conducted research in this thesis have bearings on a sectional set of disciplines, namely game design, computational intelligence and computational creativity but also computer graphics (visualisation and movie production) and for specific uses as in scenario generation for scenario-based training (cf. [136, 137]) and for generation of robot control software safety tests cases [11].

2.5 CONCEPTS

This section of the thesis presents the algorithms and techniques used in the thesis. This section begins by explaining Evolutionary Computation, It will then go on to n-grams and end with Design Patterns.

2.5.1 *Evolutionary Computation*

Evolutionary computation (EC) is a subfield of Artificial Intelligence (AI) inspired by the Darwinian principles and the mechanics of evolution. Generally speaking, EC methods belongs in the family of problem solvers called trial-and-error [65]. As such they use a population of solution candidates in a stochastic search to reach a (global optimum) solution for a problem.

EC methodologies are often used for complex optimisation problems, like the timetabling of university classes [165, 39], and they have been shown to successfully and efficiently find solutions in situations with large search spaces with multimodal problems [65, 146, 125]. EC is of course not really the exact same as natural evolution and therefore the terminology differs. The EC problem solving metaphor in relation to natural evolution terminology is shown in table 1. For instance, the fitness of an individual, is, in essence, how high the quality of the solution is.

Evolutionary approaches in computation was proposed as early as 1948 by Turing [234] and followed by Bremermann [36] who in 1962 was able to demonstrate experiments on optimisation with the aid of "evolution

Table 1: The basic evolutionary computing metaphor linking natural evolution to problem solving [65]

Evolution		Problem solving
Environment	$\longleftrightarrow$	Problem
Individual	$\longleftrightarrow$	Candidate solution
Fitness	$\longleftrightarrow$	Quality

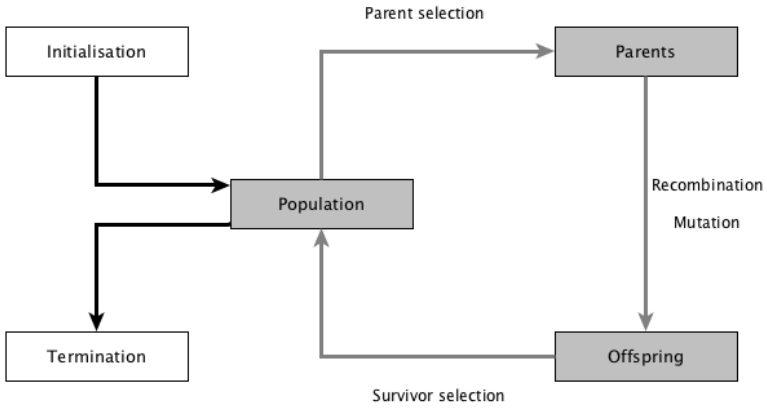


Figure 10: The general scheme of an evolutionary algorithm as a flowchart [65].

and recombination". EC was further developed in the 1960s and the 1970s with three different inovations; evolutionary programming [77], genetic algorithm [99, 100] and evolution strategies [177, 183]. In the 1990s this was followed by genetic programming [21, 120, 121].

Due to the four main directions of EC; evolutionary programming, genetic algorithms, evolution strategies and genetic programming, there are several different variations on EC but they share the following components; a *population* of individuals in some *environment* with limited resources and the *competition* of these resources cause a situation of natural selection. Since EC is purely artificial, the competition of resources is not real but rather the

effect of ranking of the individuals and the *selection* of individuals. The combination of these operators cause an improvement of the fitness in the population. With the aid of a fitness function (quality function, objective function or evaluation function), it is possible to evaluate the individuals and rank them and send some of these individuals to the next generation.

Listing 1: The general scheme of an evolutionary algorithm in pseudocode [65]

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

By applying recombination and/or mutation to the population, hopefully, better candidates are created. Recombination is an operator applied to two or more selected candidates (parents), producing one or more new candidates (children) [65].

Mutation is another operator applied to a candidate which result in a new candidate. Creating offspring and ranking them together with the ancestors with the fitness function, in order to keep the best over a set of generations will generate a suitable solution (a member with high enough quality or the specified number of generations have passed). Eiben and Smith [65] stress that two main forces forms the basis of evolutionary systems, namely; variation operators (creates the necessary diversity in the population) and selection (the force that increases the mean fitness of the members in the population).

For the research done in this thesis; genetic algorithms are of particular interest since several of the artefacts are based on the principal scheme of listing 1 and is illustrated in figure 10.

2.5.1.1 *Components and the principle of Evolutionary Algorithms*

Eiben and Smith [65] explains the main components of an evolutionary algorithm:

REPRESENTATION: In order to be able to solve the problem (or at least to get a solution candidate) a translation and codification of a solution in the problem domain (called a *phenotype*) must be made to the EA (called a *genotype*). Representations are central for EC and are selected to suit the problem at hand. A binary string or a string of symbols may suit some problems. Other problems may need an array of real numbers or a graph representation. For example, if the classic computational TRAVELLING SALESMAN PROBLEM (TSP) was to be solved by EC, it would perhaps be suitable to have the representation as a string of symbols to represent the cities and the order of the sequence is how it should be traversed. Suppose there are five cities to visit; CITIES = {A, B, C, D and E}. A possible solution sequence for the five cities could then be CITIES = {A, E, D, B and C} and the first city to visit is A, then E, and then D, etc.

FITNESS FUNCTION: Its role is to serve as an evaluation of, as well as represent, the requirements the population should comply to. As such, it sets the basis for the actual selection of individuals in the population by assigning a quality measure to genotypes. For the TSP the fitness function would be the distance travelled to complete the sequence in the possible solution CITIES = {A, E, D, B and C}.

POPULATION: The population holds a set of possible solutions. It could be viewed as a container for a defined number of individuals. In evolutionary computational terms, it is not the individuals that change but instead remain static and thus it is the population that improves.

PARENT SELECTION: In essence, this is the strategy that differentiate by all of the individuals in order to decide which ones of the individuals that become the parents of the next generation. Overall, in EC, the parent selection is usually based on picking individuals with higher quality

to create offspring. Sometimes, certain approaches allow individuals with low quality to continue on to the next generation, by giving them a small chance with the sole purpose to counter too greedy strategies which may get stuck at a local optimum.

VARIATION OPERATORS: The purpose of variation operators is to create new individuals based on old individuals. The operators are either unary (mutation) or n-ary (recombination). The mutation operator is applied to an individual to create a small random and unbiased change. The mutation operator is usually different in the different directions of EC. Solving the TSP we could pick subsequences from one parent and then add the other cities from the other parent.

In the genetic algorithm direction, a common mutation operator for binary encodings, regard each bit and allows every bit to change according to bitwise NOT with a small probability (see figure 12 for an example where the third, fourth and eighth positions are selected for mutation). The recombination operator is often called a crossover operation, which is perhaps more literal to what the operator does, namely crossing two parent genotypes into one or two offspring.

Both the mutation and recombination operator are stochastic. In the case of the recombination operator the choices of what parts of the parents and how this is done is dependent on random drawings [65]. As with the mutation operator, the recombination operator is different in the different EC directions and is typically seen as the main search operator in genetic algorithms but it is never used in evolutionary programming [65]. In figure 11 the crossover happens at a single point (after the fourth bit position).

SURVIVOR SELECTION: In most cases in EC the population size remains constant and therefore a selection strategy has to be applied to distinguish among the individuals based on their quality (fitness value). In some approaches the concept of age is also applied giving the selection strategy a focus on offspring and not only selecting survivors based on fitness value.

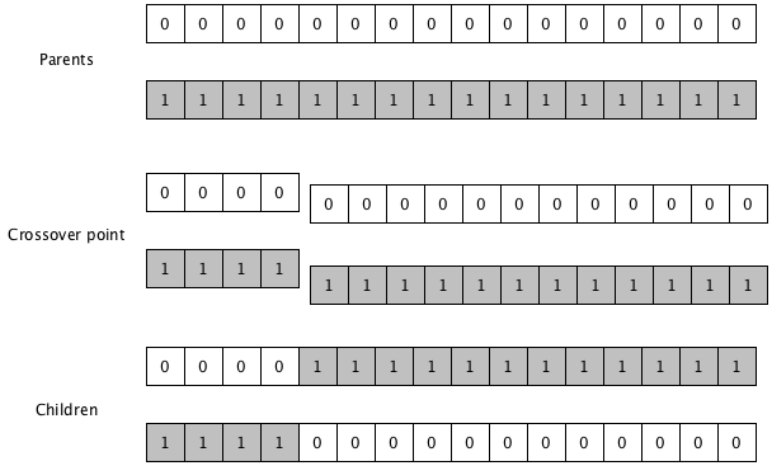


Figure 11: An example of a crossover operation.

2.5.2 n-*grams*

In 1948, Shannon published his seminal work "A Mathematical Theory of Communication" [194] which focuses on the problem of how to best encode information with the purpose of transmitting it. In this work Shannon uses probability theory as a method for measuring the uncertainty of a message. While simultaneously inventing information theory, Shannon [194], also uses n-gram-based approach to create a model of a natural language.

Later, Shannon [195], also used n-grams to statistically demonstrate the average information carried by each letter of the English alphabet and to show the upper and lower bounds of information entropy of English, which allows for a statistical base for language analysis with use in data commu-

Figure 12: An example of a mutation operation [65]

Table 2: 3-gram probability values for an n-gram predictor [149]

	..R	..L
LL	$\frac{1}{2}$	$\frac{1}{2}$
LR	$\frac{3}{5}$	$\frac{2}{5}$
RL	$\frac{3}{4}$	$\frac{1}{4}$
RR	$\frac{0}{2}$	$\frac{2}{2}$

nication and encryption.

A model of the probability distribution of n-letter (or words, etc.) sequences is called an n-gram model. Essentially, an n-gram model is defined as a *Markov chain* of order $n-1$ [180] where the probability of a symbol depends only on the previous symbol. In principal, the modelling is based the traversal of a known body of text (a corpus) where counting symbol sequences gives the probability of their presence in the corpus. In the sequence (corpus) "LRRLRLLLRRLRLRR" where the symbol "L" means left and "R" stands for right, a 3-gram model gives the probabilities for the next symbol to be an "L" or an "R" as shown in table 2 [149].

An application for n-gram models are: action prediction, language analysis [149] and language identification [180]. Applying probabilities to identify the next symbol or word in a sequences is essential when the input is noisy or ambiguous, like with speech recognition or handwriting recognition [112]. Jurafsky and Martin [112] exemplifies the use of prediction in spelling correction with the phrase "Their are two midterms in this class" where "Their" is a mistyped "There". The bi-gram (an n-gram of size 2) "There are" is much more likely to be correct and therefore a spellchecker would be able to detect and correct such an error.

2.5.3 *Design Patterns*

Design patterns is a concept emanating from the idea that design solutions can be described on a more abstract level so they can be re-used in similar contexts. In 1977, the architect Christopher Alexander developed a language of patterns, a large tome containing 253 patterns related to the construction of the physical environment. The initiative aimed to empower individuals to express their ability to design or participate in design processes with the aid of a informal grammar. Design patterns was developed as part of a movement focusing on understanding design methods (*cf.* [110]). The patterns cover the problem area on a wide range from regions, via cities down to furniture. The pattern language could be seen as a generative grammar with two major components; "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice" [7]. Intrinsically, a pattern has two components (problem and solution) but its usefulness is not in a specific solution to a problem, but that a pattern is a general starting point of a solution to a problem you are faced with.

The approach was later applied to computer software design in 1994, when Gamma et al. [82] published a set of descriptions or solutions for how to solve reoccurring problems in object-oriented design [82]. The solutions are more structures compared to the ones described by Alexander and they are more like templates that can be employed in different situations. For instance, the SINGLETON pattern ensures that a class only has one instance and a global access point to it. Gamma et al. [82] demonstrate this pattern in C++ with an operation that allows access to the unique instance (implemented as a static member function, see listing 2 for the class declaration). Gamma et al. [82] introduced a set of sections of the pattern description;

PATTERN NAME AND CLASSIFICATION A name for the pattern that captures its essence. For the SINGLETON pattern it is to ensure that a class only has one instance.

INTENT What the design pattern does, its rationale and what particular design issue or problem it address. A SINGLETON provides a solution to the situation of a class only have exactly one instance.

ALSO KNOWN AS Other known names of the pattern.

MOTIVATION A scenario that illustrates the problem and how the solution solves the problem. The SINGLETON pattern addresses problems when the software designer wishes to certify that there is only one-of-a-kind resource, like only one file system but many drives or only one printer spooler but many printers.

APPLICABILITY In what situations the pattern can be applied and how to identify these situations. The SINGLETON must have exactly one instance of a class and it must be accessible from a well-known access point.

STRUCTURE A graphical representation of the classes in the pattern.

PARTICIPANTS The classes and objects that takes part in pattern. SINGLETON is the only participant in this pattern but the Instance operation is important. It may be responsible for creating the instance but not necessarily.

COLLABORATIONS A description of who the participants collaborate with. Clients access a SINGLETON instance through its Instance operation.

CONSEQUENCES How does the pattern support its objectives, trade-offs and results. SINGLETON has several benefits: controlled access to sole instance, reduced name space, permits refinement of operations, permits a variable number of instances if the design decides so and it is more flexible than class operations [82].

IMPLEMENTATION Practical hints and techniques that the programmer or designer should know when trying to implement the pattern (see listing 2).

SAMPLE CODE Small examples that illustrates the implementation in C++ or Smalltalk.

KNOWN USES Examples of the pattern in real systems in different domains. SINGLETON could be used as metaclass (the class of a class) since each metaclass has one instance.

RELATED PATTERNS Patterns that are related to this pattern, their differences and patterns that should be used with this pattern. Since SINGLETON is a creational pattern it is related to ABSTRACT FACTORY, BUILDER and PROTOTYPE.

Listing 2: The Singleton class declaration [82]

```
class Singleton {
        public:
                static Singleton* Instance();
        protected:
                Singleton();
        private:
                static Singleton* _instance;
}
```

2.5.4 *Game Design Patterns*

The use of design patterns to understand and design games was initially suggested by Kreimeier [122] and later followed by Björk & Holopainen who developed a large collection bordering on 300 patterns [30]. Related and similar approaches to understand game design are the 400 rules project [23] and the game ontology project [246].

A set of publications have explored game design patterns in less general ways than Björk & Holopainen [30, 29] as they typically focus on a destinct aspect like game sound design [8], or a specific game genre like First Person Shooters [102] and Role-Playing Games [207, 64]. Applications of game

design patterns include teaching and communicating game design but also direct design activities like brainstorming, fine-tuning of ideas or exploring unused ideas. More academic purposes include comparing and identifying differences between a game and remakes [40] or sequels [220]. Likewise, exploration of different, less central to core gameplay, play experiences such as camaraderie [25] and pottering [133] in games can be done with the aid of game design patterns.

As an example of game design patterns and rather fitting to the topic of the thesis, Björk and Holopainen [30], have previously defined the LEVELS pattern as: "*A Level is a part of the game in which all player actions take place until a certain goal has been reached or an end condition has been fulfilled*." Björk and Holopainen [30] uses a set of sections to describe the pattern; name, definition, description (including examples of their usage), using the pattern (as a designer in a design situation), consequences (the effect of using the pattern in a game), relations (to other patterns), and references (to related previous work).

Björk and Holopainen [30] have included hierarchies of patterns in their pattern collection which denotes relations between different sets of patterns, something that is similar to the concept of organising patterns in families by Gamma et al. [82]. The SINGELTON is related to PROTOTYPE in such a way that PROTOTYPE instances of a class only have a limited different combinations, like a limited set of templates, whereas a SINGELTON is a unique instance. Likewise, areas in a game world that the player can perceive but currently cannot enter (*i.e.* INACCESSIBLE AREAS), are related to LEVELS because they can be sections in a level that is temporarily blocked by locked doors that can be opened by finding a key in another part of the LEVEL. This relation is commonly realised as part of the action adventure *The Legend of Zelda* [160].

NAME A singel word or phrase describing the concept. In [30] aliases are deliberately excluded to minimise the number of names that needs to be remembered.

CORE DEFINITION A sentence that describes the core idea of the pattern. LEVEL is defined above.

GENERAL DESCRIPTION A short description that is followed by a motivation for the pattern. General properties and examples are also included. For instance, in *Asteroids* [17] the player advances to the next level if all asteroids have been shot. Higher levels become more difficult because the asteroids moves faster and increase in numbers. For example, LEVELS can be differentiated if the theme changes from one LEVEL to the next LEVEL.

USING THE PATTERN Common choices a designer is faced with when trying to use the pattern.

CONSEQUENCES What consequences of gameplay that can appear when the pattern occurs in the game. LEVELS lets the game designer to delimit the game world and its complexity.

RELATIONS Relations between the described pattern and other patterns. Björk and Holopainen [30] uses five categories that identifies the relations: *instantiates*, *modulates*, *instantiated by*, *modulated by*, and *potentially conflicting with*.

REFERENCES A list of related previous work that either inspired the patterns creation or work that contain the main aspects of the pattern. LEVELS are modulated by SAVE POINTS and CUT SCENES etc.

Another example is the SNIPER LOCATION pattern [102] which describe a common First Person Shooter pattern where a player can attack other players with long-range weapons while remaining protected. Hullett and Whitehead [102] used another component while describing the patterns: affordances, which states what aspects of the pattern that can be varied by the designer.

Bordering to design patterns in games as well as procedural content generation, is the approach of having a tool that consists of a set of scripts that is modified by a user to model specific Non-Player Character gameplay situations in Role-playing games like the pattern catalogue proposed

by McNaughton et al. [145] for the *Aurora* toolset and engine, used in the game *Neverwinter Nights* [28]. This approach was followed and extended by Onuczko et al. [164] and McNaughton et al. [144].

NOTES

[1]The term "digital games" will be used throughout the thesis to describe console games and computer games. In regards to related and referenced work the thesis will use the original term.

[2]The ball bouncing game on the Magnavox *Odyssey* and *Tennis for Two* [241] predates Atari's version of *PONG* but is to some extent less well known.

[3]An open world game is a type of game where the player can move freely in a virtual world and is free to choose how or when to approach objectives.

[4]An MMO is a digital game with the ability to supporting hundreds or thousands of players simultaneously.

[5]Or more commonly – console generations.

[6]The first generation was released in 1972 and ended in 1976 with the introduction of exchangeable ROM-cartridges.

[7]*Visual artist* should be considered as a general term for project members concerned with the task of designing visual artefacts for the digital game e. g. 3D-models, animations, user interfaces, textures, etc.

[8]The term independent video games or "indie" games is typically games that are developed by individuals or small teams without the (financial) support of a game publisher.

[9]In [188] the term "algorithmic" is used instead of "automatic" (generation).

[10]The PCG system fulfils the role of the *material*.

3 RESEARCH FOCUS

The important thing is not to stop questioning.
Curiosity has its own reason for existing.

— Albert Einstein (In *LIFE magazine*, 2 May 1955)

This section of the thesis presents the research from the perspective of a set of research questions and the limitations of the research.

3.1 RESEARCH QUESTIONS

The objective of this thesis is to explore, to understand and to show how digital game content can be generated in relation to pre-existing content with the aid of a design method called design patterns. Apart from the design-time (offline generation) it is valuable to explore how this can be done during run-time (online generation). Related to this is the *quality* aspect of the generated content, including that the game content is generated following a certain style. In addition to this is the *evaluation* of content. The research uses two application domains: platform levels and dungeons (see section 2.3) and the problem domain is the game industry (see section 2.1). The main research question (MRQ) motivated and generated a set of sub-questions (RQ) during the research process. We formulate the main research question as follows:

- MRQ: How can design patterns be used for Procedural Content Generation?

In order to be able to answer the following research questions and to create PCG systems that address the challenge, it is crucial to develop and establish knowledge that bridges the gap between design patterns and PCG. Design patterns have previously been used to describe and communicate game design on a fairly high abstraction level whereas PCG systems typically exist on a lower and more computational centered level. In order to support this, a detailed analysis of existing game content have to be performed. During the content analysis the possibilities of design patterns (as an analytical tool) will be understood.

Details are provided by examples and specific instantiations of game design. These game design instantiations are not particularly useful for realisation and implementation as software since they are explicit instances and therefore not fulfilling a common PCG requirement, namely, variation. However, given the practical use of design patterns in object-oriented programming and the problem-solution descriptions design patterns have, it is possible to view them as a useful tool for game and level designers during the design process when the designers solve the problems of designing levels.

- RQ I: How can particular design styles be expressed in a pattern-based PCG system?

The purpose of generating a design style, or rather, more specifically to follow a particular school of design thinking or to follow a specific designer's recognised style is especially useful in a situation of downscaling a game development project when design knowledge is available, but there is limited access to actual human designers, who can provide new content. In PCG, representing style, refers to activity of defining a generative model that follows a particular designer's recognised style or a particular school of design thinking. One motivation for the research question is to address some of the suggested long-term goals and research challenges for PCG [229]. In this re-

search we addressed two of the research challenges, *Representing Style* and *General Content Generators* (see section 2.2).

- RQ II: How can pattern-based PCG systems be evaluated?

PCG systems are by its nature generative systems and as such the amount of content they can generate are (often) limitless. The usefulness of a PCG system is based on the different qualities of the generated content. The development of PCG systems often incorporates some kind of exploration and fine tuning of parameters used in the system and thus it would be useful to quickly be able to evaluate levels without incorporating playtests after every minor change, but instead focus the effort of costly user tests at predetermined moments or major changes. It is often counterproductive to expose end users to incorrect, boring or repetitive content since it may affect sales.

- RQ III: How can variation be achieved in a pattern-based PCG system that retains a certain style or similarity with previous design?

The construction of a PCG system with the goal to maintain a certain design style face the possibility that the solution space (the output) is limited in variation and therefore risk a state of player boredom.

- RQ IV: How can level progression (e.g. pacing, structure and challenges) be implemented in a pattern-based PCG system while still incorporating style and variation of game content?

Digital game content is usually constructed in such a way that a certain order of its consumption is intended, e.g. a certain order of challenges, rewards, enemies, etc. is particularly interesting for a PCG system to be able to generate in order to fulfil style and variation.

3.2 RESEARCH LIMITATIONS

The thesis focus on the technical challenges and does not consider the different aspects of organisational and business issues of game developers like

particular needs of game and level designers while interacting with the PCG systems (e.g. graphical user interface, playtest logging, etc.). Therefore we have not considered the detailed use of a PCG system in a company setting. We have limited us to see the suggested process and artefacts be used in a content pipeline within a game company in the following roles; tool, material, designer and expert that Khaled et al [117] have suggested. Finally, the scope of the thesis is limited to the use of PCG in digital games. It would be interesting to adopt the research presented here with an interface supporting the designers work with content similar to what has been done in research concerning *Mixed-initiative* PCG systems *c.f.* [132, 208].

4 METHODOLOGY

Everyone designs who devises courses of action aimed
at changing existing situations into preferred ones.

— Herbert Simon (In *The Sciences of the Artificial*)

This section of the thesis presents the methodological considerations, and motivations for the study, as well as the research framework, concluding with research design choices made for this work in relation to the research questions (see section 3.1).

4.1 METHODOLOGICAL CONSIDERATION AND MOTIVATION

The research put forth in this thesis is essentially belonging in design science to Information Technology. The design science paradigm has its background in engineering and the sciences of the artificial [196, 97]. As such, it focuses on developing new IT artifacts [97] and addresses design tasks faced by practitioners [134] in a domain. It was chosen as the primary research strategy due to its focus on applied research where problems are solved with the aid of *artefacts*. In essence, this thesis has emanated a set of *artefacts* with the purpose of solving practical problems in the digital game industry by the means of Procedural Content Generation.

4.2 THE DESIGN SCIENCE RESEARCH FRAMEWORK

March and Smith [134] suggested a research framework to strengthen prior frameworks in information technology [92, 108, 138] which were mainly focusing on *variables* arguing that these approaches "fails to provide direction for choosing important interactions to study" [240] as well as they fail to describe the large body of design science research done previously in IT. Furthermore, it does not account for the matter that IT research is focusing on "artificial phenomena operating for a purpose within an environment" and its task together with IT's adaptive nature and that it therefore is a subject to change [134].

The suggested framework by March and Smith has a focal point on *research activities* and *research outputs* (see section 4.2.2 and 4.2.3). The research activities are based on broad types of design science and natural science activities and include: build, evaluate, theorise and justify. Moreover, the research outputs consist of; constructs, models, methods and instantiations [97]. Theorising about the artefacts and trying to justify these theories is connected to natural science whereas building and evaluating the artefacts are tied to design science.

4.2.1 *Environment and Knowledge base*

Since March and Smith put forth their framework, there has been suggestions providing refinement and extensions to it. One of the most influential extensions is the framework suggested by Hevner et al. [97]. It is closer coupled with the behavioural science paradigm, which seeks to develop and verify theories that explains human and organisational behaviour. It is further tied to design science by adopting its goal of extending the boundaries of human and organisational capabilities with innovative artefacts. Most importantly for the research presented in this thesis is the framework's definition of *environment* and *knowledge base* since they provide the input for our research.

The *environment* defines the problem space [196, 97] where the phenomena of research interest exists. In the area of Information Technology, it usually covers people, organisations and their existing or planned technologies. Hevner et al. [97] argues that the goals, tasks, problems and opportunities that makes up business needs in the *environment* define research problems that are relevant.

The *knowledge base* makes up the "raw materials" for Information Technology research by providing two major components of research; foundations and methodologies [97]. The foundations are the stepping stone for the research as it includes *theories, frameworks, instruments, constructs, models, methods* and *instantiations*. While the foundations works as the baseline and background for the research, the methodologies provides rigour when applied appropriately. The methodologies are made up by data collection, data analysis, techniques, formalisms, measures and validation criteria. Given that Information Technology research is based in both behavioural science and design science, the methodologies stretch from empirical data collection and analysis to computational and mathematical methods.

The *environment* as well as the business needs for the domain have been described in the introduction and background chapters (see chapter 1 and section 2.1). The *knowledge base* consists of analysis and/or modelling design patterns in games (see section 2.5.3), algorithms (see section 2.5) and existing PCG knowledge (see section 2.2). The purpose of the analysis and modelling design patterns was to understand specific game content and to implement algorithms that generate content coherent to the patterns.

4.2.2 *Research output*

During the Information Technology research process a set of artefacts are built and evaluated [97]. These research artefacts consists of *constructs*, *models*, *methods* and *instantiations* and act as outputs [134]. The building of the

artefact includes determining functionality and architecture as well as actually creating the artefact with the use of suitable theory [170].

CONSTRUCTS form the vocabulary of the domain and they constitute a conceptualisation which is used to describe problems within the domain or to specify their solutions. As such they make up the specialised knowledge in a discipline in the form of both formalised (entities, attributes, relationships, identifiers, constraints) and informal knowledge (consensus, participation, satisfaction) [134].

MODEL artefacts could simply be viewed as descriptions [134] of how things are. In the natural sciences the model term is often synonymous with theory and a common way to propose a phenomena to be understood in terms of concepts and the relationships between the concepts. In the framework proposed by March and Smith [134] the concern of model is utility and not truth[11]. March and Smith [134] exemplifies the aspect of utility with a semantic data model which is valuable when used for designing an information system. A *Model* is a set of statements that express the relationships between constructs. In design activities, for instance, models may form problem and solution statements.

METHOD artefacts are used to solve a task and they could both be a guideline or an algorithm, and they are based on underlying constructs founded in the solution space [134]. If the method is, for example, an algorithm it would typically be based in a language and/or using a model as an input, perhaps to translate or transform into another model or representation while solving the problem. As such, the utilisation of a method may affect and influence constructs and models for a task. March and Smith [134] exemplifies this with the development of a model that uses other constructs, is a design task in itself, that requires powerful methods and techniques.

INSTANTIATION artefacts are a realisation of an artefact in its environment [134]. The instantiation are typically some kind of information

system or tool and may therefore also precede other artefacts and not just operationalise previously created artefacts. It is quite possible that a research activity begins with the realisation of one instantiation with the aid of models, methods, and/or constructs and later initiate other models, methods, constructs and instantiations. Newell and Simon [155] sees computer science as "an empirical discipline" and that each "new machine that is built is an experiment" which is observed and analysed.

4.2.3 *Research activities*

March and Smith [134] argues that *research activities* in design science are twofold: build and evaluate. In the process of building an artefact it is demonstrated that such an artefact can be built. The construction of the artefact is followed by the other activity: evaluate, which refers to putting forth criteria and assess the artefact's performance against these. In essence, the questions are: does it work and how well does it work. The second question may need development of metrics and the measuring of the artefact according to those metrics. From this point of the process it is possible to theorise and justify its finding with the aid of mathematical justifications or with the aid of data collection and analysis [134].

Peffers et al. [170] emphasises that evaluation methods can take many forms due to the problem venue and the artefact. Examples of possible evaluation methods are: a comparison between the artefact's functionality and the solution objectives, measuring performance, doing satisfaction surveys, elicit client feedback, and to execute simulations [170]. Hevner et al. [97] on the other hand, suggests five design evaluation methods, namely: experimental, analytical, observational, testing, and descriptive. However, Peffers et al. [170] methods could be mapped to fit into the ones mentioned by Hevner et al. [97]. For example, simulations from Peffers et al. [170] are joined by controlled experiment as a method in the experimental method suggested by Hevner et al. [97].

4.3 RESEARCH PROCESS

The research process consisted of a set of activities that produced research output in the form of artefacts. In figure 13, squares represent model artefacts, circles represent instantiation artefacts and octagons represent (analytical) evaluations together with the icon of the two human figures which also represents (experimental) evaluations. The arrows in the model represent how artefacts affected the construction other artefacts.

The research process started out with an initial attempt to answer the main RQ with literature studies of game design patterns and with the development of a model artefact covering the level design patterns of the platform game *SMB* [158] (see PAPER I).

Closely tied to the *model* artefact designed in PAPER I is the two *instantiation* artefacts created for PAPER II since they both are based on the patterns presented in it. The first *instantiation* could be described as a constructive level generator which had a set of patterns which could be modified by parameters indicating the amount of enemies and rewards as well as different types of blocks. The second *instantiation* was implemented as a search-based level generator that searches through the solution space after sequences of level content (represented as integers) that makes up instances of patterns present in the original levels of *SMB*. Together with the two different *instantiation* artefacts, a minor user study was conducted (experimental evaluation).

For the next phase, two forms of evaluations were performed [97]; experimental (three controlled experiments) and analytical (dynamic analysis: expressive range [202]). The experiments sought out to understand the fitness landscapes of the search-based approach (initially tried as one of the approaches in PAPER II with the aid of three different fitness-functions. The fitness-functions were constructed to; 1) certify that every pattern could be found, 2) explore how common the different patterns were and 3) explore

how different weights could affect the patterns presence in the solution space.

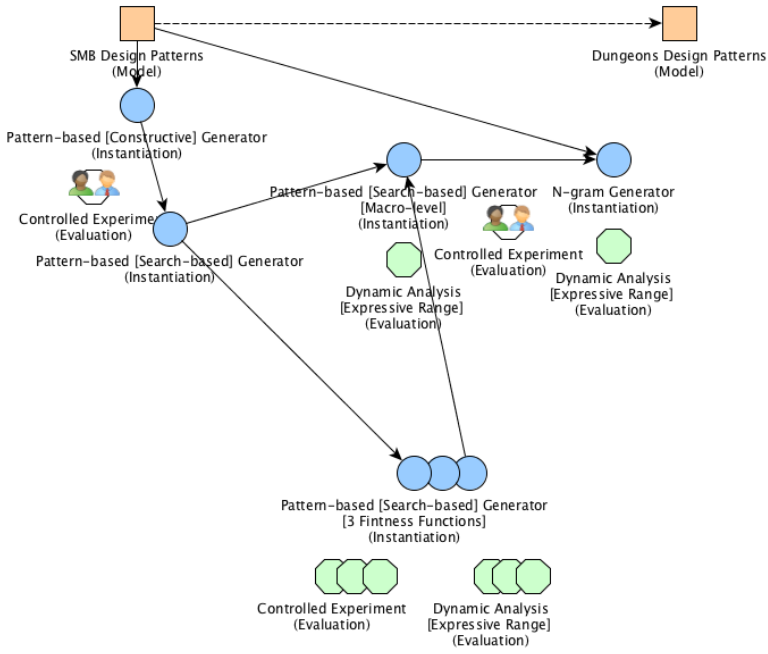


Figure 13: An illustration of the relations of the different artefacts and evaluations.

PAPER I The work in this thesis started out with the analysis of the level design of *Super Mario Bros.* [158] and design the *model* artefact as a collection of game design patterns for platform games.

The paper relates to the RQ I and RQ III.

PAPER II Closely tied to the *model* artefact designed in PAPER I is the two *instantiation* artefacts created for this paper since they both are based on the patterns presented in it. The first *instantiation* could be described as a constructive level generator which had a set of patterns which could be modified by parameters indicating the amount of enemies

and rewards as well as different types of blocks. The second *instantiation* was implemented as a search-based level generator that search through the solution space after sequences of level content (represented as integers) that makes up instances of patterns present in the original levels of *SMB*. Combined to the two different *instantiation* artefacts a small user study was conducted (experimental evaluation).

The paper relates to the (RQ II and) RQ III.

PAPER III For the next phase, two types of evaluations were completed [97] experimental (three controlled experiments) and analytical (dynamic analysis: expressive range [202]). The experiments sought out to understand the fitness landscapes of the search-based approach (initially tried as one of the approaches in PAPER II with the aid of three different fitness-functions. The fitness-functions were constructed to; 1) certify that every pattern could be found, 2) explore how common the different patterns were and 3) explore how different weights could affect the patterns presence in the solution space.

The paper relates to the RQ III.

PAPER IV In this paper, the search-based approach in fitness-function 3 in PAPER III was extended into three abstraction layers; 1) micro-, 2) meso- and 3) macro-patterns. Abstraction layer 1 are vertical slices of original content, layer 2 fully playable sequences and layer 3 following the same pattern order as original levels have.

The paper relates to the RQ IV.

PAPER V At this point the research the concept of expressive range [202] initiated a more rigorous exploration of evaluation of PCG and metrics. 7 generators and the original levels of *SMB* were explored with six different metrics.

The paper relates to the RQ II.

PAPER VI Extending the approach of using sequences as how content is constructed (PAPER II, PAPER III and PAPER IV) together with the re-

search challenge of *Representing Style* [229] and using the actual instances of design patterns in *SMB*.

PAPER VII In order to expand the approach with design patterns as a mean to understand and generate game content, a *model* artefact founded on an empirical basis, namely the dungeons in several RPGs, was developed for this paper.

The paper relates to the RQ III and RQ IV.

PAPER VIII With regard to a deeper understanding of the *instantiation* artefacts created and thus concluding the main study, a user evaluation was conducted (experimental evaluation) by comparing the player experience of a set of generators.

The paper relates to the (RQ II).

4.3.1 *Literature studies*

The literature studies performed for this thesis are based on traditional narrative reviews executed through database searches and snowballing (forward and backward). In order to counter weaknesses in the chosen keywords and defined search phrases the findings were correlated with reading abstracts of publications at specific conferences and workshops were research in game design patterns or procedural content generation are published[12].

4.3.2 *Evaluation*

Hevner *et al.* [97] suggest a set of evaluation methods; observational, analytical, experimental, testing, descriptive. In the thesis we have focused on creating certain artefacts: model artefacts, and instantiation artefacts. In this thesis we have focused on analytical and experimental evaluations based on the nature of the artefacts [170].

NOTES

[11]Regarding to this it should be noted that March and Smith [134] states that "the concern of theories is truth".

[12]Conferences: Digital Games Research Association, SIGCHI Conference on Human Factors in Computing Systems and CHI Play, Foundations of Digital Games, IEEE Conference on Computational Intelligence and Games, AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, Genetic and Evolutionary Computation Conference (GECCO) and Evostar. Workshops: 1-6th Workshops on Procedural Content Generation and 1-4th Workshops on Design Patterns in Games. Journals: Gamestudies, Eludamos, Games and Culture, Simulation & Gaming, IEEE Transactions on Computational Intelligence and AI in Games

5 CONTRIBUTIONS

This chapter summaries the research contribution of the thesis. The chapter starts by presenting the key points and contribution for each research question (RQ) in numerical order before continuing with the main research question (MRQ) and concluding with a summary of the research constructs the thesis has generated.

Table 3: Relationships between research questions (RQ) and papers

RQ	Papers
MRQ	I, II, III, IV, V, VI, VII, VIII
RQ I	I, VII
RQ II	II, V, VIII
RQ III	I, II, III, VI
RQ IV	IV, VI, VII

RQ I: How can particular design styles be expressed in a pattern-based PCG system?

The contribution for this research question is covered by two design pattern collections (*model artefacts*) in paper I and VII. Design patterns were chosen as a basis for the *model artefacts* due to the two component (problem and solution) description it uses, as well as its prior use as description of design in the context of game design. The design pattern approach gives the ability to model more than one solution to reoccurring (design) problems. In order to make the design patterns suit a PCG system with the goal to generate levels, a lower level of abstraction than commonly used were chosen.

All in all, the first design pattern collection consists of 23 patterns grouped into 5 families and was modelled for the game *Super Mario Bros.* [158]. The collection allowed us to create a PCG system with a likeness to the specific style of the original levels in paper II, III, and IV and it made the pruning of the corpus in paper VI possible. Other researchers have built on this research, see e.g. Thompson [220] where the model was used to analyse game sequels with regard to the original game in order to establish whether the game series had innovated between the games.

In addition, the second collection (the design pattern collection in PAPER VII) was developed to allow a PCG system to generate levels that express the four different *dungeon* styles of the different games analysed in paper VII.

RQ II: HOW CAN PATTERN-BASED PCG SYSTEMS BE EVALUATED?

Evaluation is a recurring research problem in the area of PCG due to the indirect nature of the use of the designed artefact. In effect, a PCG system is commonly evaluated through its output. Since generators' main functionality is to generate game content and in potentially infinite amounts, this is a problem in self. How do you evaluate an artefact that generates artefacts? The contribution for this research question is the suggested metrics in paper V, its relations to other metrics in the same paper and to some extent the applied evaluation approaches in papers II, V and VIII.

In paper II three different approaches were evaluated with player tests. The players was divided into three groups and played each generated level in a different order to limit bias of previous played levels. The players' experience was evaluated with a survey, where the specific levels were evaluated on a six value scale measuring; 1) Boring-Fun, 2) Not similar-Similar (to the original SMB) and 3) Easy-Hard. Whereas the results shows limited differences between the different approaches; the search-based approach in paper II was deemed to be more fun than the others but also a bit easier to complete than the constructive approach in paper I.

In paper V a large-sale comparative evaluation of level generators was conducted. In total, 7 generators and the original levels were evaluated with the help of 6 expressivity metrics.

In paper VIII a comparative user evaluation was carried out with 32 users pair-wise comparing levels from five different generators, two generators developed in the scope of the thesis (papers IV, VI) and three generators previously evaluated with expressivity metrics (paper V). Overall, the n-gram generator (paper VI) was seen as most Mario-like and the Multi-level Level generator (paper IV) was seen as generating well made levels.

RQ III: HOW CAN VARIATION BE ACHIEVED IN A PATTERN-BASED PCG-SYSTEM THAT RETAINS A CERTAIN STYLE OR SIMILARITY WITH PREVIOUS DESIGN?

In relation to this research question a set of generators were implemented (*instantiation artefacts*) and reported in papers I, II, III, VI and VI. The four different artefacts used related but different approaches as follows; in paper I, a constructive parameter generator was implemented, in paper II and III search-based approaches were used, and finally in paper VI a learning based approached was used.

The initial approach mentioned in paper I consisted of a PCG level generator which used predefined sequences of level design patterns (see paper I) taken from SMB [158] which could be modified with parameters allowing enemies and rewards to be present and the difficulty for a level was based on the presence of enemies and limitations of rewards. However, the variation this approach allowed was limited and consequently a search-based approach was tried in paper II with the intent of seeing variation as an effect of the traversal of the solution space.

In the first version of the search-based approach, the first level of the original game [158] was divided into vertical slices and each slice was given an integer value. The fitness function assigned the fitness value to a member based on the presence of predefined sequences taken from SMB [158]

(mainly the first level). The specific fitness values for each sequence were decided based on repeated test runs and fine tuned manually.

The second search-based approach, followed the same design principle but included more predefined sequences and was more extensively evaluated with three different versions of fitness functions. In order to understand the change in the output, due to the use of the different fitness functions, the concept of *expressive range* [202] was used to analytically evaluate the artefact. One outcome of this was the ability to understand how the different patterns appeared in the solution space.

The learning-based approach is a a type of probabilistic language model called n-gram model. The n-gram model is based on a corpus of level content which is the basis for the conditional probability tables and put together sequences from these tables to construct new sequences. In paper VI, levels were generated with n-grams with $n = (1, 2, 3)$ were only $n = 3$ worked well whereas $n = 1$ generated levels with broken content and $n = 2$ generated correct but un-characteristic content for the original game. The approach is fast enough for online generation and generates levels in the same style as in the original game but is very dependent of the predefined corpus. Uncommon symbols leads to limited variation. One negative effect of using the original content, as it is, was that long sequences of standard ground were common since they appear twice in every original level (in the beginning and the end). However, after pruning down these long sequences to more common length sequences the generated output became similar to the original content style.

The first and second implementation (*constructs*) [134, 97] used an *experimental* evaluation [97] in paper II. The third and fourth implementation (*constructs*) were evaluated with *experimental* and *analytical* evaluations in paper III and VI.

RQ IV: HOW CAN LEVEL PROGRESSION (I.E. PACING, STRUCTURE AND CHALLENGES) BE IMPLEMENTED IN A PATTERN-BASED PCG SYSTEM WHILE STILL INCORPORATING STYLE AND VARIATION OF GAME CONTENT?

The implementations (*instantiation artefacts*) in paper IV and VI represent two different approaches; the prior is utilising a search-based bottom-up approach and the latter one is using a type of probabilistic language model called n-gram model. The search-based bottom-up approach uses a common evolutionary search strategy but works in a two step fashion, were members of the population are ranked by 1) presence of pre-defined level design patterns (more patterns are better, more complex patterns are better) and 2) the order of the pre-defined level design patterns (the more the order of the patterns match original game levels the better).

MRQ: HOW CAN DESIGN PATTERNS BE USED FOR PROCEDURAL CONTENT GENERATION?

Summing up, the studies related to the main research question resulted in two design pattern collections (*model artefacts*), a set of different implementations (*instantiation artefacts*) and their evaluations (considering the pattern-based metrics belonging in *method artefacts*). All in all the main research question includes the following *constructs* [134, 97]:

- Level design pattern collection for the seminal platform game *Super Mario Bros.* (paper I)
- Level design pattern collection for four types of *dungeons* (paper VII)
- Constructive level generator (paper II)
- Search-based level generator (paper II, III)
- Multi-level search-based level generator (paper IV)
- Learning-based level generator (paper VI)

- Pattern-based metrics (paper V)

The evaluations of the implementations were both experimental (paper II) and analytical evaluations (paper III, IV, VI). A comparative user evaluation was also conducted (paper VIII).

6

CONCLUSIONS AND FUTURE WORK

The thesis has described conceptual and applied research in game design patterns but foremost in the area of Procedural Content Generation. It has focused to explore the possibilities of generating game content in a realistic setting where existing content artefacts set the direction for further generated content. In order to deepen the knowledge of PCG artefacts in relation to possible end-users, some empirical research has been conducted and included in the thesis.

The performed studies have shown that the use of game design patterns and pattern-based PCG is a promising approach to understand and to generate game content. The use of game design patterns have been applied on a different level than previously. Moreover, it has been demonstrated how design patterns instances at different levels: micro-, meso- and macro-patterns can be used for generation of game content. Closely tied to this is the use of patterns together with different approaches to pattern-based PCG. Finally, the empirical studies show that the proposed methods perform better than existing PCG methods in many respects.

A possible development for the pattern-based PCG approach is to explore other methodologies for generation. If it is possible to define a pattern as a part of a larger model it may be possible to continue the research in both a more formal way and a semi-formal way and thus other generation approaches are available to use and explore.

On the more formal side of generation methodologies, formal languages, plans and logic or constraint satisfaction may be fruitful to combine with the design pattern approach. A semi-formal way would be to describe a possible solution to a pattern as a series of game actions and this descrip-

tion could then be mapped to content (similar to micro-slices) that match these game actions. The semi-formal way may be similar to the situation of using a program language to describe the solution of a pattern, bytecode is then generated from that description which could be executed by a software interpreter i.e. the generator itself.

It would be interesting to see how Machine learning could be applied to analyse content from a pattern-based perspective, possibly with the use of autoencoders and Recurrent neural networks.

In the area of game design patterns, a natural step would be to conduct a series of analyses of game content in different game genres or game domains. This would then both validate the approach of using low-level game design patterns to model and understand *game content* as well as extending the research in game design patterns. In regard to this, a set of level design patterns covering the different domains would also be a possible future contribution. Perhaps it would be fruitful to investigate how patterns could be used in different genres, or in relation to PCG, look into how patterns can reoccur in different games.

Another possible extension to the study presented in the thesis could be to apply the approach to generate different kinds of content like racetracks and RTS maps, not just levels and dungeons. In fact, applying the approach to content that is not connected to game space could also be beneficial. One such endeavour could be to pursue generation of quests and missions in RPGs with a pattern-based approach.

Part II

PAPERS

7

PAPER 1 – PATTERNS AND PROCEDURAL CONTENT GENERATION

Steve Dahlskog and Julian Togelius

ABSTRACT

Procedural content generation and design patterns could potentially be combined in several different ways in game design. This paper discusses how to combine the two, using automatic platform game level design as an example. The paper also present work towards a pattern-based level generator for *Super Mario Bros.* (SMB), which is based on an analysis of the levels of the original *SMB* game where we found 23 different patterns.

PUBLISHED IN

Proceedings of the First Workshop on Design Patterns in Games

doi:10.1145/2427116.2427117

PATTERNS AND PROCEDURAL CONTENT GENERATION

7.1 INTRODUCTION

This paper discusses the relation between procedural content generation (PCG) and design patterns in games, and presents work-in-progress on a design pattern-based level generator for *Super Mario Bros.* (SMB). Procedural content generation in digital games refers to the automated or semi-automated creation of game content using algorithms. Design patterns are a way of structuring design and the design process into recurring elements. This way of thinking originated in architecture, has had major impact on software development and has recently been applied to game design.

In the paper, we will first discuss procedural content generation and in particular its role in providing variation in games. We will then discuss design patterns, and how they can (and have been) used for PCG. We then describe ongoing work with identifying and describing level design patterns in *SMB*, and building a level generator based on these patterns.

7.1.1 *Procedural content generation*

In the digital game industry the PCG has successfully been used for the creation of variations of content to enable replay of a digital game. The types of game content that have been generated range from game worlds, levels and items to ornamental decoration. Examples of generation of complete game worlds include *Terraria* [175] and *Civilization* [147]. *Diablo* [34], *Borderlands* [90] and the *SpeedTree* [105] middleware are examples of generation of levels, items and decoration, respectively. An interesting example of PCG being used as a form of data compression is the space trading game *Elite* [3] where David Braben and Ian Bell succeeded to squeeze in 8 galax-

ies with 256 stars each into the limited (22 kB) memory capacity of the BBC Microcomputer with the help of pseudo-random numbers [214].

7.1.2 *Structures, noise and meaning*

Games are in many aspects a combination of designed structures. Rules govern the game's use in the play process in a way that creates meaning to players by allowing or disallowing actions. Levels guide the player through the game world, and sometimes, game space as well, because of its way of structuring challenges and rewards in the game space. The game's story builds meaning to the actions of non-player characters (NPCs) and other entities populating the game world and together with quests, the story provide purpose to the challenges the player is presented with. Thus we conclude that structures in games are fundamentally entities that are interconnected with each other by relationships in a stable and consistent matter for a single game. They should also be observable and recognisable from the user's perspective, either for the player character (PC), on the player's experience level or beyond the game, like game genres and themes. It is from the different structures we define meaning and consistency in abstract things as digital games.

"Structure (in [...] games) operates much like context, and participates in the meaning-making process. By ordering the elements of a system in very particular ways, structure works to create meaning." [181]

Typically the designed structure is a functional one if helps to create meaning for the player. By analysing the structures in games we can find the reason to why a certain game makes meaning in a certain way. Digital games are with few exceptions (e.g. *Dark Room Sex Game* [57]) relaying on its ability to convey information on the game state through visual output. In action games, more often than not, the player needs to scan, interpret and assess the information of the game state at high speed. The assessment of the game state allows the player to interact with the game. In design practice the user's ability to act is conveyed through affordances and the user's inability to act is conveyed through constraints [162]. In order to be able to

procedurally generate suitable content for a digital game the algorithm has to be able to produce output that is meaningful to the player. Unless the player is able to understand the output's meaning it will be experienced as noise or make the player chose a less advantageous or perhaps even make the player fail.

In the process of game development the meaning and structure is expressed in the content the game designer and level designer provides. The game designer provides the development team with the overall picture of how the game should function. "Level designers use a toolkit or 'level editor' to develop new missions, scenarios, or quests for the players. They lay out the components that appear on the level or map and work closely with the game designer to make these fit into the overall theme of the game." [78]. However, this job is not an easy one because for some "Level design is an art..." [78]. We believe that a helpful tool like PCG should be configured in such a way that the meaning and theme of the game and of its content is conveyed through it. The content in a game is not just randomised and thrown out pieces but rather carefully crafted and well placed pieces. If a platform is unreachable in a platform game it has to have a purpose other than being there for the player character to walk on. It is not hard to imagine the frustration of a player trying to reach a reward that is out of reach. Level design is perhaps a form of art because of the balance between providing challenges and rewards. A challenge that is too simple or too hard makes the player bored or frustrated. The essence is to provide the content in a way that a player can rise up to challenges and barely make them. In state-of-the-art game development this is solved in the painstaking process of play-testing and iterative design [78]. It would therefore be interesting to try to utilise previous play-testing efforts and a possible way of doing this is to be able to read a previous level design and generate new content from this in such a way that the meaning of the content is not lost but still new.

7.1.3 *A little less randomization, a little more variation, please*

In a previously proposed taxonomy for PCG [228], the question of the "right amount of variation" in relation to different runs of an algorithm. This poses two questions concerning what the "right amount" of variation consists of, namely; 1) is the same amount of variation desired across all content types and 2) how much, and what type of, variation does the game designer want? It is very plausible that the game (or game mode) itself is in some way dictating how much the right amount of variation due to its design. Considering the job of the level designer there is a risk that the game becomes too varied so that the overall theme of the game gets lost or the meaning or structure of the game confuses the player. However, the aim is to create an enjoyable experience for the player. In the game industry this is achieved by carefully crafting the game and its content in relation to the play tests that are performed on invited players of the right target group. In PCG terms, randomisation is necessary but not sufficient for automatic level design purposes, as the output of the generator must be controlled in order to fit with the meaning and structure of the game.

7.2 DESIGN PATTERNS

In the seventies, the architect Christopher Alexander developed a language of patterns. The intent was to allow individuals to express their ability to design with the aid of a informal grammar. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice" [7]. In essence the pattern has two components (problem and solution) but the value lies not in the specific solution but in the generalisation of a solution space. This way of thinking was brought into computer software design in 1994, when Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a set of descriptions or solutions for how to solve common and recurring problems in object-oriented design. The solutions are not a

finished design per se but instead templates that can be used in many different situations [82].

7.2.1 *Design patterns in games*

In games, design patterns can be seen as providing answers to problems faced by the game and/or level designer. However, we can also see each pattern as a problem posed by the designer to the player. We view the content in a level as challenges or problems the player must find a solution in order to continue the progression through the game. The idea of automating the construction of problems with a given solution is a strategy to avoid the limitations of constraint checking and thus allowing content to be produced without creating impassable obstacles for the player. Furthermore, if we use previously play-tested problems we can with some certainty reuse problems that are fitting different skill sets and skill levels and thus provide more appropriate content for particular players or player types.

The seminal work of Björk and Holopainen introduced design patterns to game design, and provides the foundations for the contemporary discussion about the topic [30]. The book describes hundreds of design patterns, at different levels of abstraction and with reference to different game genres and tasks of game design. In this paper we will focus on patterns in the design of game levels (and similar spatial designs, e.g. maps and tracks) as opposed to e.g. patterns in game user interface design or rewards.

Hullett recently analysed levels of a common first-person shooter (FPS) game in order to find recurring design patterns that had an impact on gameplay [102]. The patterns he found include arenas, sniper positions and galleries, commonly seen in many FPS games.

The work that is most closely related to our current concern is the work already done on design patterns in platform games. Smith et al. [203] analysed the design of platform game levels, and later devised the Tanagra mixed-initiative level generator [208]. Tanagra uses a constraint solver to generate level geometry in interaction with a human designer. The geometry is generated according to a number of patterns. These patterns occur at

two different levels, the single-beat (micro) level where patterns such as the "gap pattern" and "spring pattern" can be found, and at a slightly higher level, where patterns such as "valley" and "mesa" are composed of three micro-patterns each. These patterns are implemented with some flexibility, as the constraint solver can decide to stretch them to some extent to fit in the overall structure of the level.

Peter Mawhorter [139] describes a level generator for Super Mario Bros based on "occupancy-regulated extension" (ORE). The generator works by connecting a number of small level chunks like pieces of a puzzle. ORE could be seen as a compositional approach to implementing design patterns in procedural level generation. However, the description of ORE which can be found in the literature only allows for small and static non-parameterised level chunks.

7.3 COMBINING PCG AND DESIGN PATTERNS

PCG and design patterns could plausibly be combined in several different ways, even when limiting the context to level design. Perhaps the most straightforward way is creating compositional content generators, that view each pattern as a spatial design element and simply combine these elements by connecting them next to each other. This can be done either with static elements, as in Mawhorter's ORE, or with parameterised elements, as in Tanagra. Patterns could be connected sequentially in one dimension (as in Tanagra), two dimensions (as in ORE) or potentially in three dimensions. It is also conceivable to stack patterns, i.e. place several patterns at the same place. This would have the effect of modulating one pattern by another. Some patterns may fit well to certain patterns if it is placed before or after.

Another way of using patterns in the PCG process is to use patterns as objectives, e.g. as evaluation/fitness functions or constraints in search-based PCG. The existence of particular patterns could be seen as desirable or undesirable properties, biasing the search in content space so that the resulting content would be more likely to include or not include certain patterns. Such an approach could potentially lead to more variation than the

composition-based approach, but is also more computationally expensive and harder to predict. An example of this approach is the "choke point" evaluation function in a recent attempt to evolve maps for the StarCraft real-time strategy game [224]. Maps which contain choke points are assigned higher fitness and the results of the level generator are therefore likely to contain this particular pattern.

With both approaches, patterns can be selected that are particularly well suited to a particular player, for example in order to maximise entertainment as predicted by a player model.

7.4 A PLUMBER IN A STRANGELY DESIGNED LAND

In the classic (action) platform game *SMB* [158] the player guides the protagonist Mario (in single player mode) through the world of the Mushroom kingdom where platforms, holes in the ground (gaps), huge green pipes, boxes and blocks acts as aid and obstacles. Furthermore the land seems to be filled with aggressive and deadly enemies like Goombas, Bloopers, Bullet Bills, Buzzy Beetles, Cheep-Cheeps, Hammer Bros., Koopa Troopas, Koopa Paratroopas, Lakitu, Piranha Plants, and Spinies.

SMB consists of 8 worlds with 4 levels each and 11 bonus areas. The bonus areas are often small levels containing extra rewards, like coins and power-ups, while other bonus areas contain warp zones. The warp zones functions as "portals" [203] to other worlds or levels other than the next in sequence allowing a more experienced player move through the game without risking the loss of Mario's "lives". The last level of each world (the levels named 1–4, 2–4, etc.) takes place inside castles and end with a fight against the main antagonist Bowser. These "boss fight" levels are different than the other levels in such a way that they contain long straight sections with few obstacles and end with a hanging bridge over a lava pit where Bowser is supposed to be dropped into.

In the following section, we present our case study of analysing level content in order to find level design patterns in *SMB*.

7.5 LOOKING FOR PATTERNS IN ALL THE RIGHT PLACES

In order to find patterns in *SMB* we apply a combination of heuristic analysis [55] and rhythm groups [42] [203]. Rhythm groups "are often fairly small, encapsulating challenging sections of gameplay" [203]. By dividing every level into sections separated by areas where no or limited threat is imposed upon the player's avatar we get reasonable sized sections to compare, group and classify into patterns.

Every level in *SMB* contain about 15 beats (only 10 beats in level 1–4 and the maximum 30 in 8–1 with an average of 15.5). However, not all of these beats are unique enough to qualify as patterns but *SMB* contains more than the 4 geometry patterns and the 4 multi-beat patterns similar to those used in Tanagra [208]. We base the names of the different groups on the names used in Tanagra [208]. It should be noted that we base our suggestions on the analysis of level 1–1, 1–2, 1–3, 2–1, 2–3, 3–1, 3–2, 3–3, 4–1, 4–2, 5–1, 5–2, 5–3, 6–1, 6–2, 7–1, 7–3, 8–1, 8–2 and 8–3. Thus omitting 1–4, 2–4, 3–4, 4–4, 5–4, 6–4, 7–4 and 8–4 due to their focus on the fight with Bowser and 2–2 and 7–2 which have an underwater setting. The levels 4–3 and 6–3 includes a sort of timed platform which falls down if Mario remains too long on them which might yield a doubled number of suggested patterns due to the time limitation.

7.5.1 *Examples of Super Mario Bros design patterns*

In this section of the paper we intend to present our suggested patterns briefly together with illustrations of a few of those. The full list of discovered patterns can be seen in table 4 and 5. The illustrations aim to clarify our suggested patterns together with one or more solutions of how the player can solve the problem.

One could argue that the 23 suggested patterns are really only variations on five patterns, one for each group. This points to the problem of choosing a relevant level of abstraction when analysing a level into patterns. We have chosen a relatively fine-grained analysis, as we want to point out that there

are meaningful differences in terms of gameplay between patterns that are superficially very similar. For example, two 2-hordes afford different solutions than one 4-horde (you could jump and land between the two enemy groups in the first case, but not in the second). We note that it would be plausible to see the five groups we identified as "macro-patterns" and the 23 patterns within them as "micro-patterns", but we will not pursue this semantic point any further here.

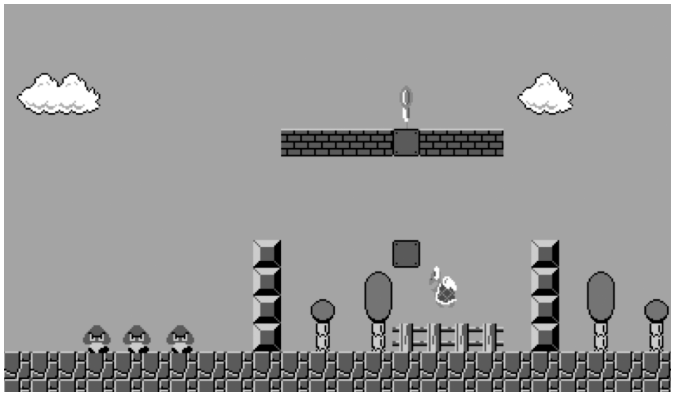


Figure 14: The 3-horde and The Roof valley patterns.

Figure 14 contains the two patterns "3-Horde" and "Roof valley". "3-Horde" can be solved with a triple short jump, a long jump or a medium jump onto the first or last Goomba. The medium jump onto the last Goomba can with good timing allow the player to reach the pillar in the following "Roof valley"-pattern. The "Roof valley"-pattern is provided with a Koopa Paratroopa which, if timed properly, provides a boosted jump out of the valley. If the player misses the timing of the jump altogether, he is faced with a Paratroopa in the valley in which he has failed to jump out of. Figure 15 also contains two patterns; a now familiar "3-Horde" and a "Pillar gap"-pattern with Piranha plants forcing the player to both time the jumps with the movement of the plants in and out of the pipes movement as well as the escalating height of the pipes. By using Piranha plants the player may lose a life even though a correct "side-jump" between pipes normally would save

him because he might end up in one of the plants that is moving in and out of the pipes. If a designer wished make this obstacle easier to pass he could remove the plants. Figure 16 contains two patterns that share the same middle pipe. If we compare the first "Valley empty"-pattern with the second "Valley enemy"-pattern we see that the difference in the height of the pipes suggest that the jump must be more precisely coordinated in relation to the pipe in the second pattern than in the first but if the player times the jump as a landing onto the Goomba, the bouncing of the Goomba will exert enough vertical speed that the ending pipe will be overshot. In figure 17 we can see two simple "Gap"-patterns surrounding the more interesting combination of "2-Horde", "3-Path" and "Risk and reward" patterns. A power-up mushroom is placed in the second horizontal platform in such a way that two Goombas drops down upon the player if the timing is not right. If the player chooses to jump onto the question mark-block so that a Goomba is flipped the mushroom will go the other way and maybe leave the screen (in the original *SMB*, Mario can not go left to scroll the screen left) and the player does not get his reward.

7.5.1.1 *In depth descriptions*

Due to the limited space available we have chosen to describe only a few of the patterns in more detail (see table 6, 7, 8, 9 and 10). We have chosen to exemplify the different groups of patterns by picking one from each group (see table 4 and 5).

7.6 THE PLAN FOR PATTERN-BASED MARIO LEVEL GENERATION

The previously suggested 23 patterns can be varied for enhancement of the play experience in several ways. For instance, the patterns with enemies can be varied by selecting different types of enemies. In *SMB* level 1–2 a Koopa Troopa is followed by two Goombas which enables the player to get variation in how to solve the situation; the player can either short jump and medium jump over the "Enemy" and "2-Horde" patterns or choose to jump onto the Koopa Troopa twice to use its shell to knock out the two Goombas.

The prior solution is less risky than the latter and similarly the reward is lower with the prior solution. The player may also choose to run underneath the platforms and ignore the mushroom. The third possible path is to approach the two Goombas a bit more cautiously and jump over them and then use the third platform to avoid the last gap altogether. Figure 18 may serve as an illustration on the previous situation replacing the Koopa Troopa with the two stand alone Goombas. By recognising a player choosing the solution with the higher reward the PCG algorithm could generate patterns with higher demand on player skill. Within each group of patterns a player that solves a pattern successfully can be faced with a pattern from the same group one step down the list. For all patterns the length of platforms can be changed both for variation and for difficulty. Similarly the patterns can with parametrisation be varied in length and height as well as difficulty in conjunction with adding risk and rewards. Possible parameters are the number of gaps, the length of a gap, the length of a platform, enemy types, amount of enemies and if risk and reward should be present. Figure 19 illustrates a 3-Path and a Risk and Reward pattern combined. At the current stage, we have implemented a composition-based level generator that randomly chooses among the 23 patterns, and generates playable levels for *Infinite Mario Bros*, (IMB), a public domain clone of Super Mario Bros that has been used extensively in game AI and PCG research [223]. It should be noted that IMB already contains a random level generator function with predefined sections. However, our level generator is based on existing content that was placed in *SMB* trying to take advantage of the effort that Nintendo put into designing and play-testing its original successful product. By doing so we hope that an experienced player will enjoy these levels as much as he did playing the original *SMB* but that the variation the randomisation provides will keep the player interested in playing the game for a longer time. In the immediate future, we will add functionality for parameterising the patterns, and selecting patterns to fit particular player profiles. We will also address the problem of preserving playability while stacking patterns (see figure 20); this will likely be done through simulation-based evaluation functions. We also intend to design metrics in-

dicating when a player master a skill enough to be faced with a certain pattern and how often.

7.7 CONCLUSION

In this short paper we have discussed the potential roles of design patterns in PCG, and presented an analysis of the levels in the original Super Mario Bros game into design patterns. Further, we discussed ways of creating levels in Super Mario Bros by combining these patterns. By ordering the patterns in sequence of difficulty we can vary the content in the new levels according to what a player does. In order to further vary the content we can use parametrisation (platform length, gap length, enemy type, risk and reward) in conjunction with a specific pattern. The patterns can be placed in sequence or used together to create varied content.

7.8 ACKNOWLEDGMENTS

Thanks to the anonymous reviewers, Paul Davidsson, and Gillian Smith for insightful comments on the paper.

Table 4: Patterns for Super Mario Bros. grouped by theme part 1.

Enemies	
Enemy	A single enemy
2-Horde	Two enemies together
3-Horde	Three enemies together
4-Horde	Four enemies together
Roof	Enemies underneath a hanging platform making Mario bounce in the ceiling
Gaps	
Gaps	Single gap in the ground/platform
Multiple gaps	More than one gap with fixed platforms in between
Variable gaps	Gap and platform width is variable
Gap enemy	Enemies in the air above gaps
Pillar gap	Pillar (pipes or blocks) are placed on platforms between gaps
Valleys	
Valley	A valley created by using vertically stacked blocks or pipes but without Piranha plant(s)
Pipe valley	A valley with pipes and Piranha plant(s)
Empty valley	A valley without enemies
Enemy valley	A valley with enemies
Roof valley	A valley with enemies and a roof making Mario bounce in the ceiling

Table 5: Patterns for Super Mario Bros. grouped by theme part 2.

Multiple paths	
2-Path	A hanging platform allowing Mario to choose different paths
3-Path	2 hanging platforms allowing Mario to choose different paths
Risk and Reward	A multiple path where one path have a reward and a gap or enemy making it risky to go for the reward
Stairs	
Stair up	A stair going up
Stair down	A stair going down
Empty stair valley	A valley between a stair up and a stair down without enemies
Enemy stair valley	A valley between a stair up and a stair down with enemies
Gap stair valley	A valley between a stair up and a stair down with gap in the middle

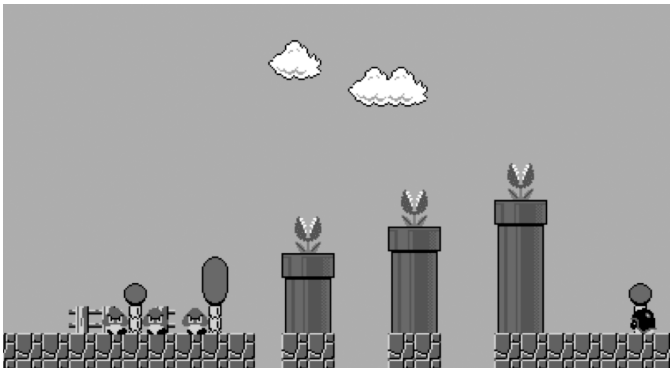


Figure 15: The 3-horde and the Pillar gap patterns.

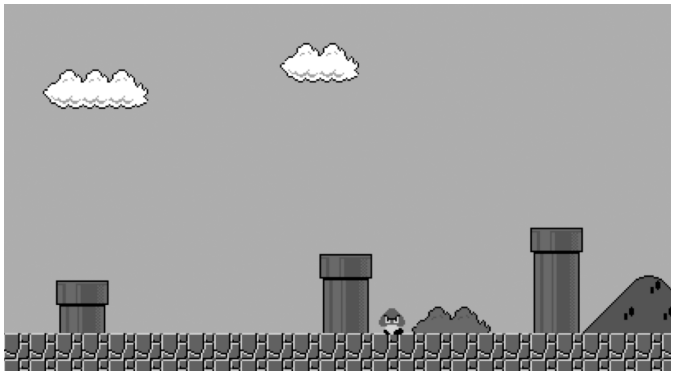


Figure 16: The Empty Valley and the Enemy Valley patterns.

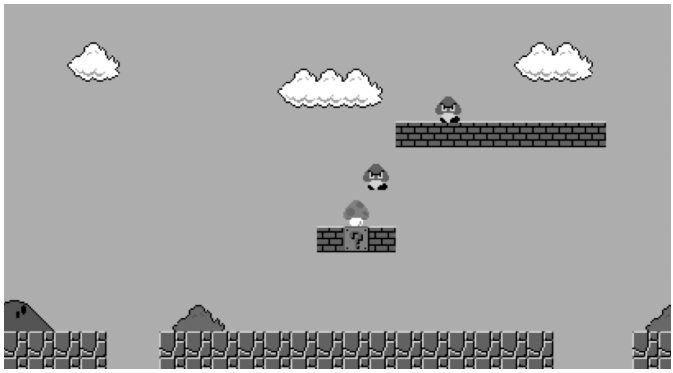


Figure 17: The Gap, the 3-Path, the Risk and Reward and the Gap patterns.

Table 6: 4-Horde Pattern Description.

Enemies – 4-Horde	
Problem	The player can act and traverse the level slowly without risk. The player masters the long jump and can therefore jump over multiple enemies.
Solution	By placing four enemies in a tight formation the maximum jump length is not enough to pass over the enemy which forces the player to use timing to land on any of the enemies for a second jump over the remaining enemies.
Using the pattern	Suitable use is on long platforms so that the enemies do not fall down any gaps. The pattern can also be used in conjunction with valleys to limit the landing area of the player. The pattern can be used to force the player to time running actions if they are placed on high platform allowing them to drop down on Mario.
Comments	Not all enemy types are suitable for this pattern. For instance, enemies that Mario cannot jump onto like Spiny may cause impassible sections of a level. Power-ups should always be considered when applying this pattern. If a too powerful power-up is placed in the wrong position the difficulty of the pattern can be drastically lowered. This pattern should not be misinterpreted as two 2-Horde patterns, the distance between the enemies in the formation is crucial.

Table 7: Pillar gap Pattern Description.

Gaps – Pillar gap	
Problem	The player can jump and traverse vertical obstacles without risk. The player masters high jumping and can therefore jump over high obstacles.
Solution	By placing a series of pillars with a limited width a less skilled player may overshoot a jump and miss the pillar and fall down the gap. The introduction of varying height of the pillars the player needs to master both vertical obstacles as well as the length of the jump.
Using the pattern	Suitable use is near the end of the level so that Mario is high enough to get to the top of the flagpole at the end of a level.
Comments	No power-up can save the player. But a skilled player may jump on the side of the pillar and perhaps bounce out of the gap danger.

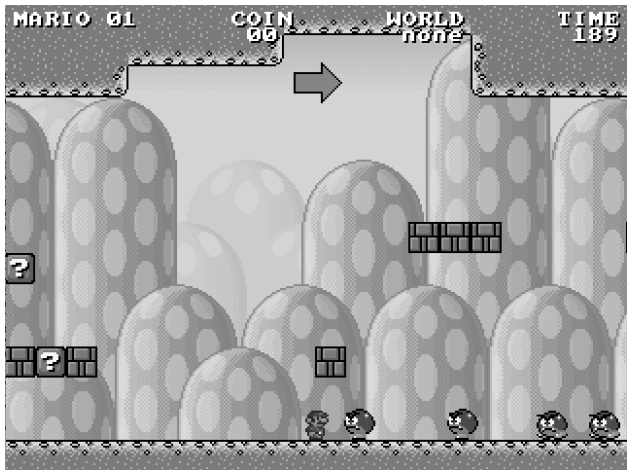


Figure 18: Mario in a “Multiple path” facing “Enemy”, “Enemy” and “2-Horde”.

Table 8: Enemy valley Pattern Description.

Valleys – Enemy valley	
Problem	The player can jump and traverse vertical obstacles without risk. The player can act and traverse the level slowly without risk.
Solution	By fencing in an enemy between two vertical obstacles they player is forced to engage the enemy without it falling through a gap. If a Koopa Troopa is used and the player jumps on it and then jumps on the shell the player will risk losing power-ups or a life due to the high-speeding shell bouncing between the vertical obstacles.
Using the pattern	The pattern can be used most types of enemies with the exclusion of Bullet Bill unless the distance between the vertical obstacles are placed with enough distance in between.
Comments	The pattern needs a sufficient platform length.

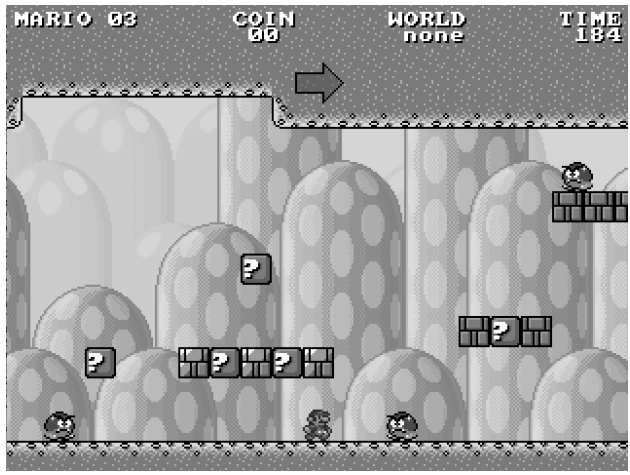


Figure 19: Mario leaving a "3-Path" and entering "Risk and Reward".

Table 9: Risk and Reward Pattern Description.

Multiple paths – Risk and Reward	
Problem	The level layout is more or less linear and the player's choice is limited.
Solution	Provide multiple paths where rewards, gaps and enemies are placed so that the player is forced to choose a specific way through this section of the level. The player can choose the specific path according to his/her skill and risk appetite as well as what the player consider their "favorite" obstacles or enemies to be. The need of a specific reward or power-up may also affect the choice.
Using the pattern	The primary use is to create variation. The pattern could also be used to introduce a needed reward or power-up or to add a more relaxed section of the level.
Comments	The introduction of a different type of decision making may affect the player's reaction time and therefore the distance between the beginning of the pattern and enemies and gaps must be thought through.

Table 10: Stair up Pattern Description.

Stairs – Stair up	
Problem	The player needs to be on a different height and the player character cannot jump high enough.
Solution	By providing tightly placed platforms, blocks or pipes with increasing height the player can jump onto them and reach a higher position.
Using the pattern	The pattern is usable before any section where the player character needs to be high enough but unable to due to limitations in jump ability. In *SMB* it is often needed before the end of the level so that Mario may reach the highest point on the flagpole. It is also useful for variation before a multi-path pattern allowing the player to drop down instead of jumping up.
Comments	The stair should not be placed too high to reach for the player character or be so high that the player is limited in jumping by the top screen unless this is the intended effect.

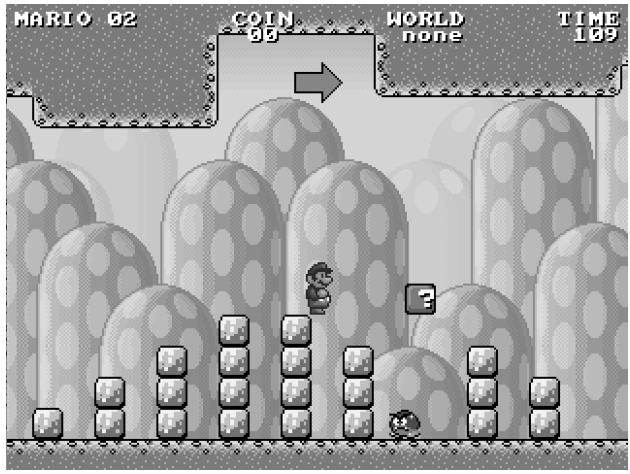


Figure 20: Mario in an interesting combination of pillars and "Stair-up", "Stair-down" and "Roof" without gaps.

8

PAPER 2 – PATTERNS AS OBJECTIVES FOR LEVEL GENERATION

Steve Dahlskog and Julian Togelius

ABSTRACT

This paper discusses how to use design patterns in procedural level generation, with particular reference to the classic console game Super Mario Bros. In a previous paper, we analyzed the levels in this game to find a set of recurring level design patterns, and discussed an implementation where levels were produced from concatenation of these patterns. In this paper, we instead propose using patterns as design objectives. An implementation of this based on evolutionary computation is presented. In this implementation, levels are represented as a set of vertical slices from the original game, and the fitness function count the number of patterns found. Qualitative analysis of generated levels is performed in order to identify strengths and challenges of this method.

PUBLISHED IN

Proceedings of the 2nd Workshop on Design Patterns in Games
SASDG, Chania, Crete, Greece ©2013
ISBN 78-0-9913982-1-8

PATTERNS AS OBJECTIVES FOR LEVEL GENERATION

8.1 INTRODUCTION

This paper aims to combine design patterns and procedural content generation (PCG) in the domain level design. Our approach is geared toward the classic 2-dimensional platformer type of game and especially the classic *Super Mario Bros.* (SMB) [158]. This approach is based on a previous article where we analyzed *SMB*-levels to find 23 different reoccurring patterns of 5 "families"; Enemies, Gaps, Valleys, Multiple paths and Stairs. In that paper, we also used the suggested level patterns to implement a level generator that concatenated these patterns with some variation in height, length or difficulty. In this paper we consider a different take on the relationship between patterns and PCG, seeing patterns not as building blocks but as objectives. This inverts the relation between pattern generation and detection, and conceptually separates building blocks from design objectives. In order to increase the variation of the content but still incorporate the suggested level design patterns of *SMB* we implemented a prototype based on evolutionary computation. In this implementation the representation of level content is made by taking recurring pieces from *SMB*. These pieces are just one tile wide and thus we will refer to them as *vertical slices*. By chopping up Mario levels into finer elements in this way, we can considerably increase the variety of generated levels while still only using existing material. The fitness function for the evolutionary algorithm counts the number of patterns found in the candidate levels. This approach allows us to create levels that resemble the original Super Mario Bros levels both on a micro level, by using existing vertical slices, and on a macro level, by incorporating the same design patterns.

8.2 BACKGROUND

In this section we intend to clarify where our approach is grounded by shortly review patterns, design patterns in games and automatic generation of game content as well as what game content in games may be.

8.2.1 *Design patterns*

During the seventies, Alexander et al. developed a language of patterns for architectural use with the goal to allow others to express design abilities. "Each pattern describes a problem which occurs [...] in our environment, and then describes the core solution to that problem..." [7]. The advantage of the pattern idea is the allow a designer to use a general pattern to solve reoccurring problems. This idea has gradually spread to other areas. In object oriented software development Gamma et al. have defined a set of templates for solving general design and programming problems [82].

Design patterns and games

Design Patterns in digital games (*DPG*) can be used for different activities ranging from scholastic approaches to practical development of (digital) games. Suggested use range from creative aid in design activities where DPG can support knowledge transfer between designers while they generate, communicate and modifiy design ideas and concepts. DPGs can also be used as an analytical tool as well as a learning tool for scholars in both game analysis and game design activities. The design patterns may be used as a tool to understand player behavior during play-testing.

Björk and Holopainen have extensively documented design patterns for game design [30, 29]. Design patterns have also been explored in a variety of aspects, ranging from pattern-related design in relation to game mechanics [5] to specific game contexts, like old-school action games [41], RPGs [206], FPSs [102] and social network game applications in the style of "'Ville" games [126].

8.2.2 *Game content and game development*

Digital games can be seen as the union of two types of digital artifacts: game content and a game engines. The game engine's purpose is to handle user input, to control AI-agents and to present the game content to the user. Typically the game content consists of polygon meshes, texture maps, level geometry, non-player characters, player-characters, missions, quests, items, etc.

Cost drivers

Digital games have become more expensive to develop [213, 115], at least for the average commercial project. The driver of costs is primarily that the advancement of the technical platform that runs the digital game (hardware and game engine) requires more labour to be spent in content development. Customers expect more media and with higher quality [78] which in turn demands more of the engine that uses this media which therefore inherently becomes more complex and may even drive the cost to develop and maintain further.

Game development and PCG

Procedural content generation (*PCG*) is the process of automatically or semi-automatically generating game content. Game developers used PCG successfully to generate game content for different purposes. Examples range from saving developer time with the game *Darwinia* [107], or saving money as for *Just Cause* [18], exploring possible game content as with *The Sentinel* [75], to have variation and creating unique game content as with *Minecraft* [150] and to saving main memory as in *Elite* [3, 214].

Lately *PCG* have spurred interest from researchers and several aspects have been explored as a result. Examples vary from search-based methods to find maps for RTS games [224], levels for platform games [139], and answer set programming for generating mazes [199].PCG could be used in game design and game development in several different ways, depending

on whether the algorithm is seen as a tool, an expert, a designer in its own right etc [117].

There are a couple of approaches that could combine PCG and design in fruitful ways. Firstly, we could use PCG for specific, well defined (and therefore well tested) design tasks, as for instance in *Civilization* [147] where it is used to generate new world maps during run-time (i.e. *online*) or during development as to generate a large set of varying game items, like in *Borderlands* [90], where PCG were used in several aspects but put to extensively use for generating a large set of weapons. Secondly, we could let the PCG process be the central content provider to the whole title as in *Galatic Arms Race*, *FTL*, *Terraria*, *Minecraft*, *Dwarf Fortress* [68, 218, 175, 150, 6]. A third approach would be to start with handcrafted content and letting the PCG process emulate or copy the designers choice with some kind of variation. This approach can be utilized both for *online* or *offline* PCG. The *offline* version could be used both during the principal development of the game title or after release in order to provide more content to generate more sales of the main title or it could be used to create add-on packages for players who has finished the main game.

8.2.3 *Fitting into the pattern*

Let us recall what level designers do; "Level designers use a toolkit or 'level editor' to develop new missions, scenarios, or quests for the players. They lay out the components that appear on the level or map and work closely with the game designer to make these fit into the overall theme of the game." [78]. In order to be able to do this together with PCG we have previously applied a content analysis based on a combination of heuristic analysis [55] and rhythm groups [42, 203] and we suggested a set of level design patterns for the game *Super Mario Bros.* (SMB) [158] and presented a prototype of a level generator[13] [48]. In the previous prototype, we used parameterized but fairly straight forward patterns taken from *SMB* that were randomly picked, modified for difficulty and placed in sequence to form a level. The level generator prototype was able to generate *SMB* lev-

els with high similarity to the original game but some limits in the aspect of variation of the content. The only variation the prototype demonstrated was due to the parameterization (different hight, different length, amount of rewards, amount of risk) of the existing patterns.

For the work-in-progress level generator that we present in this paper we try to approach the problem of generating content by using patterns as the objectives for the evolutionary method rather than generating pattern building blocks as we did with a previous prototype. Our motivation to seeking this approach rather than the reverse is to allow for a greater "creative freedom" or in computer science terms; a larger[14] design space for the PCG-engine than the previous prototype could.

Our goal is to be able to produce a level generator that recreates the particular design and look-and-feel of the levels from the original *SMB*, while still being novel and offering new challenges.

Examples of patterns

Due to the limited space in this paper we will only partially include the suggested patterns that we utilize to generate content (see table 11, figures 21, 22, 24, 25 and [48]).

8.2.4 *Related work*

Previous work in the same are as ours is the *Tanagra* mixed-initiative level generator [208] founded in analysis of platform game levels by Smith et al. [203]. Tanagra is an interactive tool for level designers that utilize a constraint solver for the creation of level geometry according to a number of patterns. The patterns are of two types, single-beat (like "gap pattern" and "spring pattern") and composite patterns (like "valley" and "mesa") which are implemented with some flexibility that can be extended with the aid of the constraint solver to fit the the level. Another approach is the "occupancy-regulated extension" (ORE) [139], that works by adding bits and pieces of jigsaw-puzzle-like parts from levels in a compositional way to solve patterns in level generation.

Table 11: Examples of patterns for *Super Mario Bros.*

Enemies	
Enemy	A single enemy
2-Horde	Two enemies together
3-Horde	Three enemies together
4-Horde	Four enemies together
Roof	Enemies underneath a hanging platform making Mario bounce in the ceiling
Multiple paths	
2-Path	A hanging platform allowing Mario to choose different paths
3-Path	2 hanging platforms allowing Mario to choose different paths
Risk and Reward	A multiple path where one path have a reward and a gap or enemy making it risky to go for the reward

The use of patterns as objectives, e.g. as fitness functions in search-based PCG similar to ours is the "choke point" evaluation function while evolving maps for *StarCraft* [32, 224]. The function assigned a higher fitness to maps that contains choke points and the result of the level generator is thus likely to have that pattern.

8.3 MARIO

The original SMB platform game, initially published by Nintendo in 1985, has been the inspiration for Markus "Notch" Persson's public domain Java-based clone which in turn have been modified for the Mario AI and PCG competitions [223]. In *SMB* Mario has to traverse 8 worlds with 4 levels each. The 4th level of each world is a "Boss-fight"-level with different layout than the other levels.

8.3.1 *The original representation*

The original game from *Nintendo* that was released in 1985 (on NES-cartridge) is a bit problematic to analyze and use as a base for content generation since the implementation is optimized for space rather than readability. This basically means that in order to build something within the original implementation you have to first know the "hex value"[15] of the geometry you want to build, in what page (each level is implemented as a string of pages) you want to place it and where you want to place it in this page. The first level in *SMB* (World 1–Level 1, which we for future reference purposes will call W1L1) contains 13 pages and ends a few tiles after the flag pole[16]. The second piece of geometry you encounter as a player is (see figure 24) implemented as a horizontal brick with the length of five with two "Question mark"-block placed on top in the second and fourth tile (one containing a mushroom and the other a coin). This form of representation is effective when representing the horizontal and vertical block lines (including rows of coins), pipes (different heights), rock-stairs and the castles flagpoles. In fact, some geometry and enemies have double functions, as in the underwater setting in W2L2 and W7L2 (which can be changed to a level on land by manipulating a few hex values). In this case the vertical underwater vegetation becomes vertical bricks and some *Cheep-cheep* enemies becomes *Bullet Bills*. The horizontal green blocks becomes land-based blocks. The representation and optimization for *SMB* level geometry may have affected the design of the levels in a similar way that the limited memory capacity of *Atari 2600* affected the design of the games on that platform [151], since it is more costly in the *SMB*-representation (in terms of memory) to draw more objects and the cost of longer sections are relatively cheap (low or no extra cost compared to draw a single tile).

8.4 REPRESENTATION AND GENOTYPE-TO-PHENOTYPE MAPPING

In this section we present our approach to prototype a search-based level-generator for *SMB*. Our intentions were to apply an evolutionary algorithm

that evolves levels containing the identified patterns. Three different representations were explored: ρ0, ρ1 and ρ2. Since the method we apply is in the realm of stochastic optimization and metaheuristics the problem of how to represent the genotype (the data structure that the evolutionary algorithm acts on) and its relation to the phenotypes (the data structure that is evaluated by the fitness function) was given critical concern [228].

We initially approached the representation problem in the most direct way, thinking of representing levels as two-dimensional matrices with integer values for each block mapping directly to the phenotype (ρ0). This would lead to a very large search space with only a small region consisting of playable levels. This idea was therefore quickly discarded. Next, we considered viewing the geometry of *SMB* as a range of integers spanning from 0 (representing a hole in the ground) to 10 (the maximum height of an obstacle in *SMB*). This representation will be referred to as ρ1.

However this idea was abandoned when we inspected and compared the levels of the original *SMB* with our generated levels and noticed the scarce presence of ground based obstacles. In the original *SMB* most non-moving obstacles are combinations of rocks, pipes and land-based or mushroom-based platforms[17]. Apart from that, the enemies in *SMB* were not present in ρ1 but gave some food for thought for the next idea of how to represent the genotypes when we tried to introduce them in this representation. The fact that a *Goomba* is placed in a certain piece of elevated geometry suggest an explosion of possibilities concerning a specific type of the genotype. Simply put; the need to differentiate between a specific enemy type (11 different ones) and the height (11 different ones) of a geometry type (12 different ones) grows towards a search space that is computationally expensive when we factor in the length of a level and the size of the population. This computational expensive solution may not pose a practical problem until one decide on applying this PCG solution in an online[18] situation with actual users and tries to generate content on the fly.

Thus, further studies of the original content of *SMB* (see section 8.3) led us to decide on a different approach than in our previous prototype (see

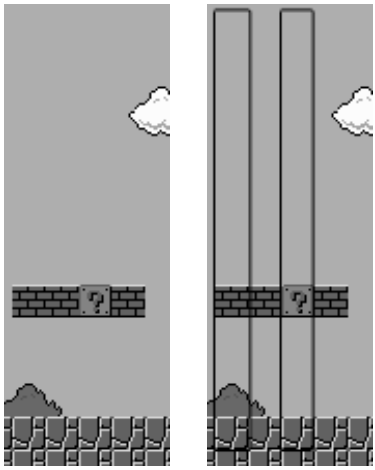


Figure 21: A simple 2-Path-pattern instance in *SMB* to the left. This can be reproduced with only 2 vertical slices indicated with black frames shown to the right.

section 8.2.3) approaching the content as individual pieces and thus keep with the pattern approach.

8.4.1 *Vertical slices*

Our current solution (which we refer to as ρ2) is based on the idea to approach content from the perspective of *Mario* and not the view of the player. In the perspective of the player we travel from left to right but as *Mario* we travel *forward* one step at the time jumping onto objects with varying vertical placement. From this perspective the content of *SMB* can be viewed as vertical slices that together with other slices make up our previous suggested patterns.

Level genotypes are represented as strings of length 200 with an alphabet of 24 symbols. Each symbol corresponds to a vertical slice of Mario level with a length of 1 block and a height of 13 blocks. Levels (phenotypes) are constructed by simply appending vertical slices, giving all levels a length 200 blocks. The 24 slices used for the alphabet are representative samples from patterns extracted from the original *SMB*.

Initially, we were concerned that since the vertical slices are not always compatible with each other, we might need an extra constraint checking function that would be time consuming to design, implement and complex to maintain and debug. However, with the *Mario*-viewpoint and the more detailed analysis of the content in the original *SMB* we concluded that the variation of vertical slices is surprisingly limited. If we observe figure 21 we have a section of *SMB* W1L1, that can be classified as a simple instance of the "2-path"-pattern. In our representation this section is simply a series of vertical slices of two types; the first one is used three times (in position 1, 2 and 4) and the second one is used once (in position 3). The two types contains a ground block at the lowest hight of the level and a brick-block or a question mark-block at height 4. In order to separate the instance of the pattern from other instance we also need a simple piece of ground at height 1. See figure 23 for an explanation of how slices are appended to create levels.

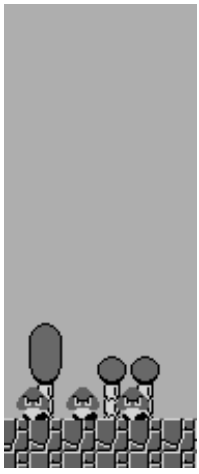


Figure 22: A 3-horde-pattern in the wild (*SMB* World 8 Level 1).

Figure 23: Adding vertical slices to form an instance of the pattern in figure 22.

8.4.2 *Putting pieces together*

In order to explain the vertical slices and how we combine them into patterns we will use an example with an instance of the *Enemy: 3-Horde*-pattern [48] (as in figure 22). The instance of the pattern could then be described as a sequence of three identical vertical slices. Each of the slices are simple geometry (in the example; a ground-tile at "ground" level) with an enemy in a manner portrayed in figure 23.

Example 1

A simple 2-Path-pattern instance in *SMB* which can be reproduced with only 2 vertical slices (one slice with a brick-tile and one slices with a ?-block) see figure 21.

Example 2

By adding a vertical slice with two blocks (a brick-tile and ?-block) and reusing two vertical slices from figure 21 we get this instance of a 3-Path-pattern in figure 24.

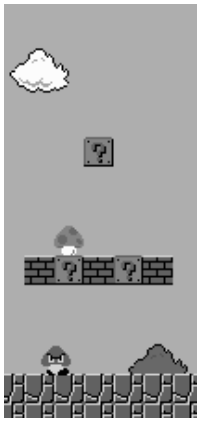


Figure 24: A 3-Path-pattern.

Example 3

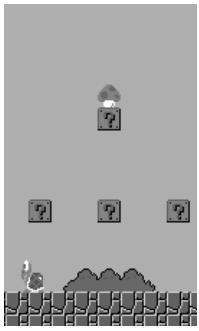


Figure 25: Another 3-Path-pattern.

By adding a vertical slice with two ?-blocks and reusing a vertical slice from figure 21 we get this instance of a 3-Path-pattern in figure 25.

8.5 FITNESS FUNCTION

In our implementation (see figure 26) we use a fitness function to decide which level is the best suited to generate offspring and finally be chosen as the level to be played. In essence we perform a linear search through each member of the population and assign a fitness value to each member, the higher the value is (indicated as low, medium, high in table 12), the greater the chance of surviving the next generation is. Some patterns are only supported by stacking of beginning and endings of patterns where the parts add up to a higher value (use "medium–high" as a value to compare with other values in table 12) except for *Gap enemy* which only need low–medium.

If a level sequence contains a pattern (see section 8.4.2) the individual population member gets a higher fitness value. If a level contains a sequence of symbols representing three consecutive Goombas it is assigned a positive value. Similarly a sequence of rocks with increasing height is assigned a value depending on how long the sequence is. The fitness value assigned is higher if the pattern is uncommon in a random sequence. Unplayable sequences are given a high negative number (but not $-\infty$) allowing breeding with a lower chance of survival in order to allow mutation or cross-over keeping the good part of the genotype for another generation. Uninteresting sequences are given a low negative number in order to remove uninteresting parts of levels. We allow some uninteresting sequences, like a string of simple ground blocks in order to keep some kind of beat-like[19] [208] expression generated from this search-based approach.

The fitness function also contain beginnings and endings of patterns thus allowing stacking of patterns on top of each other. A beginning or ending is typically rewarded less than a full complex sequence. However, if a beginning, a full pattern and perhaps another beginning or ending is in a sequence this will give a cumulative higher value and thus solving the suggested improvement of stacked patterns in the previous prototype [48].

Table 12: Patterns supported in the fitness function.

Enemies	
Enemy	Low
2-, 3-, & 4-Horde	Low
Roof	Medium
Gaps	
Gaps	Low
Multiple gaps	By stacking
Variable gaps	By stacking
Gap enemy	Low–Medium by stacking
Pillar gap	High
Valleys	
Valley	Low
Pipe valley	Medium
Empty valley	By stacking
Enemy valley	By stacking
Roof valley	By stacking
Multiple paths	
2-Path	Medium–High
3-Path	Medium–High
Risk & Reward	By stacking
Stairs	
Stair up & Stair down	Low
Empty stair valley	Low
Enemy stair valley	By stacking
Gap stair valley	By stacking

8.6 EVOLUTIONARY ALGORITHM

In each evolutionary run, we use 200 levels as representations of our population and each genotype is initialized as a uniformly random string of symbols drawn from the 24-character alphabet of vertical slices. We used a simple $\mu + \lambda$ evolution strategy with $\mu = \lambda = 50$ with a combination of mutation and one-point crossover as genetic operators[20].

Before any evolution operation is performed on the population it is evaluated according to a fitness function (see section 8.5). After that the population of 200 members are ranked according to its fitness value. The top 50 percent of the population are kept and the weakest 50 percent are discarded, thus leaving 100 level positions for evolutionary purposes.

We then let the top 50 percent breed with each other and so utilizing the "empty" positions in our population. The breeding is executed as a one-point crossover between pairs in ranking order, in such way that the best ranked is breed with the second in ranking, resulting in two new offspring, and so on. Our implementation of the one-point crossover has a fixed place for the crossover point in the middle of the parents' strings and from this point the strings are simply swapped with each other. In order to certify that we do not get stuck in a local maximum of the search-space we apply a simple mutation operation to the offspring by inject a new random character from our alphabet in a random position. Since we have the opportunity to run the level generator in offline mode our evolutionary search runs for 10.000 generations in the current version of the implementation.

8.7 EXAMPLES OF GENERATED LEVELS

The evolutionary approach together with vertical slices result in levels with both patterns, stacking of patterns and similarity to the original game (compare figure 28, 29, 30, and 31). However, our current implementation does not support the concept of beats *well enough*[21] where the content alternate between high-intense and low-intense parts of the levels. Adding better support to this might solve the tendencies to tightly stack patterns and overfill-

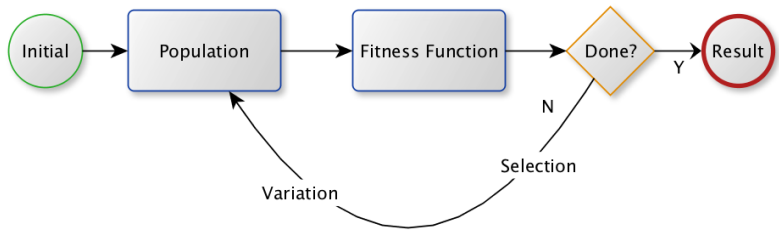


Figure 26: Principal execution of the level generator.

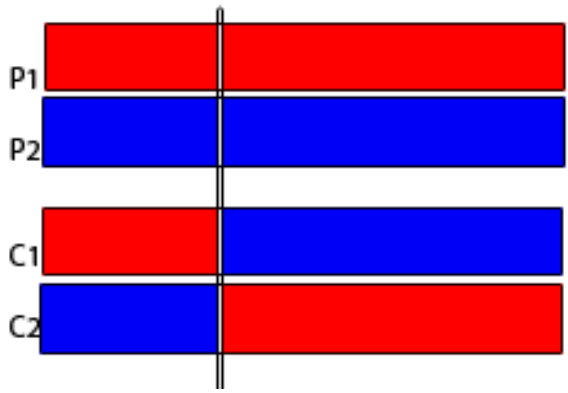


Figure 27: One-point crossover, where parent 1 (in red) and parent 2 (in blue) result in mixed-colored offspring child 1 and 2.

ing game space as in figure 30 and 31. The reverse version of our approach α-level (see section 8.8) in figure 28 is overfilling the game space but at least our level generator does not do as bad.

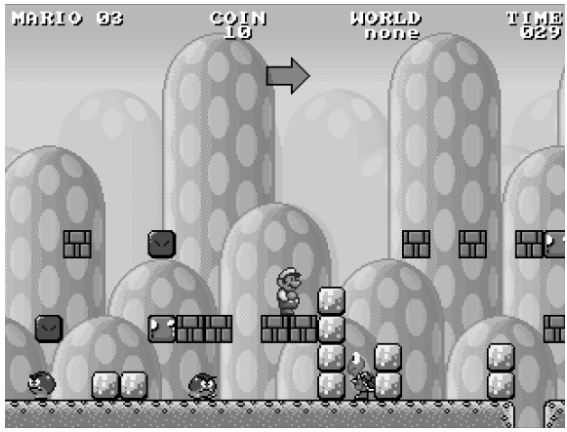


Figure 28: α-level showing tendencies to overfill levels.

8.8 EVALUATION

In order to get some feedback on our prototype we devised a simple play-test with three different levels generated from three different stand-points. We refer to the different levels as α, β and γ. Level α was using a reversed version of our fitness function that in principle, punished any form of pattern or beginning or end of a pattern. Level β was generated using our actual fitness function and level γ used a combination of our pattern-based prototype and imitated original content from *SMB*. The play-testers consisted of 24 experienced players (23 male, 1 female) in their twenties. The test platform consisted of ordinary but high-end PC:s with keyboards as control unit (UI). Our player feedback was gathered through a simple survey. In order to limit bias on previous play-through of other versions three different groups were created with 8 individuals each playing the levels in different order (Group 1: α, β, γ, Group 2: β, α, γ, Group 3: γ, β, α.).

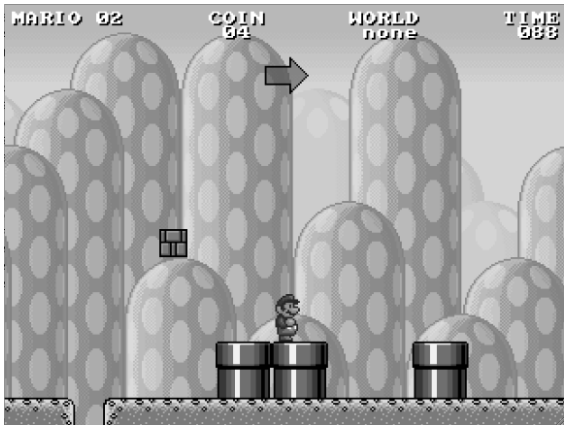


Figure 29: β-level showing tendencies to stack patterns.

Apart from an open-ended question regarding the overall experience with the level and some questions covering general information the play-testers supplied level-specific information on three questions on a 6-value scale. The level-specific questions were on; 1) Boring–Fun, 2) Not similar–Similar and 3) Easy–Hard. The results show a marginal difference between the approaches: our search-based approach (β) seemed to be slightly more fun than the others, more similar to the original than α but easier to beat than γ (see table 13).

8.9 DISCUSSION

Search-based optimization solutions, like evolutionary approaches, work well with patterns in regards to variation and can in our implementation solve the issue of being able to stack patterns. However, the fine-tuning of the fitness function may be problematic when introducing new patterns since the values for rewarding or punish the level can affect previous configuration. We suggest that other content analysis methods are applied when utilizing building-blocks smaller than beats or patterns because this can give an appropriate frequency of reward, enemies and geometry.

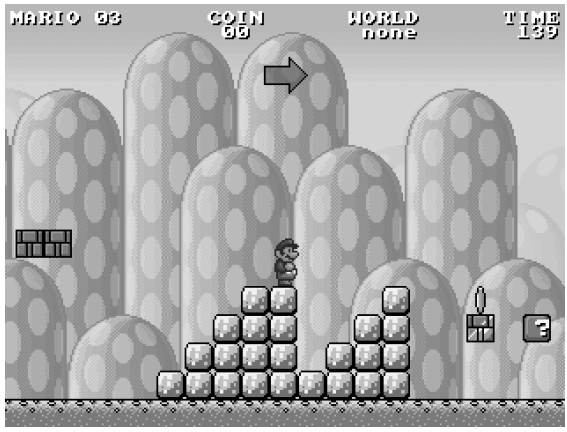


Figure 30: β sometimes stack patterns too close.

Since *SMB* has several levels with distinct *look–and–feel* in the different worlds and levels we intend to implement a set of fitness functions that allow for generating world and level specific content. We have planned and prepared the next phase of the prototyping projects which will include a frequency analysis of game content in order to fine tune the fitness functions according to the different worlds and levels of the original *SMB*.

8.10 CONCLUSION

This paper has discussed how patterns can be used in procedural level generation, and in particular how they can be used as objectives rather than building blocks. An implementation of the idea of patterns as objectives for generating levels for Super Mario Bros was presented. In this implementation, the original levels of Super Mario Bros recur in two ways: as the fine-grained "vertical slices" that are recombined in the evolvable level representation, and the higher-level patterns that serve as objectives. Thus, the evolved levels retain much of the look and feel of original Mario levels, yet the generator can output a large range of diverse levels. An exploratory user study comparing levels that were generated with different fitness functions

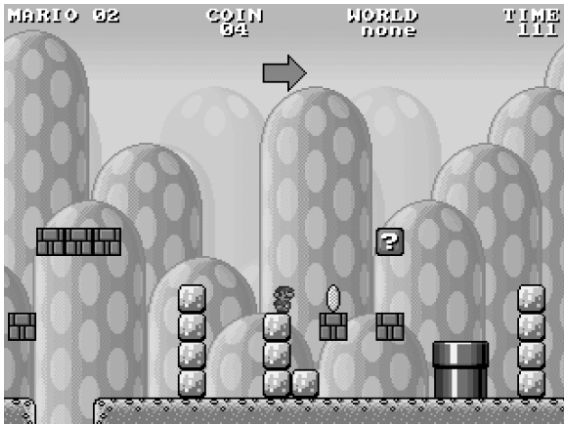


Figure 31: β almost overfill game space as α does.

gave some indication that those that were evolved to maximize the number of patterns appear more similar to the original game.

NOTES

[13]Presented at the *First Workshop on Design Patterns in Games, 2012.*

[14]But still more distinct and limited than full randomization of levels.

[15]A value represented in a positional numeral system with a base of 16 (where the symbols usually are 0-9 and A-F). A byte value is conveniently represented as 00-FF instead of 0-255.

[16]Including the first "empty" screen.

[17]See *SMB* level W1L3 for an example of land-based platforms and level W4L3 for an example of mushroom-based platforms.

[18]I.e. during *runtime*.

[19]A rhythmic variation between exciting parts and calm parts where the player can regain energy to tackle the next exciting section.

[20]The one-point crossover is illustrated in figure 27.

[21]According to some comments by the play-testers. See section 8.8 for more play-test feedback.

Table 13: Results by level.

Version	Avr.	Median	Standard deviation
α Fun	3.75	4	1.041
α Similar	4.125	4	1.062
α Hard	3.5	3	0.791
β	3.792	4	1.159
β	4.625	5	0.906
β	2.375	2	0.989
γ	3.708	4	0.923
γ	4.833	5	0.875
γ	3.292	3	1.006

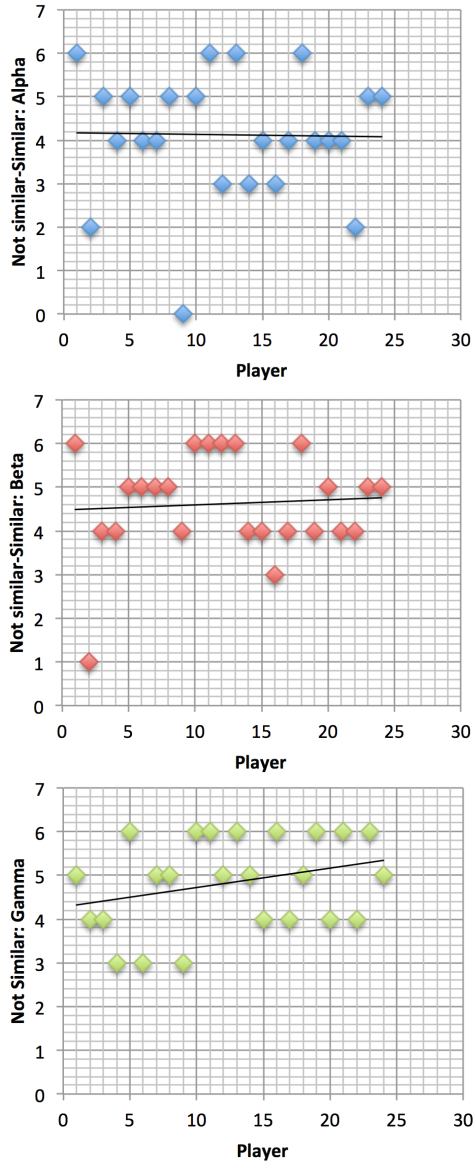


Figure 32: Not similar–Similar, Blue = α, Red = β and Green = γ.

9

PAPER 3 – PROCEDURAL CONTENT GENERATION USING PATTERNS AS OBJECTIVES

ABSTRACT

In this paper we present a search-based approach for procedural generation of game levels that represents levels as sequences of *micro-patterns* and searched for *meso-patterns*. The micro-patterns are "slices" of original human-designed levels from an existing game, whereas the meso-patters are abstractions of common design patterns seen in the same levels. This method generates levels that are similar in style to the levels from which the original patterns were extracted, while still allowing for considerable variation in the geometry of the generated levels. The evolutionary method for generating the levels was tested extensively to investigate the distribution of micro-patterns used and meso-patterns found.

PUBLISHED IN

Proceedings of the Applications of Evolutionary Computation 2014

doi: 10.1007978-3-662-45523-4_27

PROCEDURAL CONTENT GENERATION USING PATTERNS AS OBJECTIVES

9.1 INTRODUCTION

The study of Procedural Content Generation (PCG), i.e. how game content such as levels, items, quests and characters can be created algorithmically, is currently one of the most active topics within academic research on artificial and computational intelligence in games. A large variety of methods have been proposed to generate an even larger variety of types of game content, subject to various objectives and constraints [193]. The work is motivated both by a real industry need for lowering the cost and saving time of content production and enabling endless user-adaptive games, and by academic interest in formalising game design and building creative machines. A recent "vision paper" for PCG research lists a number of open research challenges [229]. One of them is to learn to imitate style: could you build a content generator that was shown a number of examples of the creative output of a human or team of humans, and that then learned to produce more artefacts in the same style that were clearly original but still recognisably of the same style?

Another active research area has been that of game design patterns. A design pattern is a general concept, which has its roots in architecture, but has been applied both to software design and to game design. Game design patterns have so far been identified manually, and the investigation on how to integrate patterns into PCG has just started.

In this paper we demonstrate how practical game design patterns can be combined with procedural content generation to generate game levels that imitate a certain design style, and report the results of a series of experiments using a platform game benchmark. We have previously analysed the classic game *Super Mario Bros.* (SMB) [158] and suggested a collection of

patterns and a PCG tool that produce levels by randomly picking copies of these patterns and modifying them according to a desired length and difficulty level [48].

Our prototype is based on evolutionary computation, where we will search the solution space of combinations of simple building blocks for levels that contain structures at a higher level. This way, we introduce a certain measure of control and constrain the shape of the final level through both the objective function and the choice of building blocks, while allowing a significant amount of variation. In the prototype the representation is relying on existing content in SMB, namely on one tile wide *vertical slices*, which we will also refer to as *micro-patterns*. The micro-patterns are extracted from the original SMB levels. A level is simply a sequence (or string) of micro-patterns — this applies both to the original levels and our generated levels. However, not any sequence is interesting but in our prototype we search for specific sequences or patterns that exists in the original game. These sequences will we refer to as *meso-patterns* and they are our search objective for our evolutionary approach.

We have previously reported initial work on this idea in a workshop paper [49]. Compared to that paper, the current paper describes a more mature system, and reports more in-depth results with several variations of the fitness function and a better characterisation of the generator output.

9.1.1 *Background*

In the seventies, Alexander et al. proposed a pattern language for architectural application on all levels (regions, cities, neighbourhoods, buildings and rooms) thus allowing everybody the ability to express design. Not only structural and material issues are covered but also life experience like the *Street Cafe*-pattern. The pattern language consists of a set of problems in an environment together with a core solution to its corresponding problem [7] thus giving a designer a tool to handle reoccurring problems. This powerful idea has spread to other areas like object-oriented software development where Gamma et al. have defined a set of templates for solving general de-

sign and programming problems [82]. In the context of games have Björk and Holopainen suggested an extensive collection of patterns for game design [30]. Similarly, others have looked into game mechanics [5] and specific game contexts like FPSs [102], RPGs [206], and action games [41]. There have also been some attempts to formulate abstract level design patterns that can be specialised to concrete metrics for different level types [131].

Procedural content generation refers to the (semi-)automatic process of creating game content. One common approach to PCG is the search-based approach, to use evolutionary computation or other stochastic global search / optimisation algorithms [228] for searching the content space. An oft-encountered trade-off in PCG is between control and variation. Methods that have a high variation in output according to some measure usually afford little designer control. Variation can be measured as *expressive range*, the variation along relevant metrics of generated artefacts [202, 187]. Control comes in several flavours: control over style, player experience, difficulty or even playability (e.g. specifying that there is a path from start to end of a level).

9.1.2 *Examples of patterns*

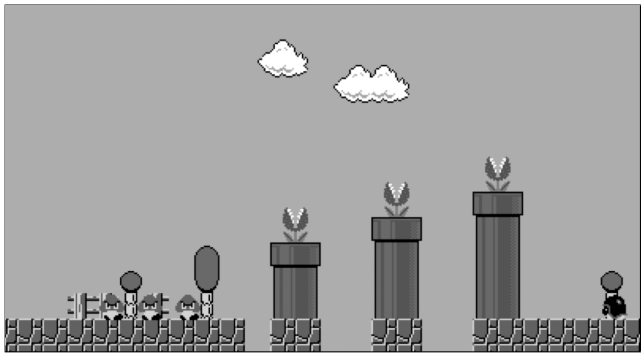


Figure 33: Three consecutive patterns in SMB.

Because of the limited space available we can only briefly mention the patterns that were found [48] in (SMB). The patterns can be grouped into 5 groups; 1) Enemies and hordes, (single and multiple variations), 2) Gaps (single, multiple, variable length, combined with enemies and structures), 3) Valleys (a boxed-in area with structures, possible combined with enemies), 4) Multiple paths (structures horizontally dividing game space combined with enemies and rewards) and 5) Stairs (structures supporting vertical repositioning combined with enemies and gaps). In figure 34 we can see two instances of the 3-Horde pattern (Enemies) and in figure 33 we have a 3-Horde-pattern, a Pillar Gap-pattern and a Enemy-pattern.

9.2 RATIONALE

Our application domain in this paper is the classic 2-dimensional platformer, *Super Mario Bros.* (SMB) [158] and our generator is implemented using the Java-based Mario AI Benchmark[22] [113].

The levels of SMB could be seen as 2D matrices where the cells contain various items such as blocks, coins, enemies, etc.; this is also the internal representation of levels in the Mario AI benchmark. Mario (when small) has the size of 1 cell, and most levels have a length of 100-300 cells and a height of 20 cells. A slice, or micro-pattern, is simply a vertical column of this array – a subarray with length 1. By analysing the levels of the original SMB, we have identified a library of such slices. New levels could be created by combining slices from this library, drawn at random. Such levels would have some similarity to the original levels, as they would not contain any slices that did not exist in the original game. They would not, for example, contain slices where enemies stack on top of each other or the player starts in mid-air. However, these levels would be uninteresting at best, and probably unplayable, as they might contain too long gaps, unclimbable walls, long stretches of nothing, and generally no discernible structure. However, in the space of all possible sequences of slices there should be many permutations that are well-designed, playable levels that are similar to the original SMB levels not only on micro level but also on meso- and macro-levels. How can

we find those levels? In order to guarantee playability we punish unplayable sequences.

9.2.1 *Representation*

Our level representation is a sequence of symbols of length 200, where each symbol stands for a specific *micro-pattern* (a vertical slice) taken from the original human created content. The slice is one tile wide and in our example we have a slice containing a Goomba standing on a ground tile. This tile could be copied in sequence two or three times to make a 2-Horde or 3-Horde pattern (as in fig. 34).

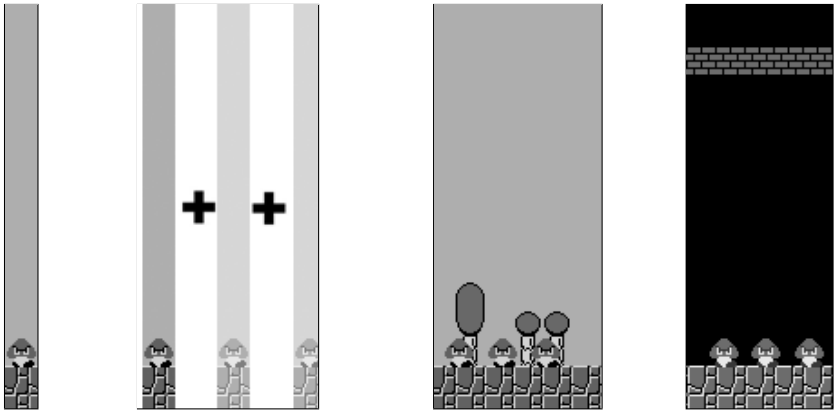


Figure 34: To the far left we have a vertical slice (micro-pattern) with a Goomba on low ground. To the left a sequence of copies of the same slice making up a 3-Horde meso-pattern that in the original game can be found quite often as in World 8, Level 1 seen to the centre-right and in World 1, Level 2 to the far right.

By adding new slices the solution space grows. The levels of the original SMB contain fewer than 200 slices like this. In our representation, we use an alphabet consisting of 23 frequently occurring micro-patterns. Most of the slices come from unique-looking levels like W1L2 (the first level under

ground) and are not reused elsewhere in the game. The advantage of the representation is the ease with which one can generate a level either by the constructive or the generate-and-test approach [228]. One could for example base a constructive PCG algorithm on a *phrase-structure grammar* with pre-checked production rules or by randomly picking slices and evaluate according to constraints. However, we will suggest another approach in the next section.

9.2.2 *Evolutionary algorithm*

The search-based approach taken in this paper is based on a fitness function that rewards the presence of meso-patterns, the higher presence the likelier a member is selected. We apply a simple $\mu + \lambda$ evolution strategy where $\mu = \lambda = 100$ is combined with single-point mutation and one-point crossover. In other words, of a population of 200 we apply *selection* (discarding half of the population), *reproduction* (keeping half of the population and using pairwise breeding to generate new members), *recombination* (fixed one-point-crossover) and *mutation* (the slice at a randomly chosen position in the level has its symbol replaced by a randomly chosen slice).

9.2.3 *Fitness function*

In order to understand how our micro- and meso-patterns interact in the search space we implemented three fitness functions (FF 1-3). The fitness functions were designed in the following way; FF1) a simple uniform reward value for every *unique pattern*, FF2) a simple uniform reward value for *every occurrence of patterns*, and finally FF3) a non-uniform reward weighted value for every occurrence of patterns. The first fitness function worked as a validation of the strings indicating that they could be found (i.e. more than one out of our meso-patterns can be found). The second fitness function was used to explore the frequency of how meso-patterns "appear" in the search space (i.e. how common are the different meso-patterns). The third

fitness function was used to explore how the use of weighted values affects the frequency of meso-patterns.

In order to have some input on the weights to use we chose a simple strategy of calculate a weight by inverting the average occurrence of the patterns giving an infrequent pattern a high weight and a frequent pattern a low weight. By doing so, we propose that we can counter the effect of normal distribution while picking random symbols during the task of initiating and mutating the members of the population. Another issue this strategy would counter, is the varying complexity that the individual patterns have. If we would continue to use a uniform reward strategy for the fitness function, complex strings would run a greater risk to be starved to death in our population due the space it takes over uncomplicated patterns (i.e. short patterns are easily fitted into a member in relation to a long pattern). In order to find different variations of the patterns we designed a set of 43 strings of symbols in different categories of the patterns (i.e. 5 categories of patterns and 23 patterns [48]). These strings, (which we will refer to as rules) were used for a simple linear search, covering each member of the population in each generation.

9.3 RESULTS AND EVALUATION

We performed the experiments in three stages. First, we evolved a large number of levels using the "unique patterns" version of the evaluation function (FF1). We then repeated this experiment using the "all occurrences" version of the evaluation function (FF2). Based on these runs, we evaluated which micro-patterns were most commonly used, and which meso-patterns were most commonly found. These evaluations were used to calculate the weights for a weighted version of the fitness function (FF3). The third and final experiment, using the weighted version of the evaluation function, aimed to see if we could bring about that all patterns were found in a more balanced way.

9.3.1 *Finding patterns*

Table 14: Fitness value variation for 1000 levels counting fitness value based on rules; only one occurrence (FF1), multiple occurrences (FF2) and weighted multiple occurrences (FF3).

Generations	MIN	MAX	MEAN	DEV.	MED.
0 (FF1)	3	8	4.61	0.81	5
10 (FF1)	5	11	7.47	1.02	7
100 (FF1)	8	27	14.94	2.51	15
500 (FF1)	8	31	18.18	3.17	18
1000 (FF1)	9	31	18.97	3.23	19
0 (FF2)	4	10	5.7	1.12	6
10 (FF2)	7	18	11.17	1.74	11
100 (FF2)	13	86	36.98	10.46	37
500 (FF2)	16	183	68.62	30.17	63
1000 (FF2)	18	227	82.17	37.38	73
0 (FF3)	4	202	77.83	36.16	77
10 (FF3)	8	301	121.92	62.83	118
100 (FF3)	20	1030	264.07	149.64	241
500 (FF3)	34	2361	430.33	348.98	337
1000 (FF3)	34	2449	486.20	401.76	374

For each fitness function, we made 1000 independent runs and recorded the fitness values based on the strings. The fitness value worked as a simple "count a rule when it is fulfilled", but only the first time it occur in a level for FF1, for every time it occurred in FF2 and with weighted values in FF3. We can see that the evolutionary approach manages to find more meso-

patterns over time. In order to measure the effect of our efforts of guiding the evolution to find more elaborate patterns we recorded which rules were present in the best member out of our 1000 runs (see tables 15-20).

Measuring the occurrences of a rule in large population should give an indication on how complicated it is to generate an instance of a meso-pattern (rule) in relation to the micro-patterns. Several of the meso-patterns use the same micro-patterns and since the micro-patterns initial occurrence is based on equal chance to be present in the population and a member we can be certain that, given enough time, the search-based approach will affect the distribution of micro-patterns.

Table 15: Found patterns (rules) in FF1-FF3 together with the calculated weight for FF3 based on 1000 runs.

Pattern	Mesa		Straight
Occurrence in FF1	682	686	1001
Average in FF1	0.68	0.69	1.00
Occurrence in FF2	498	480	523
Average in FF2	0.5	0.48	0.52
Weight	2.01	2.08	1.91
Occurrence in FF3	1042	1118	1317
Average in FF3	1.04	1.12	1.32

For FF1, the distribution of fulfilled rules show promise on only 12 of the rules (with occurrence value of 845–2605) and all rules have been fulfilled. However, this is not sufficient to answer the question on how easy they are to find in relation to each other. It is possible that the more complex rules are starved to death in an evolutionary search. In order to explore this we ran FF2 and counted multiple occurrences. The effect of counting multiple instances gives the conclusion that Enemies and Hordes starves most other rules (except two instances of Multi-way and only mildly two other Multi-way). Problematically as it is, we apply weights for FF3 to counter the

Table 16: Found patterns (rules) in FF1-FF3 together with the calculated weight for FF3 based on 1000 runs.

Pattern	Multi-way										
Occurrence in FF1	239	193	50	93	68	193	168	239	197	132	136
Average in FF1	0.24	0.19	0.05	0.09	0.07	0.19	0.17	0.24	0.20	0.13	0.14
Occurrence in FF2	25	83	221	329	11	83	37	25	13	120	127
Average in FF2	0.03	0.08	0.22	0.33	0.01	0.08	0.04	0.03	0.01	0.12	0.13
Weight	40	12.05	4.53	3.04	90.91	12.05	27.03	40	76.92	8.33	7.87
Occurrence in FF3	574	264	317	*40*	559	264	298	574	589	697	687
Average in FF3	0.57	0.26	0.32	0.04	0.56	0.26	0.30	0.57	0.59	0.70	0.69

Table 17: Found patterns (rules) in FF1-FF3 together with the calculated weight for FF3 based on 1000 runs.

Pattern	Enemy					
Occurrence in FF1	2605	1198	572	2606	1208	525
Average in FF1	2.61	1.20	0.57	2.61	1.21	0.53
Occurrence in FF2	13751	10411	1897	13584	8678	722
Average in FF2	13.75	10.4	1.9	13.6	8.68	0.72
Weight	0.07	0.1	0.53	0.07	0.12	1.39
Occurrence in FF3	*444*	*50*	*8*	*444*	*33*	*16*
Average in FF3	0.44	0.05	0.01	0.44	0.03	0.02

Table 18: Found patterns (rules) in FF1-FF3 together with the calculated weight for FF3 based on 1000 runs.

Pattern	Hordes					Gaps			
Occurrence in FF1	920	931	1007	1007	892	111	286	269	286
Average in FF1	0.92	0.93	1.01	1.01	0.89	0.11	0.29	0.27	0.29
Occurrence in FF2	3694	4995	8209	8209	3563	14	83	68	132
Average in FF2	3.69	5	8.21	8.21	3.56	0.01	0.08	0.07	0.13
Weight	0.27	0.2	0.12	0.12	0.28	71.43	12.05	14.71	7.58
Occurrence in FF3	*0*	*90*	*93*	*93*	*0*	1720	*44*	*33*	*88*
Average in FF3	0.00	0.09	0.09	0.09	0.00	1.72	0.04	0.03	0.09

Table 19: Found patterns (rules) in FF1-FF3 together with the calculated weight for FF3 based on 1000 runs.

Pattern	Valley			Stair				
Occurrence in FF1	87	81	61	845	846	664	705	716
Average in FF1	0.09	0.08	0.06	0.85	0.85	0.66	0.71	0.72
Occurrence in FF2	17	14	17	355	352	257	289	287
Average in FF2	0.02	0.01	0.02	0.36	0.35	0.26	0.29	0.29
Weight	58.82	71.43	58.82	2.82	2.84	3.89	3.46	3.48
Occurence in FF3	193	178	162	1233	1197	1110	915	1025
Average in FF3	0.19	0.18	0.16	1.23	1.20	1.11	0.92	1.03

Table 20: Found patterns (rules) in FF1-FF3 together with the calculated weight for FF3 based on 1000 runs.

Pattern	Pipes					
Occurrence in FF1	66	47	43	46	61	67
Average in FF1	0.07	0.05	0.04	0.05	0.06	0.07
Occurrence in FF2	28	14	9	14	8	10
Average in FF2	0.03	0.01	0.01	0.01	0.01	0.01
Weight	35.71	71.43	111.1	71.43	125	100
Occurence in FF3	*5*	43	57	**12**	966	30
Average in FF3	0.01	0.04	0.06	0.01	0.97	0.03

multiple-occurrence starvation effect. The weights were calculated as the inverse function ($\frac{1}{x}$ when $x \neq 0$) of the average occurrence. The result for FF3 show positive effect for most of the meso-patterns (26 out of the 43 rules) except for the Gaps-, Enemy- and Horde-patterns for which the result, on the other hand, is absolute catastrophic (in tables 15-20 the negative change is indicated in italic).

9.4 EXPRESSIVE RANGE

Smith & Whitehead [202] introduced the concept of *expressive range* of a level generator and suggested a set of possible metrics that illustrates diversity of the generated content. For PCG-tools it is interesting to show if the tool is able to generate content that is not identical. *Linearity* and *Leniency* were suggested as metrics for platform levels.

We have implemented versions of these metrics thus: Leniency is calculated across the whole level with $+1$ for gaps and enemies, and the reverse for the opposite -1 (for jumps with no gap associated, because jumps associated with danger is harder than jumps without danger). Linearity will be counted from the lowest point of the level, due to the fact that most mi-

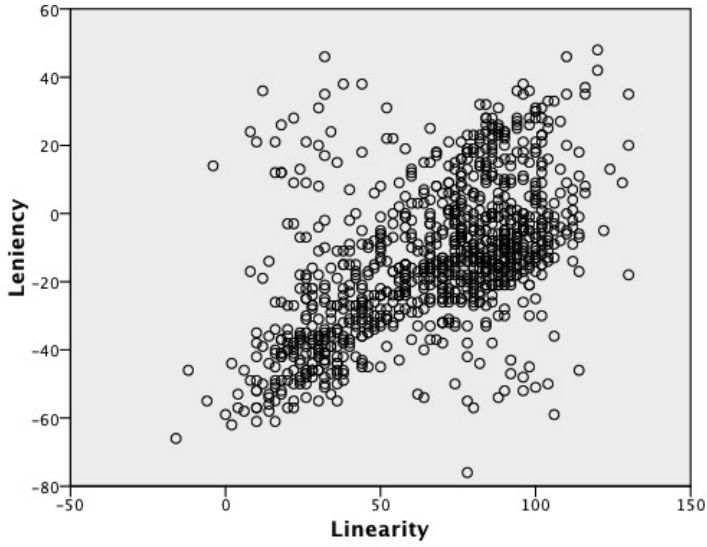


Figure 35: The distribution of levels generated with FF1 on the two expressivity dimensions.

cro patterns are connected to that and therefore all micro patterns forcing the player to jump due to a height difference of more than 1 tile will be considered as raising the non-linearity of the level.

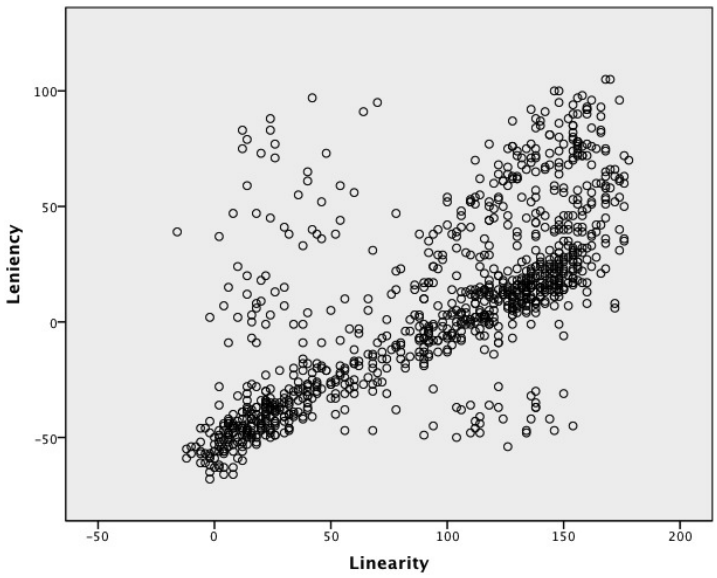


Figure 36: The distribution of levels generated with FF2 on the two expressivity dimensions.

In figure 35, 36 and 37 we show a density plot based on the two metrics; leniency (LEN) and linearity (LIN) with 1000 generated levels for the fitness functions 1, 2 respectively 3 (FF1-3). FF1 have an expressive range in LEN of -75 to $+50$ with a concentration of levels around -20 to ± 0 as well as an expressive range in LIN of -20 to $+130$ with a concentration in the range $+50$ to $+100$. FF2 gives LEN: -75 to $+100$ and LIN: -20 to $+170$. FF2 has two clusters; LEN/LIN -75 to $-25/\pm 0$ to $+50$ and -25 to $30/+85$ to 160. Comparing the two fitness functions (FF1 & FF2) expressiveness yields that FF2 can generate both more difficult and more linear levels. The correlation that may exist is due to the gap and enemy placement in linear space in SMB (and in the micro-patterns) and it is more apparent due to the higher

alignment to meso-patterns in FF2 than in FF1, which is more affected by the normal distribution in the variation of micro-patterns and get a less apparent cluster and range. FF3, however differ on all ranges; LEN: −105

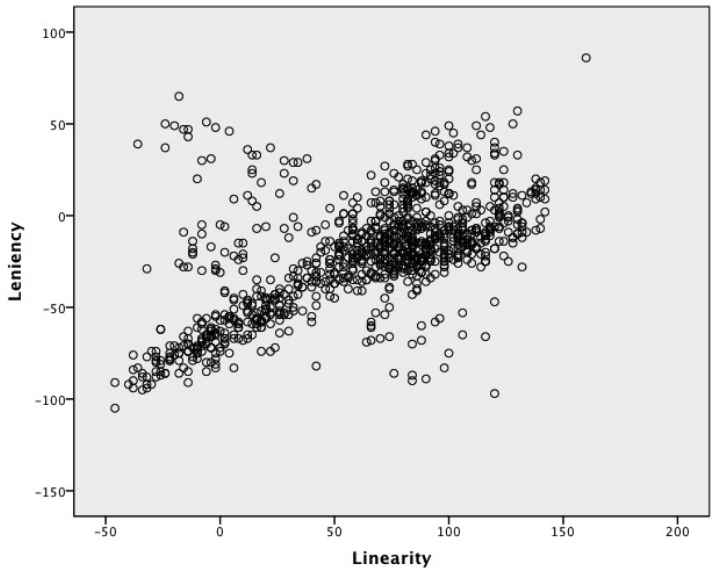


Figure 37: The distribution of levels generated with FF3 on the two expressivity dimensions.

to +80 & LIN −50 to +160. The two clusters; LEN/LIN: −100 to −30/ − 25 to +25 and −30 to +20/ + 50 to 130, are less apparent divided from each other and most of the individual members are not spread out as thin as before. The weighted fitness value gives a wider expressive range but the levels are more close if we observe the outliers suggesting that we could say that the *expressive spread* is affected with weighted patterns. The levels are more easy but also less linear. This is no surprise due to the low presence of meso-patterns of Gap-, Enemy- and Horde-type.

Figure 38: An example of a generated level.

9.5 DISCUSSION

Our approach could be viewed from a level designer's standpoint if we see the design process as handled by our three pattern levels; 1) at the micro-level, which contain the smallest representation level, in our approach the vertical slices function, 2) at the meso-level, where the combined slices in a certain order function to solve the challenges the designer wants to expose to the players to, and 3) at the macro-level handling the flow and overall (play-)experience of a level and/or game. If we implemented a planner that solved the issue of deciding on order of meso-patterns, difficulty (perhaps with the aid of metrics like leniency), training and educating the player, the full task of the level-designer, namely; to "... use a toolkit or 'level editor' to develop new missions, scenarios, or quests for the players. They lay out the components that appear on the level or map and work closely with the game designer to make these fit into the overall theme of the game" [78], could be solved for an entire game or genre.

In our fitness functions FF2 and FF3, we used weighted sums of the meso-pattern counters. There are well-known problems with fitness functions based on weighted sums, in particular that not all components are maximised at the same rate. An alternative would be to treat the problem as a multi-objective optimisation problem, and use specially designed evolutionary algorithms for this purpose. However, most such algorithms are designed for only a handful of objectives, which is problematic as our problem has dozens.

9.6 CONCLUSION

In this paper, we have introduced a pattern-based level generator for platform games. The general principle is to identify both micro-patterns and meso-patterns in the original game levels, represent new levels as combinations of micro-patterns and search for such combinations that express as many meso-patterns as possible. This way, micro-patterns are used as building blocks and meso-patterns as objectives. This principle, and the generator based on it, can easily be extended to a large range of different game types and game content types. To validate and explore the workings of our prototype level generator, we ran experiments with three different variations of our fitness function. We found that the generator could easily find certain patterns whereas others where harder to find, but that a rebalancing made it possible to find other patterns, sometimes at the cost of more frequent patterns.

9.7 ACKNOWLEDGMENTS

We would like to thank Noor Shaker for the generated level image.

NOTES

[22]The benchmark is based on the clone *Infinite Mario Bros* by Markus "Notch" Persson.

10

PAPER 4 – A MULTI-LEVEL LEVEL GENERATOR

ABSTRACT

Generating content at multiple levels of abstraction simultaneously is an open challenge in procedural content generation. Representing and automatically replicating the style of a human designer is another. This paper addresses both of these challenges through extending a previously devised methodology for pattern-based level generation. This method builds on an analysis of Super Mario Bros levels into three abstraction levels: micro-, meso- and macro-patterns. Micro-patterns are then used as building blocks in a search-based PCG approach that searches for macro-patterns, which are defined as combinations of meso-patterns. Results show that we can successfully generate levels that replicate the macro-patterns of selected input levels, and we argue that this constitutes an approach to automatically analysing and replicating style in level design.

PUBLISHED IN

Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games

ISSN: 2325-4270 doi: 10.1109/CIG.2014.6932909

A MULTI-LEVEL LEVEL GENERATOR

10.1 INTRODUCTION

Procedural content generation in games (PCG) refers to the algorithmic creation of game content, with no or limited human input. Recent years has seen a marked increase in interest in PCG in the game development community, where it is now routinely used both for runtime level generation in certain types of games (e.g. rogue-likes) and for offline generation of certain types of content, such as vegetation and terrains. This development is paralleled by a significant increase in PCG research in academia. Unlike in commercial game development, the focus tends to be on more ambitious forms of PCG than what is currently seen in released games, and using more complex methods [231].

In a recent overview paper, a number of long-term goals and research challenges for PCG are described [229]. The paper suggests the following grand goals of PCG: *Multi-level Multi-content PCG*, *PCG-based Game Design* and *Generating Complete Games*. It is argued that work addressing any of its nine more concrete research challenges would contribute to progress towards realising these grand goals of PCG. Further, five very concrete actionable steps are listed, each of which is envisioned to address one or several of the research challenges.

In this paper, we address two of the research challenges, namely *Representing Style* and *General Content Generators*, and one of the actionable items, namely *Competent Mario Levels*. Representing Style refers to being able to create a generative model of the style of a particular designer or a particular design school, whereas General Content Generators refers to being able to generate either different types of content (on different levels of abstraction) for a single game or content for multiple games. The Competent Mario Levels actionable step refers to creating level generators for Super Mario Bros

that can create varied, interesting, good-looking, playable and entertaining levels.

The way we address these challenges is to extend an existing pattern-based level generator for Super Mario Bros. In previous work, we have described a method which builds levels for Super Mario Bros out of "micro-patterns", i.e. thin level slices, and uses an evolutionary algorithm to search for levels that contain multiple instances of "meso-patterns", which are larger designed structures [48, 49, 51]. It was observed that while this method generated playable levels with interesting micro-structure, the levels lacked a sense of progression, unity or other macroscopic properties. The working hypothesis of this paper is that such macroscopic structure can be achieved with an extension of this method by using objectives at a higher abstraction level. This in turn requires that such objectives can be extracted from existing game levels.

10.1.1 *Contributions in this paper*

In previous work, we have identified meso-patterns in Super Mario Bros [48], and devised a search-based approach to level generation in the Mario AI Benchmark where micro-patterns are used as building blocks and meso-patterns as objectives [49, 51]. In this paper, we introduce a third level of abstraction, macro-patterns, defined as the occurrence and sequence of meso-patterns. We also describe a level analyser, which extracts patterns from existing levels. Finally, we describe the results of experiments in evolving levels using macro-patterns as objectives. For this purpose we have also devised a new mutation operator for game levels based on cutting and pasting micro-patterns.

10.2 BACKGROUND

Our work builds on previous work in both design-oriented and technical game research. Here, we describe previous work on PCG in games, design

patterns, and the combinations of these, and we also present the benchmark game used for the experiments.

10.2.1 *Procedural content generation in games*

Game content refers to any game asset excluding non-player characters, (NPCs) behaviour and the game engine - for example levels, rules, textures, narrative and in-game items. PCG has recently attracted considerable interest in the digital game research community, as evidenced by hundreds of publications and the establishment of a dedicated workshop running annually since 2010. This is at least partly due to there being multiple good reasons to attempt to create algorithms that generate content, including: reducing the cost and time of game development, enabling infinite and/or adaptive games, studying game design by formalising human creativity, and attempting to surpass such creativity. In the current context, we are interested both in the computational study of game design, and in creating fast algorithms that can reliably supply a game with large amounts of quality content.

The last few years has seen a surge of interest in an approach to PCG called search-based PCG, where evolutionary algorithms or other global stochastic optimisation algorithms are used to generate content [228]. The two most important considerations here are content representation (how the genotype, e.g. levels, is represented as a phenotype, e.g. vectors of integers, on which the variation operators work) and content evaluation (how a fitness value is assigned to a content artefact).

10.2.2 *Design Patterns*

Design patterns were initially proposed by Alexander [7], an architect who created them with the intent to empower individuals to express their ability to design. Design patterns are basically a rather informal grammar containing a set of descriptions covering reoccurring design problems in a domain. This problem description is paired with a suggested core solution which

could be reused. In effect, the second of the two components (problem & solution) is very versatile due to the generalisation of the solution space. Design patterns have been adopted in object-oriented analysis and design [82], and thinking of software architecture in terms of design pattern solutions has become very influential. Björk and Holopainen later applied the ideas of design pattern to game design, listing hundreds of generic game design patterns in an influential book [30, 29]. Several authors have further identified a number of game design patterns in specific game genres [102, 41, 126, 64].

In the current paper we are principally interested in patterns in *level design*, where levels are the structures that the player character traverses (not to be confused with e.g. levels of abstraction). Given the fact that games often are designed artefacts put together with a purpose, several aspects of them can be viewed as structures. For instance, rules govern the process of play, whereas levels and game space is often indirectly controlling the movement of the player, and objects in games usually have a specific purpose effectively limiting their use for the player. In relation to this, we could view the content in a platform game (including levels) as structured design objects, i.e. objects following design patterns.

10.2.3 *Benchmark game*

In this paper we will use the game *Super Mario Bros.* (SMB) [158] as a benchmark. The game was first released by Nintendo in 1985, and is a side scrolling 2-dimensional "platformer" game. SMB has become very influential through setting a number of standards for the platformer genre, and has helped bring about the genre's popularity. In the game, the protagonist Mario (or his brother Luigi) moves from left to right, jumping onto platforms or other structures to overcome obstacles or onto enemies to squash them. SMB consists of 8 worlds, each containing 4 levels, where the three first levels span from a starting point (left-most) to the end by a castle entrance (right-most) and the fourth level ends in a "boss-fight". As there is no interface for NPC control or level generation in the original game, we build on the *Mario AI Framework*, a software toolkit which was developed

for the Mario AI Competition [223, 190, 113]. This software is based on *Infinite Mario Bros*, a clone of SMB that focused on the non-"boss-fight"-levels. In SMB the levels have a varied length of 148 to 377 with an average of 200 tiles. Various approaches to generate levels for the Mario AI Framework have been proposed, as surveyed in [190, 101]; approaches that explicitly copy the style of SMB levels include Markov chains [211].

10.3 LEVEL DESIGN PATTERNS IN MARIO

We have previously analysed the content of the original game with the aid of a framework for 2D Platformer games [203] and heuristics for playability [55] and suggested a set of (meso-) patterns that SMB levels consists of [48]. We identified patterns on two levels, micro-patterns and meso-patterns. Micro-patterns are simply vertical slices of the level. Meso-patterns are features such as groups of *enemies*, *gaps* to jump over, *valleys* boxing in parts of the level, allowing the player to choose *multiple paths* and elevating Mario with the aid of *stairs*. In this paper, we also introduce macro-patterns, which are sequences of meso-patterns.

10.3.1 *Micro-patterns*

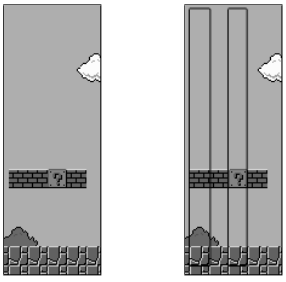


Figure 39: To the left we have a excerpt from SMB World 1–Level 1 which can be replicated with only two micro-patterns (slices) marked with black frames to the right. It also exemplifies a 2-Path pattern.

The content in SMB can be viewed from Mario's standpoint, namely, horizontally from left to right one tile at the moment. If one imagine the levels as one tile wide slices and collect them in a library, the first level, even though it is 199 slices "long" only 27 different slices are used. These slices could be viewed as micro-patterns since they, in themselves, are designed content, and they often contain several pieces put together like a Goomba, a question-mark-box, a brick-block or something else that is either an obstacle, or aid to the player. In fig 39, the left-most slice or micro-pattern contain mostly empty space (allowing Mario to jump) but also to land on a Goomba or a ground-tile. If Mario were to walk into this slice the player either loses a life or a power-up effect. These micro-patterns works in a similar way as the tiles when decomposing the problem of generating dungeons [141].

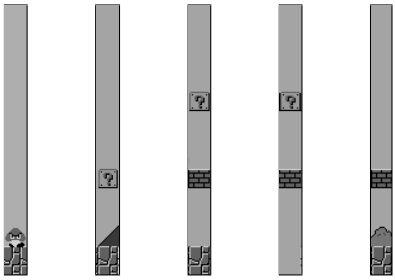


Figure 40: Examples of similar but still unique slices. The two to the right can be used to create the structure of Fig. 39 and a sub-set of them can be used to create most of Fig. 42

10.3.2 *Meso-patterns*

The next meaningful level are the meso-patterns. These are perhaps best explained as instances of the different patterns previously identified [48]. A meso-pattern is a feature which requires or affords some player action, like three Goombas walking in a single file formation on ground. If the micro-patterns are to be seen from Mario's viewpoint, the meso-patterns could

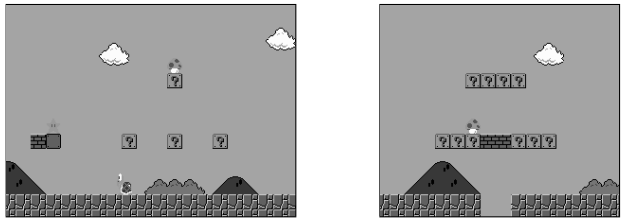


Figure 41: Two meso-patterns (to the left; a sparse *Risk and Reward* (W1L1) and to the right a dense *3-Path* (W4L1).

be viewed more like how the player sees the content: sequences of slices making up most of a screen.

It is important to note that meso-patterns are a bit more abstract than micro-patterns; each meso-pattern could be instantiated in multiple ways, by different configurations of micro-patterns. For example, a valley could consist of 5 or 10 slices, but still be a valley. If our micro-patterns are identified by an integer then a meso-pattern is a string of specific integers like $5-1-1-7$ or $1-8-8-5-8-1$[23]. (These particular strings are taken from the original game.)

10.3.3 *Macro-patterns*

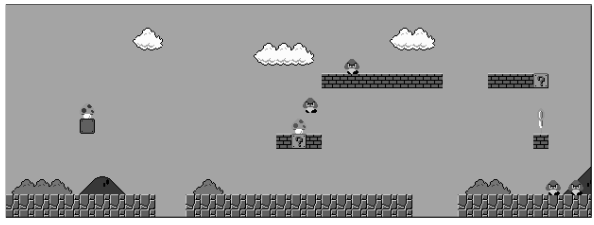


Figure 42: A Macro-pattern example from SMB, stretching over two screens, where a *2-Path* and a *Gap* continues on to a *Risk and Reward* and a *Gap* onto a *3-Path* with an end consisting of a *2-Horde*.

The meso-patterns are helpful to understand the content of SMB but does not convey more macroscopic level structure. For that we suggest a higher level – the macro-pattern level. On this level the relation between different meso-patterns becomes clear and the placement of individual power-up-mushrooms can balance difficulties that lies beyond the current screen. In figure 42 we can see an example of how patterns are connected together over more than one screen. At this level of abstraction the level designer can provide the player with a greater play experience by providing a steady and controlled difficulty curve, teach the player how to tackle new obstacles and enemies. "Pedagogic" macro-patterns, where a meso-pattern first appears in a simpler form and then in a more complex form, so that the player can first overcome the simpler challenge to then be ready to face a harder challenge of the same type, are common among the original SMB levels. Story arcs could be partly implemented, or at least supported, on the macro-level as well. Given this, it might be possible to argue for further abstraction levels covering the "Worlds" of SMB or perhaps the full game or even the whole game franchise.

For example, if we describe the first level of SMB, i.e. World 1–Level 1 (W1L1, see Fig. 43) as a sequence of meso-patterns we get the following: *Risk and Reward*, *(Empty) Pipe valley*, *(2-Horde) Pipe Valley*, *2-Path*, *Gap*, *Risk and Reward*, *2-Horde*, *Risk and Reward*, *Risk and Reward*, *(Empty) Stair valley*, *(Gap) Stair valley*, *Roof valley*, *Stair up*.

Figure 43: Level 1, World 1 from the original Super Mario Bros game, reimplemented in the Mario AI Framework (SMB-W1–L1).

Figure 44: Level 1, World 8 (SMB-W8–L1) (mid 200 tiles, start and ending empty ground is cropped).

10.3.4 *Multi-Level Level Generation*

In our suggested approach we utilise a "bottom-up"-approach where the micro-level is the foundations for the meso-level which in turn makes up the macro-level. Our method is a *search-based* PCG approach [228], described below.

10.4 PATTERN-BASED LEVEL GENERATION

We have previously presented two level generators for the Mario AI Framework that builds on the identified patterns. The first of these was a simple constructive pattern-based level generator that combined pre-fabricated instances with minor variations depending on assigned parameters on difficulty and reward settings [48]. The second generator takes a search-based approach, with a representation based on micro-patterns and objective function based on the existence and number of meso-patterns [49]. Two versions of the fitness function were developed: one which simply counted every occurrence of every meso-pattern, and one which only counted the number of individual meso-patterns that could be found in the level. It was found that levels that scored highly on either of these metrics were perceived as better-designed than those that scored lower, but also that those that were only optimised for the first variation (every occurrence) became rather dense. After further experimenting [51] with its multi-objective fitness functions, we here extend it to cover the macro-pattern level as well.

10.5 AUTOMATIC LEVEL ANALYSIS

In order to be able to generate levels that replicate the sequence of meso-patterns from existing levels, we first need to be able to extract this sequence. For this purpose we built a level analyser. The level analyser takes any Super Mario Bros (or Infinite Mario Bros) level encoded in a specific simple file format and returns a list of all the micro-patterns (slices) in the level and their frequencies, and the order of all meso-patterns. This is technically

an array of integers where each integer represents a particular meso-pattern out of those identified in [48], but can be read out as e.g. "{pipe-valley, three-horde, three-horde, stair}" etc. The same pattern detection code is used here as is used in the objective functions.

10.6 METHODS

In this section we stepwise go through our approach, by stating the principal parts; representation, algorithm and fitness function.

10.6.1 *Representation*

Our level generator output is a single SMB level with the length of 200 and a height of 14 tiles. The internal representation of a level is an array of integers, where each integer represents a micro-pattern (see Fig. 40 for examples).

10.6.2 *Evolutionary Algorithm*

Our search-based approach uses a fitness function that rewards the presence of meso-patterns with a simple $\mu + \lambda$ evolution strategy where $\mu = \lambda = 100$ combined with the operators *single-point mutation* and *one-point crossover*. This means, when we use a population of 200 members, that we discard the 100 members with lowest fitness and use the best 100 members as parents for breeding pairwise. All of the newly generated offspring are also subject to mutation. We consequently deem members with unplayable content as unfit for breeding by setting their fitness value extremely low.

10.6.3 *Variation operators*

In previous work our mutation operator simply exchanged a single micro-pattern for another, randomly selected [51]. Given the relative length of a

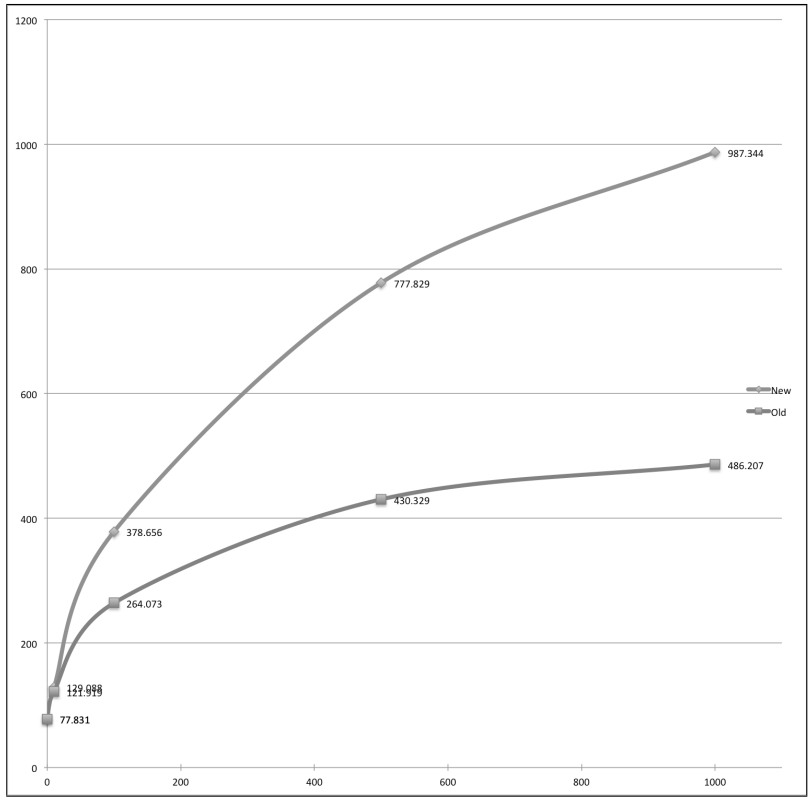


Figure 45: A comparison between the effect of the mutation-operators.

micro-pattern (1 slice = 1 block), in relation to a full level in SMB (148-377 slices) and the nature of our initial mutation operator; exchanging a single slice for another for the whole member (meaning a mutation effect of 0.5% out of 200 slices) we opted to incorporate a more aggressive mutation (blue line in Fig. 45). Instead of the minimalistic mutation operator working as an exchange of a single slice exchange (red line) we apply a sequence exchange. The new mutation operator change a set of five slices at a random starting position with a new random set of five slices.

10.6.4 *Fitness functions*

Our fitness functions measure the presence and order of patterns and are based on string search. A fitness value is assigned to each level based on the presence of specific sub-strings representing meso-patterns taken from SMB. A sub-string is typically seen in Fig. 39 made up with micro-patterns (see Fig. 42). Since these sub-strings vary in length and complexity some patterns are harder to find in the solution space than others. This, in turn, yields that we need to understand how to define the fitness function according to the wanted outcome. We focus our attention on the difference between finding all patterns we have defined in solution space (with the fitness function *FFMeso*) without rewarding any specific pattern over another. From there we utilise a weighted value based on previous experiments [51] (called *FFMesoB[alanced]*) in order to understand the effect of the added macro-level works in the solution space (called *FFMacro*). *FFMacro* is based on a relative reward value so that it rewards the correct order of meso-patterns according to the original SMB meso-pattern order in addition to how FFMesoB reward sub-strings. In short, if the order of meso-patterns in a member corresponds to the one in the target level it is more probable that it is chosen for breeding. Note that we are only looking for instances of meso-patterns and not the exactly same pattern implementations as in SMB.

10.7 RESULTS

Our experiments are evaluated in three ways: (1) We measure the meso-pattern (type and how many) for the best member of a 1000 generation search (200 members with the length of 200 micro-patterns); (2) we compare the fitness values distribution for the fitness functions; and (3) we apply *expressive range* analysis (see section 10.7.2).

In order to get some input on diversity aspects of the different fitness functions we have generated 100 levels for each fitness function and compare them to each other. FFMeso favours simple patterns like enemies and

hordes and seldom provide anything more complex like multi-way and pipes. FFMesoB and FFMacro provide better overall coverage of patterns. FFMesoB and FFMacro does not differentiate very much but generally FFMacro provide some improvements on longer patterns (indicated in italic in tables 23–28).

FFMeso generally perform uniform values. It should be noted that since this fitness value favours low ranked rewards and is then compared to the weighted macro fitness function very little variation is gained (see Tables 23–28 to see the pattern distribution) it should not be directly compared value by value with the other two fitness functions which are more compatible in regard to comparison. In that aspect both fitness functions can generate macro-pattern ordered in a level but FFMacro perform a bit better reaching a macro-pattern fulfilment of a maximum 7/12 and a common level of 4/12 whereas FFMesoB only reaches 6/12 (see Table 29). FFMesoB has a higher maximum altogether because it can fit in more high value patterns in the level. FFMacro tries to find the right pattern there which could be improved by rewarding macro pattern order more, but of course then running the risk of starving the meso-patterns altogether.

Figures 49a show a number of generated levels for visual comparison. These were all generated with level 1-1 (as seen in figure 43) as target level. It can be seen from these pictures that the levels generated with the Macro fitness function appear to have more large-scale structure, or at least more variation on the macro scale.

10.7.1 *Efficiency*

FFMeso and FFMesoB can run in fair online environments generating a level with the length of 200 based on a 200 member population and 1000 generations in 4 seconds with the current implementation in Java running in NetBeans IDE on a 2011 MacBook Pro. However, the FFMacro, have to account for relative reward values and an extra data structure (that keeps track of the order in relation to the wanted order) which affects execution

time tenfold effectively placing this approach in the offline PCG application range.

10.7.2 *Expressive Range*

The concept of *expressive range* could be seen as the approach to visualise and measure the variation of the generated content according to a representative metric [202, 101]. This would allow understanding the diversity and uniqueness of a level generator. In our case we will apply Smith's & Whitehead's metrics *Linearity* and *Leniency* [202].

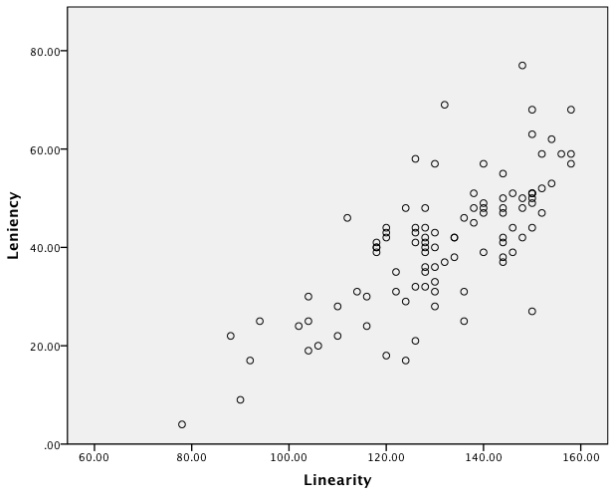


Figure 46: The distribution of levels generated with FFMeso on the two expressivity dimensions.

Our implementation works as follows; Leniency is calculated across the whole level with $+1$ for gaps and enemies, and -1 for jumps without danger. Linearity will be counted as $+1$ for any change from the floor of the level, due to the fact that most micro patterns is connected to that. In Fig. 46 the output of FFMeso and in Fig. 47 the output of FFMacro are displayed using 100 unique levels from the two different fitness function used.

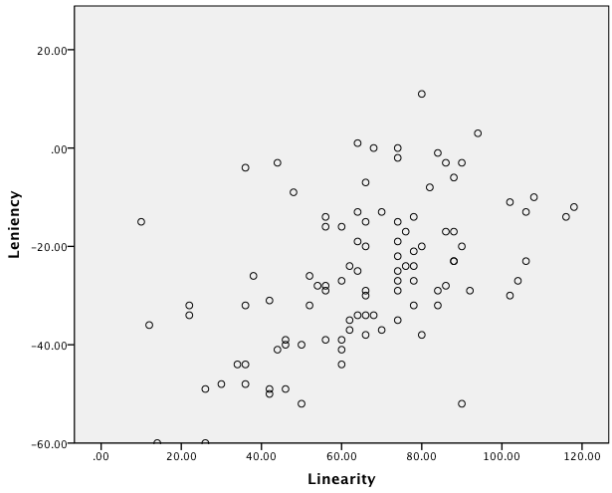


Figure 47: The distribution of levels generated with FFMacro on the two expressivity dimensions.

Comparing the FFMacro and FFMeso we can see that they occupy a different expressive space with FFMeso generating levels more similar internally than the other two fitness functions. FFMeso has a linearity range of 80 and leniency range of 80 whereas FFMacro has 75 and 105 for linearity and leniency respectively. Comparing their (FFMeso and FFMacro) individual space we can see there is very little overlap in their expressive range.

Given this (see Fig. 48) we can conclude that the *Linearity* of the FFMesoB and FFMacro are more variated but also less hard to complete (probably due to the lower number of enemies present in these levels see tables 23–28). However, since the distribution of different patterns are more like the original game SMB the FFMesoB and FFMacro are probably more interesting for a player.

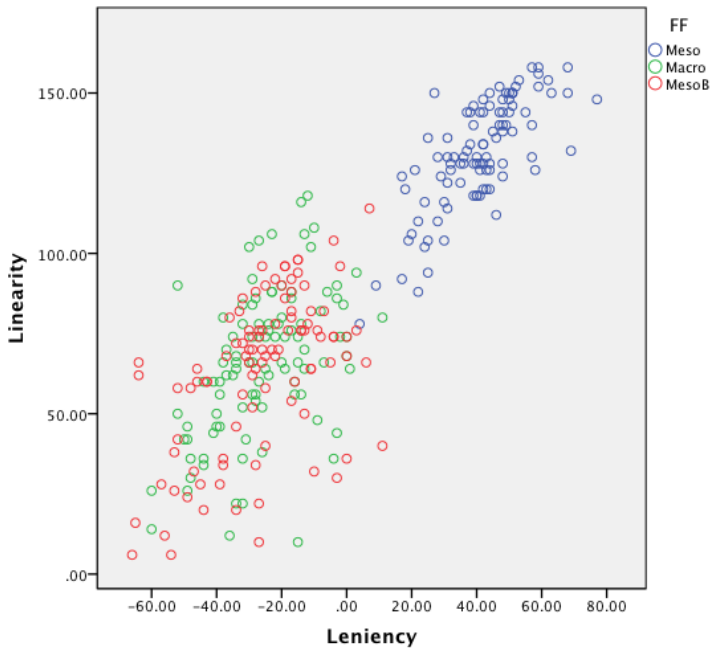


Figure 48: The distribution of levels generated with FFMeso, FFMesoB and FFMacro on the two expressivity dimensions.

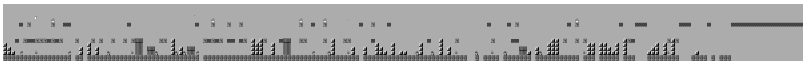
10.8 FUTURE WORK

Given the set of generators available to the PCG-research-community more in dept studies of the diversity of the different approaches would yield welcome knowledge. For instance, how well does the generator fulfil the intended goal and how does that relate to other generators abilities. Can we, with the use of metrics or empirical tests order the different generators on a spectra ranging from variation to control? Does implementation of different but similar techniques place generators close to each other on that spectra?

Considering the multi-level search-based and bottom-up-approach applied in this paper it would be interesting to compare it with other possible approaches for multi-level generators. Especially, top-down and constructive approaches would be a welcome comparison. The top-down approach could function in different ways, ranging from a more automated version were the user supplied parameters and the generator suggested levels to a more user centred approach where the designer marked our space in a level and picked pattern definitions and placed them in an order suited to the designer and mixing designer work with constraints on the generator to fulfilling patterns and even down to level where the designer defines new meso- and micro-patterns.

10.9 CONCLUSION

In this paper we have suggested a search-based PCG method and level generator for platform games that incorporates three levels of patterns, namely; 1) micro-, 2) meso- and 3) macro-patterns. These three levels handles different aspects of the level generation ranging from low level detail to full level overview. To demonstrate the effect the multi-level level generator we ran a set of experiments with three different fitness functions; FFMeso (rewarding meso-patterns), FFMesoB (a balanced version using weights derived from a previous version [51]) and FFMacro (using the same weights but with an added extra reward if the order of the patterns we in alignment to the original SMB game). During this exploration of the solution space we noted that some patterns are affecting the presence of other patterns and that the expressive range can vary based on the used fitness function. The added macro-level have increased the run-time of the level generator tenfold making the generator more suitable for offline generation rather than online.



(a) FFMacro #26 MC: 4, fitness value: 441 (lowest).

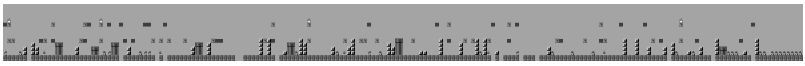
(b) FFMacro #28 MC: 6, fitness value: 1315.

(c) FFMacro #35 MC: 0, fitness value: 850.

(d) FFMacro #82 MC: 7, fitness value: 1485.

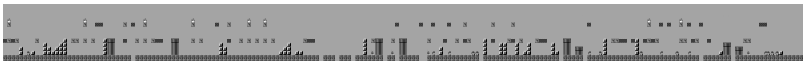
(e) FFMacro #98 MC: 6, fitness value: 2332 (highest).

Figure 49: FFMacro levels.



(a) FFMesoB #6 MC: 0, fitness value: 545.

(b) FFMesoB #30 MC: 3, fitness value: 409 (lowest).



(c) FFMesoB #64 MC: 6, fitness value: 2065 (highest).

Figure 50: FFMesoB levels.

(a) FFMeso #42 MC: 2, fitness value: 202 (highest).

(b) FFMeso #99 MC: 0, fitness value: -47 (lowest).

Figure 51: FFMeso levels.

NOTES

[23]The last string is for instance seen in Fig. 39

Table 21: Patterns for Super Mario Bros. grouped by theme part 1 [48].

Enemies	
Enemy	A single enemy
2-Horde	Two enemies together
3-Horde	Three enemies together
4-Horde	Four enemies together
Roof	Enemies underneath a hanging platform making Mario bounce in the ceiling
Gaps	
Gaps	Single gap in the ground/platform
Multiple gaps	More than one gap with fixed platforms in between
Variable gaps	Gap and platform width is variable
Gap enemy	Enemies in the air above gaps
Pillar gap	Pillar (pipes or blocks) are placed on platforms between gaps
Valleys	
Valley	A valley created by using vertically stacked blocks or pipes but without Piranha plant(s)
Pipe valley	A valley with pipes and Piranha plant(s)
Empty valley	A valley without enemies
Enemy valley	A valley with enemies
Roof valley	A valley with enemies and a roof making Mario bounce in the ceiling

Table 22: Patterns for Super Mario Bros. grouped by theme part 2 [48].

Multiple paths	
2-Path	A hanging platform allowing Mario to choose different paths
3-Path	2 hanging platforms allowing Mario to choose different paths
Risk and Reward	A multiple path where one path have a reward and a gap or enemy making it risky to go for the reward
Stairs	
Stair up	A stair going up
Stair down	A stair going down
Empty stair valley	A valley between a stair up and a stair down without enemies
Enemy stair valley	A valley between a stair up and a stair down with enemies
Gap stair valley	A valley between a stair up and a stair down with gap in the middle

Table 23: Found patterns (rules) in FFMeso, FFMesoB and FFMacro based on 100 levels and 1000 generations per level.

Pattern	Mesa		Straight
OCC. in FFMeso	16	11	132
AVG. in FFMeso	0.16	0.11	1.32
OCC. in FFMesoB	50	50	61
AVG. in FFMesoB	0.5	0.5	0.61
OCC. in FFMacro	28	55	57
AVG. in FFMacro	.28	0.55	0.57

Table 24: Found patterns (rules) in FFMeso, FFMesoB and FFMacro based on 100 levels and 1000 generations per level.

Pattern	Multi-way										
OCC. in FFMeso	4	3	3	1	0	3	2	4	1	2	1
AVG. in FFMeso	0.04	0.03	0.03	0.01	0	0.03	0.02	0.04	0.01	0.02	0.01
OCC. in FFMesoB	131	39	7	2	90	39	39	131	141	51	53
AVG. in FFMesoB	1.31	0.39	0.07	0.02	0.9	0.39	0.39	1.31	1.41	0.51	0.53
OCC. in FFMacro	137	39	8	3	90	39	38	137	143	62	62
AVG. in FFMacro	1.37	0.39	0.08	0.03	0.90	0.39	0.38	1.37	1.43	0.62	0.62

Table 25: Found patterns (rules) in FFMeso, FFMesoB and FFMacro based on 100 levels and 1000 generations per level.

Pattern	Enemy					
OCC. in FFMeso	2129	1510	221	2354	1310	97
AVG. in FFMeso	21.29	15.10	2.21	23.54	13.10	0.97
OCC. in FFMesoB	34	3	1	29	3	1
AVG. in FFMesoB	0.34	0.03	0.01	0.29	0.03	0.01
OCC. in FFMacro	32	2	0	40	5	2
AVG. in FFMacro	0.32	0.02	0	0.4	0.05	0.02

Table 26: Found patterns (rules) in FFMeso, FFMesoB and FFMacro based on 100 levels and 1000 generations per level.

Pattern	Hordes					Gaps			
OCC. in FFMeso	553	753	1380	1380	545	4	6	2	11
AVG. in FFMeso	5.53	7.53	13.80	13.80	5.45	0.04	0.06	0.02	0.11
OCC. in FFMesoB	0	1	0	0	0	*112*	22	11	25
AVG. in FFMesoB	0	0.01	0	0	0	0.112	0.22	0.11	0.25
OCC. in FFMacro	*0*	*3*	*4*	*4*	*0*	103	21	*17*	24
AVG. in FFMacro	0	0.03	0.04	0.04	0	1.03	0.21	0.17	0.24

Table 27: Found patterns (rules) in FFMeso, FFMesoB and FFMacro based on 100 levels and 1000 generations per level.

Pattern	Valley			Stair				
OCC. in FFMeso	0	0	1	18	21	18	10	18
AVG. in FFMeso	0	0	0.01	0.18	0.21	0.18	0.1	0.18
OCC. in FFMesoB	35	72	53	97	94	133	139	134
AVG. in FFMesoB	0.35	0.72	0.53	0.97	0.94	1.33	1.39	1.34
OCC. in FFMacro	38	90	86	79	89	129	102	90
AVG. in FFMacro	0.38	0.9	0.86	0.79	0.89	1.29	1.02	0.9

Table 28: Found patterns (rules) in FFMeso, FFMesoB and FFMacro based on 100 levels and 1000 generations per level.

Pattern	Pipes					
OCC. in FFMeso	0	0	0	0	0	1
AVG. in FFMeso	0	0	0	0	0	0.01
OCC. in FFMesoB	5	20	39	19	231	111
AVG. in FFMesoB	0.05	0.2	0.39	0.19	2.31	1.11
OCC. in FFMacro	7	19	40	17	223	117
AVG. in FFMacro	0.07	0.19	0.4	0.17	2.23	1.17

Table 29: Comparison of found Macro patterns

	MIN	MAX	MEAN	DEV	#0	#1	#2	#3	#4	#5	#6	#7
FFMesoB	0	6	2.6	1.52	14	11	12	37	19	4	3	0
FFMacro	0	7	3.2	1.53	7	10	9	24	36	9	4	1

11

PAPER 5 – A COMPARATIVE EVALUATION OF PROCEDURAL LEVEL GENERATORS IN THE MARIO AI FRAMEWORK

Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith and Julian Togelius

ABSTRACT

Evaluation is an open problem in procedural content generation research. The field is now in a state where there is a glut of content generators, each serving different purposes and using a variety of techniques. It is difficult to understand, quantitatively or qualitatively, what makes one generator different from another in terms of its output. To remedy this, we have conducted a large-scale comparative evaluation of level generators for the Mario AI Benchmark, a research-friendly clone of the classic platform game Super Mario Bros. In all, we compare the output of seven different level generators from the literature, based on different algorithmic methods, plus the levels from the original Super Mario Bros game. To compare them, we have defined six expressivity metrics, of which two are novel contributions in this paper. These metrics are shown to provide interestingly different characterizations of the level generators. The results presented in this paper, and the accompanying source code, is meant to become a benchmark against which to test new level generators and expressivity metrics.

PUBLISHED IN

Proceedings of the 9th International Conference on the Foundations of Digital Games

A COMPARATIVE EVALUATION OF PROCEDURAL LEVEL GENERATORS IN THE MARIO AI FRAMEWORK

11.1 INTRODUCTION

Procedural Content Generation (PCG) research is concerned with creating methods for generating game content with limited human involvement, automatically or semi automatically [229, 193]. "Content" is a broad term that involves things such as items, quests, rules and textures, but one of the most commonly generated types of content is levels. Runtime level generation has existed in published games at least since *Rogue* [232], and is important for thriving game genres as different as roguelikes, endless runners (e.g. *Canabalt* [182]) and epic strategy games (e.g. *Civilization* [73]). In recent years, many academic researchers have been inspired to work on the problems of level generation, and the academic literature now contains dozens of papers on the topic. These papers are methodologically very diverse, including approaches using agents, grammars, constraint solving, cellular automata, evolutionary computation, exhaustive search, and answer set programming [228, 96].

A question that naturally comes to mind is how to choose which of these methods to use. Are some methods best for different purposes? Is one generator capable of creating different kinds of content than another? Different games, and even different stages or modes of the same game, pose different content generation problems. For some games, the connectivity and reachability of the levels might be difficult to attain and the most important problem, for others the lifelikeness of certain structures such as walls or vegetation might be most important, or the rhythm of the level, or the fine-tuned challenge of the level. On top of that, PCG solutions have numerous tradeoffs. For example, a common tradeoff is the speed of the solution versus the possibility to guarantee certain properties of the level (such as

reachability). The degree to which the character of the generated level can be controlled via parameters might also be in opposition to the diversity of the generated content. In order to obtain meaningful solutions to these questions and problems, a method must first be created to evaluate individual content generators and compare generators to each other.

The current state of evaluation for content generators is largely ad hoc. Some generators are evaluated implicitly, via an evaluation of the game that they are situated in. This form of evaluation is not helpful to understand the qualities of the generator itself, and how to compare it to other generators. Frequently, generators are evaluated via a small sample of their output being shown as representative of the generator's capabilities; this form of evaluation-by-example lacks rigor (how do we know the small sample is representative?) and cannot help with an understanding of the space of potential content that can be created, as well as any biases in the generator. (A critical reader of such papers tend to suspect that either the shown example is the one good-looking level generated by the generator after many attempts, or that all levels look pretty much like the shown example.) For the PCG research community, it is difficult to make progress without a thorough understanding of the strengths and weaknesses of current approaches, how they compare, and whether a new generator is capable of producing novel results.

In order to make informed decisions about which content generation method would be best suited for a particular type of content generation problem, we need a way to characterize the performance of the content generator in the context of game design concerns. A promising approach to this takes the form of a set of metrics that can be applied to the output of the generator, to characterize the generator's expressive range. This paper is meant to make progress towards a common framework for evaluating content generators through the presentation of several metrics, and evaluating these candidate metrics on a collection of content generators.

We focus on levels for platform games, and in particular we investigate level generators for the Mario AI Benchmark, based on *Infinite Mario Bros* [172], an open-source game inspired by the platform game *Super Mario*

Bros [158]. This is done partly because the Mario platforming games are archetypical, their design influencing countless other games. This means that the level generation problems posed by that game are likely to be similar to the level generation problems posed by many other platform games and related games. Another reason is the popularity of the Mario AI Benchmark among academic Game AI and PCG researchers, meaning that this is probably the game for which the largest number of level generators have been made. This paper compares level generators developed by ourselves and by other researchers, in particular the participants in the Level Generation Track of the Mario AI Competition.

The main contribution of this paper is a thorough evaluation and comparative study of several existing level generators that have never been compared before. Several of these metrics have been used before (though some have been newly ported to the Mario AI framework); however, we also provide some new metrics, in particular the pattern-based metrics, and levels which have not been analysed before, in particular the original SMB levels. The result of this work is a baseline against which researchers can compare both new metrics and new generators, including an ability to easily visualize the results. The source code for the metrics and the evaluated levels has been publicly released, to make this evaluation framework available for all interested parties.

11.2 RELATED WORK

There is a relatively large body of work aimed at understanding game design in general and level design in particular. For example, Koster's *Theory of Fun* [119] focuses on the progression of challenges and learnability of games, while others prefer to understand games as systems of interlocking feedback loops [5]. Recently there has been a push towards understanding and describing games using a pattern language. Björk and Holopainen [30] catalog recurring patterns that can be found across many games, while others have examined level design-specific patterns in domains such as first-person shooters [102], 2D platforming games [205, 49], and role-playing

games [144, 206]. While some of these patterns are largely intended as qualitative descriptions of game properties, others take the view that design patterns can be solutions to specific design problems.

Most of the aforementioned work in understanding game and level design is based in qualitative analysis and theories. Several of the metrics in this paper form a step towards building upon such theories, including the new design pattern metrics described in Section 11.3.2. Further operationalization of these theories to develop more sophisticated metrics is an interesting potential area for future work.

Recently, there has been some work in trying to automatically and quantitatively measure aspects of game quality. Within search-based procedural content generation there is a need for evaluation functions, and for this reason several researchers have tried to quantitatively capture what they deem to be crucial aspects of game quality. This includes Browne's various metrics for board games, such as drawishness, length, drama and outcome uncertainty [38], and Togelius and Schmidhuber's learning-based metric [221]. There has also been some work on trying to measure the quality of platform game metrics specifically. For example, Smith and Whitehead defined two key metrics—linearity and leniency—as well as introducing a method for visualizing the expressive range of a generator [209]. Shaker et al. followed up this work by introducing further metrics, some of them based on theories of player experience and others based on data mining [191].

Finally, more broadly than PCG for games, there is some work in evaluating computationally creative systems; Jordanous provides a survey of current evaluation methods [111]. It is important to note that these evaluation criteria are being used to answer a different, though related, question: computational creativity evaluation asks the extent to which a system is creative, while PCG evaluation asks how expressive and controllable the system is. For example, Pease et al. incorporate an evaluation of the *process* that the generative system follows as well as rating the *product* produced by the system [167]. Similar evaluations have been performed on platform game level generators [45]. While we recognize the importance of process in

understanding creativity, and feel that such discussions would be of great value to PCG researchers, it lies outside the scope of this paper.

11.3 EXPERIMENTAL TESTBED

For our experiments, we use the Mario AI Benchmark [230], built on top of *Infinite Mario Bros.* The world representation in this framework is not 100% faithful to the original *Super Mario Bros.*, and some of the graphical elements resemble elements from later games in the series. In particular, not all of the items and creatures found in *Super Mario Bros*, can be found in *Infinite Mario Bros*, but the missing features tend to be infrequently used in the original game.

11.3.1 *Generators*

In this section we describe the different generators that are compared in this paper. The generators were chosen because they have been the subject of academic papers or have taken part in the level generator track of the Mario AI Championship, and so as to maximize the number of different approaches to level generation represented.

The **Notch** generator is the default level generator that comes with *Infinite Mario Bros.* It writes levels from left to right, adding components according to probabilities. Basic checks are performed to make sure the levels are playable.

The **Parameterized Notch** generator is a version of the Notch generator that takes parameters, which bias how levels are generated. These parameters are the number of gaps, width of gaps, number of enemies, enemy placement, number of powerups and number of boxes. The test explores all possible combinations of high and low values for these parameters. See [185] for more information.

Hopper was written for the Level Generation track of the 2010 Mario AI Championship. Like *Notch* and *Parameterized Notch*, it generates levels through writing them from left to right and placing features with specific

probabilities. It was built with adaptability in mind, so that the probabilities could easily be altered depending on the player's prior performance. The generated parts are alternated with pre-designed parts. See [190] for more information.

Launchpad is a rhythm-based level generator that uses design grammars for creating levels that obey rhythmical constraints. The original version of Launchpad incorporated several level elements that are not present in the framework (e.g. springs); this ported version has attempted to remain as faithful as possible to the original grammar-based implementation, substituting level components as needed. See [209] for information on the original Launchpad.

The **Occupancy-Regulated Extension (ORE)** generator was also an entry for the Level Generation track of the 2010 Mario AI championship [190]. It works by piecing together small, hand-authored chunks of levels. Each chunk has an "anchor point" used to determine how the chunks can be pieced together. It can create quite complex levels that are stylistically quite different from the original Mario levels.

The **Pattern-based** generator uses evolutionary computation to generate levels. Levels are represented as sequences of "slices", or "micro-patterns" which are taken from the original Mario. Each micro-pattern is one block wide and has the same height as the level. The fitness function counts the number of occurrences of specified sections of slices, or "meso-patterns". The objective is to find levels with as many meso-patterns as possible. See [49] for more information.

The **Grammatical Evolution (GE)** generator uses evolutionary computation together with design grammars. Levels are represented as instructions for expanding design patterns, and the fitness function measures the number of items in the level and the number of conflicts between the placement of these items. See [191] for more information.

Finally, the **original levels** from *Super Mario Bros 1* [158] are included. They have been reproduced as faithfully as possible by manual translation from the ROM code of the original game. The exceptions are those elements which are not part of the design vocabulary of Infinite Mario Bros (and thus

of the Mario AI benchmark), and the water-based levels which cannot be simulated in the current version of the framework and for which completely different design principles are likely to hold. Levels were between 148 and 377 blocks in length, with an average length of 200 blocks.

11.3.2 *Metrics*

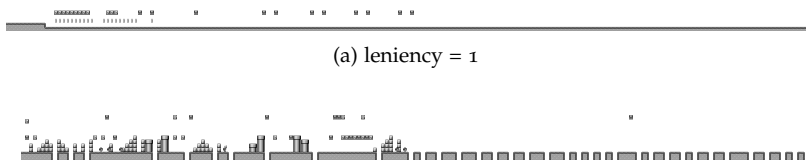
To compare the levels produced by our various levels, we have used a number of metrics, most of which come from previous literature but the two pattern-based metrics are introduced in this paper. The metrics are meant to capture relevant aspects of the levels including player experience (e.g. leniency) and level composition, but as there are many potentially relevant aspects, the current set of metrics should not be seen as exhaustive.

11.3.2.1 *Individual level metrics*

Most of our metrics work on a single level, and return a single real number as its evaluation of that level. All of our metrics are normalized by level length as appropriate, and are further normalized by the total of output of that metric.

The **leniency** metric is an attempt to capture how difficult a level is for a player. Leniency was calculated by finding all points in the level where an action by the player is needed, e.g. the edge of a platform or the end of a string of blocks, and then determining how lenient that particular challenge would be to the player. Gaps where a player dies are given a 0 weight for leniency and other enemies and jump lengths are weighted based on the challenge and death possibility given to a player. When a jump is detected, it looks ahead to see if any other obstacles would be in the way while landing. These other obstacles lower the leniency of the first challenge by a factor equal to their assumed harm level. Areas with no threat of harm are given a score of 1. Once all obstacle weights were calculated, they were then normalized based on the length of the level and how many possible paths were available at each point in the level. Two example levels from the parameterized notch randomized and the pattern-based weighted

count generators with very low and high leniency values are presented in Figure 52. Note that this description of leniency is different from those used in previous work by Smith et al. [209] and Shaker et al. [191].

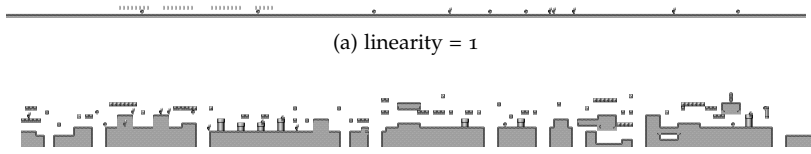


(a) leniency = 1

(b) leniency = 0.23

Figure 52: Example levels from (a) the parameterized notch randomized and (b) the pattern-based weighted count generators with very low and high leniency values.

The **linearity** metric is calculated by finding the R2 good-ness-of-fit measure for a line that has been fit to the end points for each platform in the level. This means that levels with many height differences will have low linearity, while levels that follow a straight line (flat or otherwise) will have maximum linearity. Linearity is originally defined in [209]. Figure 53 presents two examples from different generators having extreme linearity values.



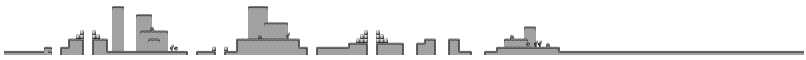
(a) linearity = 1

(b) linearity = 0

Figure 53: Example levels from (a) the parameterized notch randomized and (b) the ORE generators with very low and high linearity values.

Density is a measure of how many platforms are stacked on top of each other. The density calculator assigns a density value to each position de-

pending on how many different heights Mario could possibly stand on. The density value for a level is simply the average density value for all positions on the level. Density is defined in [191] and two example figures for levels from two different generators with comparable density score are presented in 54.



(a) density = 0.81

(b) density = 0.88

Figure 54: Example levels from (a) the notch and (b) the hopper generators with comparable density values.

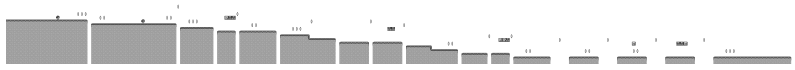
Pattern density measures how many meso-patterns from the original Super Mario Bros game can be found in the level. This metric is the same calculation as the evaluation function for the evolutionary algorithm in the pattern-based level generator, and is described in [49]. The metric is normalized according to level length. Figure 55 presents two illustrative levels for this measure.

Pattern variation, on the other hand, measures only unique occurrences of patterns and gives higher values to levels with diverse meso-patterns instead of many reoccurring meso-patterns. The metric is also normalized according to level length.

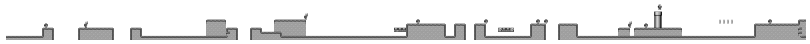
11.3.2.2 *Level distance metrics*

The other category of metrics are those that do not work on individual levels, but on pairs of levels by measuring how different they are, or in other words their *distance* in some space.

In the comparison performed for this paper we only have one level distance metric: **compression distance** is a domain-general metric based on



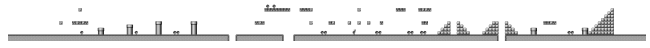
(a) pattern density = 0.17



(b) pattern density = 0.172

Figure 55: Example levels from (a) the launchpad and (b) the GE generators with comparable pattern density values.

the principle that if two strings are similar, you save more space when compressing them together. In this implementation, we use the standard gzip algorithm and compare the length of the resulting string when compressing each of two levels individually and when compressing them together. Compression distance is described in [127] and applied to platform game levels in [191]. Figure 56 presents two example levels transcribed from the original game that are found to be very dissimilar to each other according to this measure.



(a)

(b)

Figure 56: Examples from the original levels that are dissimilar according to the compression distance, ncd = 0.9.

Another example of a metric that would fit this category is the edit distance metric used in clustering *Launchpad*'s rhythm groups [209].

11.4 GENERATOR COMPARISON

All level generators were instructed to output levels of approximately 200 blocks in length, which on average would be about a minute of playing time for a proficient player, and which is close to the median length of the original SMB levels. All level generators were used to produce 1000 unique levels; from the original game, we have 22 unique levels (omitting 10 levels from the original, since 2 levels are "under water" and 8 levels are "boss-fight levels") We never analyzed the bonus areas since they primarily are "warp zones" or filled with coins.

There are seven generators included in the analysis, as described in Section 11.3.1. Of these, there are two level sets produced by the *Parameterized Notch* generator, and two level sets produced by the *Launchpad* generator, with different parameter settings. The *Parameterized Notch Randomized* level set comes from the *Parameterized Notch* generator, with the values of the controllable parameters chosen at random. The *Launchpad-Rhythm* levels come from the *Launchpad* generator, by varying only the rhythm parameters while holding the length of rhythm groups and component probabilities constant. The *pattern-based* generator has three major variants, arising from three different fitness functions being used: *pattern occurrence* (counting each meso-pattern only once), **pattern count** (counting each occurence of a meso-pattern) and *weighted pattern count*, where the patterns are weighted by their rarity. Remaining generators each had a single set of levels generated for them. The result is 12 different sets of levels to be analyzed and compared.

The remainder of this section describes the results of these experiments, including providing visualizations of expressive range for several generator/metric combinations, and a brief description of the controllability for each generator.

11.4.1 *All Metrics*

Table 30 presents a high-level comparison of all generators. For each generator, we present the average value of its levels on all metrics, and the stan-

dard deviation of the that value. A number of observations can be made based on this table. To start with, there is a lot more variance between generators (as compared to levels generated by the same generator) on some metrics than others. For pattern density, the variation between generators is comparable to the variance "within" generators (on levels generated by the same generator), whereas for lenience, linearity and density it is much higher. Therefore, it seems that the latter three metrics are better for telling level generators apart; a complementary interpretation is that all generators are bad at providing variance in those three dimensions.

Studying each metric in detail, we can see a number of patterns. The lowest leniency value can be found for occupancy-regulated extension. Looking at the levels, it is clear that they feature more gaps than other levels. This doesn't necessarily make them more difficult, as there are often many different jumping-off point for each gap. For the linearity metric, the outlier is instead the two versions of the Launchpad generator, which have much higher linearity (and larger variety with respect to linearity) than the other generators. It is here very clear that Launchpad was originally designed with another kind of platform game in mind, more akin to rhythm-based games such as *Sonic the Hedgehog* [212] which feature more or less constant forward motion.

For the density metric, the real outlier is the pattern-based level generator. As described above, the density metric counts the number of platforms at different heights at each tile. The original levels have a relatively low density as well, but not at all as low as the levels generated by the pattern-based generator. This points to that level segments with multiple level platforms are for some reason not reproduced very well by the pattern-based generator (according to its design criteria, an ideal pattern-based generator should have values similar to those of the original levels on all metrics). Several other generators, including the Notch generators, generate far too many overlapping platforms of different height as compared to the original levels.

The two pattern-based metrics are dominated by two different versions of the pattern-based generator, as would be expected given that these met-

rics are fitness functions for these two versions of the generator. The highest value on pattern density is thus scored by the pattern count generator, and the highest score on pattern variation by the pattern occurence generator. Both ORE and the original levels have very high scores on both pattern variation and pattern density, which is logical given that the pattern that the metric looks for are based on the original levels, and on that the ORE generator manages to cram a lot of structure into a short level space. The two versions of Launchpad scores low on pattern variation, pointing to the relatively sparse character of their levels; the Notch levels also scores low on this metric. Interestingly, pattern density and pattern variation appear highly correlated except when it comes to the pattern-based generator, where they diverge sharply.

The compression distance metric, being fundamentally different in that it is a between-level metric, must be discussed separately. One thing that stands out here is that the various versions of the Notch generator have the lowest compression distance. This indicates that the levels, from an information-theoretic viewpoint, are all very similar. Interestingly, this corresponds very well with qualitative observations that the levels in Infinite Mario Bros appear quite similar to each other. On the other end of the scale, the pattern-based levels have the highest compression distance, meaning that very little is gained by compressing two such levels together. This could be explained by the way that generator assembles levels out single slices, micro-patterns with length 1 block. As there are no fixed orders of slices (a given meso-pattern could be implemented through many different slice combinations), this means that a compression algorithm based on finding commonly occurring substrings would not find much to build on.

Figure 57 shows a visualization of every metric and generator using a boxplot graph. Each of the six metrics is clustered together per generator. This visualization shows that each generator has its own unique profile for the sets of metrics, not only in mean and standard deviation (as shown in Table 30), but also in the overall range of each metric.

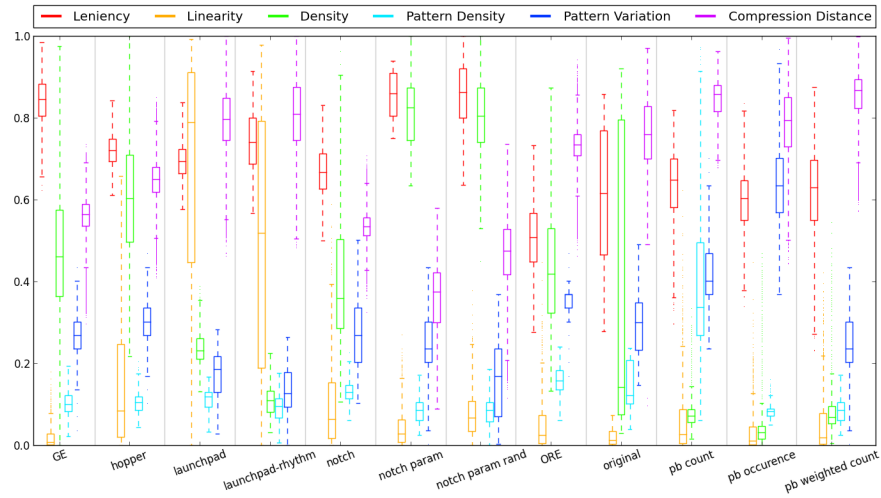


Figure 57: A visual comparison of all generators included in this analysis using all of the metrics. Each generator is evaluated using six metrics, denoted in different colors. The boxplot for each generator-metric pair shows the median, and upper and lower quartiles. The whiskers extend to data points that fall within 1.9 IQR of the upper and lower quartile, and outliers from this range are depicted as small dots.

11.4.2 *Expressive Range Visualization*

The evaluation framework thus far has viewed each metric largely independently from the others. While this provides a good high-level view on the properties of a particular generator, it does not indicate any relationships between metrics or show the shape of a generator's expressive range. For example, viewing all metrics at a high level can show that a particular generator might be pre-disposed towards creating a medium range of levels in terms of both linearity and density, but it would not be able to show any correlation between levels according to those metrics. Visualizing the expressive range of generators allows such biases to be easily seen [209]. This visualization involves plotting a 2D histogram as a heatmap, where each axis on the plot is one of the metrics, and each bucket in the histogram is assigned a color based on how many levels are in the bucket.

For more than two metrics, there is no clear way to produce a multi-dimensional histogram. While there are several potential uniform visualizations that show metric values for each individual level (e.g. a circular heatmap, or a stacked bar chart), the crucial insight gained from expressive range evaluation is identifying dense areas in a plot resulting from many levels sharing similar metric values. Thus, in this paper, we have chosen to compare expressive range by generating several graphs per generator for pairwise combinations of metrics. For space reasons, not all expressive range graphs are shown here; we have selected one of these sets of graphs with particularly interesting features to show in this paper.

In each of the graphs corresponding to metric pair shown in Figure 58, the warmest area of the heatmap is red (corresponding to 40 or more levels in that bin), while the coolest area of the heatmap is dark blue. Expressive range for the original SMB levels uses a different scale, however, as there are considerably fewer levels available to measure; the warmest area of the Original SMB level heatmaps corresponds to 5 levels being in that bin.

Figure 58 shows several subtleties of the generative space for each generator formed by the density and leniency metrics. The rough shapes of the expressive range correspond to what would be expected, given the range

of each metric shown in Figure 57. However, the version of *Launchpad* that varies its rhythm parameters (row 1, column 4) shows an unexpected correlation between density and leniency, which is not mirrored in the fixed-rhythm version. The *Parameterized Notch* generator (row 1, column 6) shows that it biases two distinct clusters of levels, again with a slight correlation between the two metrics. From all of the graphs side by side, it is clear that the expressive ranges of each of these generators are overall quite different, though there are some similarities. Overall character and shape of the generators is somewhat easier to interpret in these graphs than using a boxes and whiskers diagram.

11.4.3 *Controllability*

As discussed in the introduction, there is a potential and sometimes perceived partial conflict between expressivity and controllability in procedural level generation. While this paper is chiefly about the expressive range of the various generators involved, an important aspect of evaluating content generators is evaluating how they can be controlled. Here we discuss the ways in which each level generator can be controlled, to help us in gaining an initial understanding of the relationship between controllability and expressivity in this domain. Table 31 summarizes how each generator can be controlled by a designer; note that the table contains only the main generators, and does not list different configurations of the same generator.

The *compression distance* metric can be useful for understanding the impact of parameters in a parameterized generator. By illustrating the compression distance as a 2D matrix with a heatmap applied to it (with cooler colors representing low distance), it is possible to see patterns with sets of levels that have low distance between each other. For example, Figure 59a shows the compression distance matrix for the *Parameterized Notch* generator. The checkerboard pattern corresponds to common combinations of parameters: those that share the same parameters are more similar to each other than those that do not. When examining the compression distance matrix for *Launchpad* with varied rhythm parameters (Figure 59b), a different

effect is seen; variety is overall higher (as reflected by higher compression distance scores), with particularly high variety when the rhythm beat type is regular.

11.5 FUTURE WORK

There are many possible metrics that we have not included in this study. These include metrics that measure macro-scale progression and repetition in the level. They also include simulation-based metrics, which would use an artifical agent to play the level and analyse its playing style. Further, we could use metrics that try to judge the shape of the level, for example through computer vision methods. Or we could associate individual level patterns and situations with player experience through machine learning, and build level metrics on top of the output of such models. Lacking any previous comparative PCG evaluation, we focused primarily on existing research metrics.

A question that becomes more pressing the more metrics we accumulate is how to choose between them, or perhaps combine them. One way would be to use principal component analysis, or some similar dimensionality reduction technique. This could give us a smaller number of joint metrics that still capture the essential variance between levels. Or simpler, we could cross-correlate the various metrics and only keep the least correlated ones. However, we also need to weigh the importance of having human-interpretable metrics and results; it is important for designers and AI researchers to understand how generators differ from each other in a design-relevant context.

This assumes all metrics are somehow equally important. Clearly, that is not true for most specific intendend usages, e.g. to design an intriguing, fun or challenging level. We would therefore need to complement our computational investigation with user studies, where we associate metrics with their effects on player experience. The level distance metrics could also be validated by investigating how different to each other various levels are perceived to be.

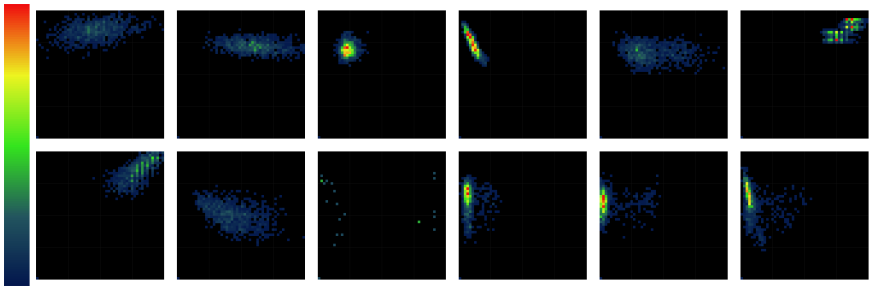
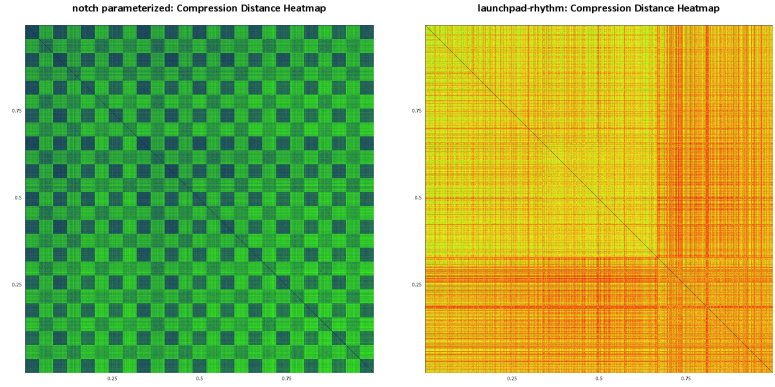


Figure 58: Heatmaps visualizing the expressive range of each generator according to the Density (x-axis) and Leniency (y-axis) metrics. The order of generators (left to right, top to bottom) is: GE, hopper, launchpad, launchpad-rhythm, notch, parameterized notch, parameterized notch-randomized, ORE, original levels, pattern-based-count, pattern-based-occurrence, pattern-based-weighted-count.



(a) (b)

Figure 59: Heatmaps visualizing the compression distance matrix, showing the impact of varying parameters. (a) Parameterized Notch generator. (b) Launchpad with varied rhythm parameters.

Finally, the comparison of generators performed here is only possible because each generator shares a common context and framework. Evaluating within a common framework is helpful; however, it also obscures the importance of creating a content generator to meet a specific game's context. Clearly, some metrics can be easily applied to multiple level generation contexts (such as compression distance) while others may need to be fine-tuned for a new context.

11.6 CONCLUSIONS

We have defined a framework for evaluating and comparing the expressivity of level generators, and quantitatively compared seven different platform game level generators (and several variations of them), along with the original Super Mario Bros levels, using six different metrics. We have also discussed the role of controllability in level generation and its relation to expressivity. Our results constitute the first quantitative comparison of multiple level generators, and contain both expected and unexpected outcomes. Among the expected outcomes are that the differences between generators on most metrical dimensions correspond to the qualitatively observed differences between levels generated by them. Among the unexpected outcomes is that parameterization plays a very large role in changing the nature of generated levels by some generators (e.g. Notch, Launchpad) but not others (e.g. pattern-based). Metrics that correlate for one generator might not correlate for another version of the same generator. We believe the information contained in this paper provides a good baseline against which to characterize new generators and metrics, and have made freely available level samples and source code.

Table 30: Overview comparison of level generators: mean value (standard deviation) of each metric on the output of each generator.

generator	leniency	linearity	density	pattern dens.	pattern var.	compr. dist.
GE	0.84 (0.06)	0.02 (0.03)	0.47 (0.16)	0.1 (0.03)	0.27 (0.06)	0.56 (0.04)
hopper	0.72 (0.04)	0.15 (0.16)	0.6 (0.15)	0.1 (0.02)	0.29 (0.05)	0.65 (0.05)
launchpad	0.7 (0.05)	0.66 (0.31)	0.24 (0.04)	0.11 (0.03)	0.17 (0.05)	0.8 (0.07)
launchpad-rhythm	0.74 (0.07)	0.49 (0.32)	0.11 (0.04)	0.09 (0.03)	0.13 (0.06)	0.81 (0.09)
notch	0.67 (0.06)	0.1 (0.11)	0.4 (0.16)	0.13 (0.02)	0.27 (0.08)	0.53 (0.03)
notch param	0.85 (0.06)	0.04 (0.05)	0.81 (0.08)	0.08 (0.03)	0.24 (0.07)	0.36 (0.08)
notch param rand	0.86 (0.08)	0.08 (0.06)	0.8 (0.1)	0.08 (0.03)	0.17 (0.09)	0.47 (0.08)
ORE	0.51 (0.08)	0.05 (0.06)	0.43 (0.15)	0.16 (0.03)	0.35 (0.05)	0.73 (0.04)
original	0.61 (0.18)	0.02 (0.02)	0.35 (0.37)	0.14 (0.06)	0.3 (0.1)	0.76 (0.11)
pb count	0.63 (0.1)	0.07 (0.09)	0.08 (0.05)	0.39 (0.17)	0.41 (0.07)	0.85 (0.04)
pb occurence	0.6 (0.08)	0.04 (0.06)	0.06 (0.09)	0.08 (0.02)	0.64 (0.11)	0.79 (0.08)
pb weighted count	0.61 (0.12)	0.06 (0.08)	0.09 (0.08)	0.08 (0.03)	0.24 (0.07)	0.86 (0.05)

Table 31: Controllability of the main generators tested in this paper, using vocabulary from [200].

generator	**control type**
GE	*indirect*, via changing evolution parameters
hopper	*parameterized*, for implicitly defined difficulty levels
launchpad	*parameterized*, for component appearance and rhythm
notch	*none*
notch (param.)	*parameterized*, for component appearance
ORE	*knowledge representation*, can change input chunks
pattern-based	*indirect*, via changing evolution parameters; and *knowledge representation*, can change input patterns

12

PAPER 6 – LINEAR LEVELS THROUGH N-GRAMS

Steve Dahlskog, Julian Togelius & Mark J. Nelson

ABSTRACT

We show that novel, linear game levels can be created using n-grams that have been trained on a corpus of existing levels. The method is fast and simple, and produces levels that are recognisably in the same style as those in the corpus that it has been trained on. We use *Super Mario Bros.* as an example domain, and use a selection of the levels from the original game as a training corpus. We treat Mario levels as a left-to-right sequence of vertical level slices, allowing us to perform level generation in a setting with some formal similarities to n-gram-based text generation and music generation. In empirical results, we investigate the effects of corpus size and n (sequence length). While the applicability of the method might seem limited to the relatively narrow domain of 2D games, we argue that many games in effect have linear levels and n-grams could be used to good effect, given that a suitable alphabet can be found.

PUBLISHED IN

AcademicMindTrek '14, November 4-7, 2014, Tampere, Finland.

doi: 10.1145/2676467.2676506

LINEAR LEVELS THROUGH N-GRAMS

12.1 INTRODUCTION

Procedural content generation in games (PCG) is the algorithmic creation of game content, either with limited or no human input. Both academia and industry (ranging from AAA-titles to independent productions) have shown interest in PCG in the past few years. PCG has been used to solve numerous content generation problems ranging from runtime level or item generation to design time generation of terrain, game rules or vegetation, using a multitude of different techniques including agents, evolutionary computation, constraint solving, etc. [231].

Recently, participants in a Dagstuhl symposium on artificial and computational intelligence and games proposed a set of long-term goals and research challenges for PCG in a overview paper [229]. These grand goals proposed for PCG are: *Multi-level Multi-content PCG*, *PCG-based Game Design* and *Generating Complete Games*. The paper also proposed nine more concrete research challenges that would support advancement towards reaching the identified grand goals of PCG. Additionally, five concrete actionable steps were proposed, all of which was envisioned to target one or several of the research challenges.

In this paper we look into one of the proposed concrete research challenges, *Representing Style*, and an associated actionable step, *Competent Mario Levels*. Representing style means producing a generative model that follows a particular school of design thinking or a particular designer's recognised style. The Competent Mario Levels actionable step proposes investigating this question by creating level generators for the classic platform game *Super Mario Bros.* (SMB) with the ability to generate varied, interesting, playable, entertaining and good-looking levels.

It is instructive to look at domains other than game content generation to see whether there are methods and ideas that could be brought to bear on a given content generation problem. Are there domains that show similarities to the game domain we are currently investigating, and what techniques do they use? The *n-gram* method has frequently been used to model style and generate novel "randomised" artefacts in two other creative domains, text and music. The n-gram method is very simple – essentially, you build conditional probability tables from strings and sample from these tables when constructing new strings – and also very fast. As simplicity and speed are virtues in PCG, as in so many other domains, it is worth investigating the merits of this method seriously. As far as we know, n-grams have not been used for level generation before (though 2D Markov chains have; see Section 12.2.3).

We investigate whether we can model the style of the original *SMB* levels by calculating n-gram statistics from those levels, treated as linear sequences of vertical level slices, and then using the resulting Markov level model to produce novel levels that are playable and similar in style to the original levels. As n-grams are used with strings of symbols (such as characters or words, when modelling natural language), we need an "alphabet" for expressing SMB levels as strings. For this purpose, we use a representation of the SMB levels we call "micro-patterns", which are thin vertical slices of a level. In previous work, these slices or micro-patterns were used to create levels where "meso-patterns" and "macro-patterns" decided the order of the slices and therefore gave the levels a structure, style and meaning [48, 49, 51]. Figure 60 shows examples of such slices.

12.2 CAPTURING PLATFORMER LEVEL STYLE WITH N-GRAMS

Automatically generating levels for *Super-Mario-Bros.*-style platformer games has been studied relatively frequently by procedural content generation researchers [101, 190]. SMB levels have several features that make them tempting to generate automatically. First, they have a quite obvious recurring, pattern-based structure: any player who plays for even a short pe-

riod of time will start to see similar or even identical patterns of platforms, blocks, and enemy placement reappear. Secondly, the levels are typically oriented in a primarily linear, left-to-right direction, so they can be thought of as a sequence of level elements, which is traversed in order.

12.2.1 *N-gram style capture*

Several other (otherwise rather different) domains are also characterised by linear sequences that are traversed in order, and exhibit recurring patterns: sequences of notes (music) and sequences of words (language) are two well-known domains in which there has been considerable research into generative methods. An interesting (and early) line of work in such domains has been to model them purely statistically, at a surface level. By surface level what's meant is that models consider only the raw sequences, and don't analyse them in terms of higher-level or semantic structures such as "C major" or "a prepositional phrase". A simple way to do this surface-level statistical modelling is based on counting *n-grams* (n-element subsequences). Given a corpus of writing or a piece of music whose style we want to mimic, we count how often each n-length subsequence appears in the original. We can then produce a new sequence in the same "style" (for a certain definition of style, as we shall discuss) by stringing together n-grams sampled from this bag, weighted according to their original counts[24].

In this paper, we experiment with precisely this kind of n-gram-based generation, but with platformer levels. Our unit is a one-block-wide vertical slice of a level. A complete level is a left-to-right sequence of these vertical slices. This gives us a basic problem formulation very similar to the sequential n-gram-based generation used in music and natural language, allowing borrowing of techniques and cross-domain comparisons.

12.2.2 *Effects in other domains*

Since our focus here is *style*, it's worth briefly recounting some typical stylistic effects that n-gram-based generators have in these other domains. In lan-

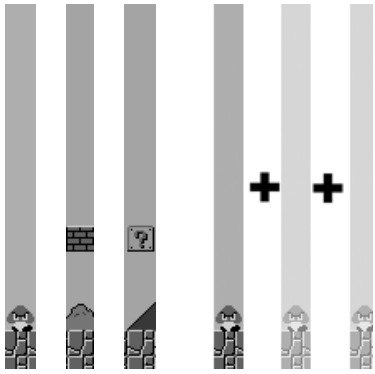


Figure 60: Different slices (micro-patterns) and a Goomba-horde.

guage generation, n-gram generation is most often perceived as a *parody* of a writer's style. Such usage dates back to at least the 1970s [24, item 176], and recurs frequently today, for example in web-based generators that produce n-gram-based mimicry of a Twitter user's updates. Such generators produce a kind of uncanny surface-level reproduction of style: they mimic the sequences of words that a particular writer typically strings together, but when interpreted as sentences or paragraphs, the result is usually nonsense, with little to no interpretable semantic meaning or higher-level structure.

It seems unlikely that level generation will be interpreted in precisely the same way, as a *parody* of a game's levels—though this possibility cannot be ruled out completely. Levels are not typically communicating the same kind of high-level semantic information via their structure as natural language is, and the surface-level style is comparably more important.

A closer comparison may be the case of music. There, the primary complaint has been that n-grams fail to produce interesting high-level structure, at least without being pushed in a way that causes them to lose most of their generativity [156, ch. 3]. With a low n, the music ends up consisting of notes that are locally reasonable, but with an overall piece that wanders in an uninteresting, aimless way, lacking conventional musical features such as

movements, recurring themes, loud and quiet periods, perhaps even an interpretable time signature. On the other hand, when n is increased to add larger context to the generator, it soon degenerates into splicing together large preexisting snippets of music—the result of an overfit (insufficiently smoothed) statistical model.

Therefore one of our initial questions to investigate here is whether n-gram generation in *Mario*-style levels results in the same basic problem, of level that are either too wandering, or too cut-and-pasted verbatim from the source material.

12.2.3 *Information content*

Viewing platformer levels as sequences of slices leads to interesting analytical and generative possibilities connected to information content and/or entropy of the various sequences. Essentially any sequence of units can be seen as a code transmitting information, and be analysed (with more or less usefulness) using tools from information theory. In fact the first instance of n-gram-based modelling of natural language was performed by Claude Shannon, in the same paper in which he introduced information theory [194].

Later papers used the approach to, for example, statistically characterise the average information carried by each letter of the English alphabet [195]. Use of the information-theory connection to provide control over a generative process also has old roots, dating at least back to 1960s work in computer-music, which used the entropy rate estimated from a Markov model as a tuning knob that the composer could use to vary the entropy of different parts of a piece [98].[25] To our knowledge, similar investigations haven't yet been performed with videogame levels.

This intent to investigate levels as sequences, and thereby gain a close connection to both information theory and previous work in sequence-based Markov modelling in other domains, is also why we don't follow Snodgrass and Ontañón [210] in modelling platformer levels as two-dimensional grids of blocks, and instead use one-dimensional sequences of vertical slices. Two-

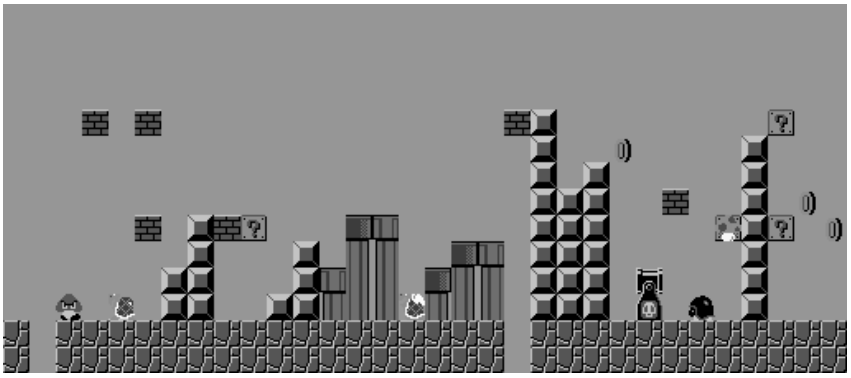


Figure 61: From left to right: the 32 most common slices from the original SMB levels. These slices would therefore be the most frequent unigrams.

dimensional extensions of Markov models, such as Markov random fields, have considerably different properties and less of a direct connection to progression over time—though they are indeed interesting to investigate in their own right, and this work is the most closely related to ours, in that it also uses corpus-based statistical modelling to capture level style.

12.3 METHODS

We represent levels as sequences of vertical level slices (or micropatterns). The full corpus of levels we used for n-gram training is comprised of 15 levels from the original *SMB* game. This includes all levels in the game except for those that have considerably different mechanics from the "normal" ones: water levels, mushroom-platform levels, and boss-fight levels are excluded. In addition, we use the slices within each individual level as "per-level" corpora, in order to investigate whether generators trained on different levels, or combinations of levels, have noticeably different styles.

The original levels vary in length from less than 200 to more than 300 blocks/slices; when generating, we chose a fixed level length of 100. If we incorporate all levels into one single slice-library, several unique slices (seen

only once in the game) are found. The largest single addition to the slice library is from level 1–2, due to its "roof" of brick-tiles almost at the top of the screen, which is an unusual arrangement. Combining levels 1–1 and 1–2 more than doubles the slice library from 29 to 73 slices. Figure 61 shows the most common slices that make up the SMB levels.

Our n-gram implementation works by creating separate tables of occurrences for unigrams, bigrams and trigrams. When generating new array (levels), probability tables are calculated based on the occurrence tables. Each new symbol is chosen directly based on its independent probability for unigrams (i.e. the probability for each symbol is exactly its frequency in the original levels). For bigrams, the probability for each symbol is its conditional probability given the preceding character; and for trigrams, its conditional probability given the two previous characters.

There are some special cases. When generating a level using a bigram, the first character will be based on a unigram trained on the same corpus (as there is no preceding character). When using a trigram, the first two characters are based on unigrams. Another special case is for bigrams or (especially) trigrams, when the preceding character or combination of characters has never been followed by anything at all in the corpus. In this case we use a *fallback*: if there is no trigram match for a character combination, we fall back to a bigram, and if there is no bigram, we fall back to a unigram.[26]

12.4 RESULTS

After initial validation of the functionality of the method, we carried out experiments to investigate the effects of varying n, the effects of varying the training corpus, and to characterise the expressive range of the n-gram generator. We also compared the characteristics of generated levels with those in the initial corpus.

12.4.1 *Effects of varying* n

The effects of varying n are rather drastic. Essentially, unigrams produce a haphazard mess, bigrams produce some local structure with much repetition and trigrams produce levels with good local structure that are stylistically similar to the training corpus. In order to demonstrate these effects we have randomly picked five example levels generated with each configuration.

In figures 62, 63 and 64, we use the same corpus; namely the first level of SMB (199 tiles wide and 14 tiles high) containing 30 different slices. For space saving purposes we have chosen levels of length 100 tiles for all our examples.

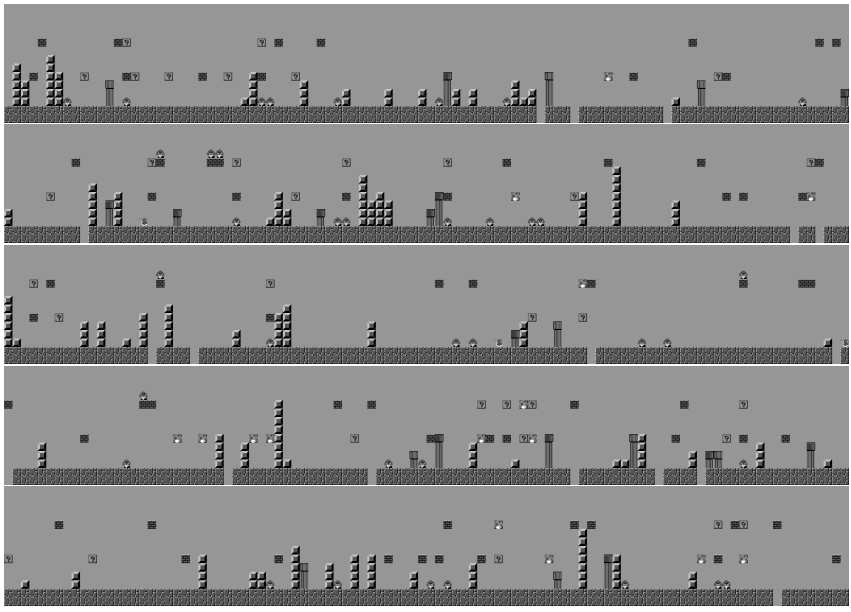


Figure 62: Unigram-based ($n = 1$) levels with SMB World 1–Level 1 as corpus.

In figure 62 we use $n = 1$ for our n-grams, resulting in a rather cluttered level layout with one additional drawback: incorrect pipes. Even though the

pipes would be mendable with some rules for the generator, this may need some testing in order to balance the occurrence of pipes. Overall, these levels lack a feel of designed structures, and there is no sense given of imitating any particular style.

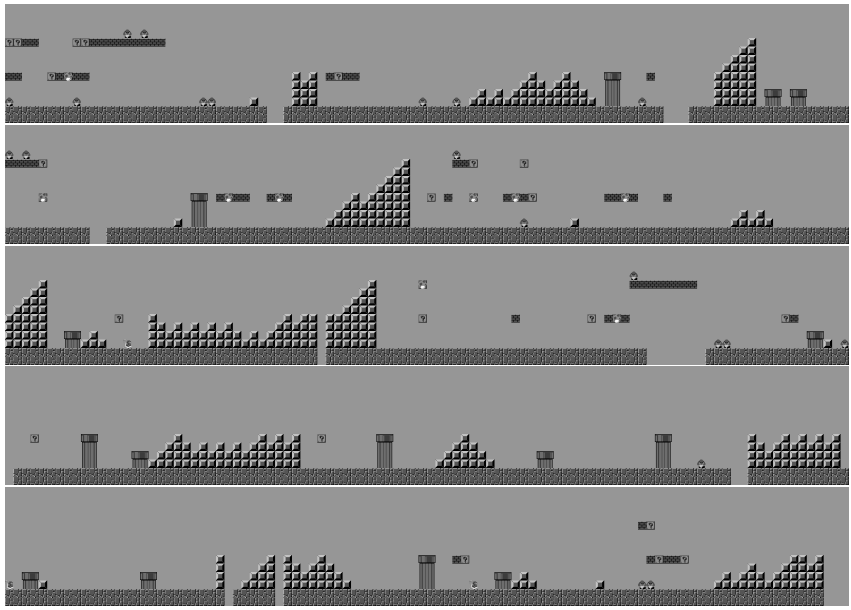


Figure 63: Bigram-based ($n = 2$) levels with SMB world 1–level 1 as corpus.

After the unigram test we moved on to bigrams ($n = 2$). Figure 63 shows example generated levels, which have working pipes, but also some strange (by SMB standards, that is) "mountain ranges", stairs going both up and down. There are also very few enemies present. Although there seem to be some kind of designed structures, the presence of mountain ranges, however distinct in style, does not convey the style of World 1–Level 1, or of any level in the original SMB. When we increase n to 3, as seen in figure 64, we get correct structures instantiating meso-level design patterns like pipe valleys and stairs (see discussion in [48]), and enemies are also present. Overall, these levels bear a strong stylistic similarity to SMB World 1–Level 1.

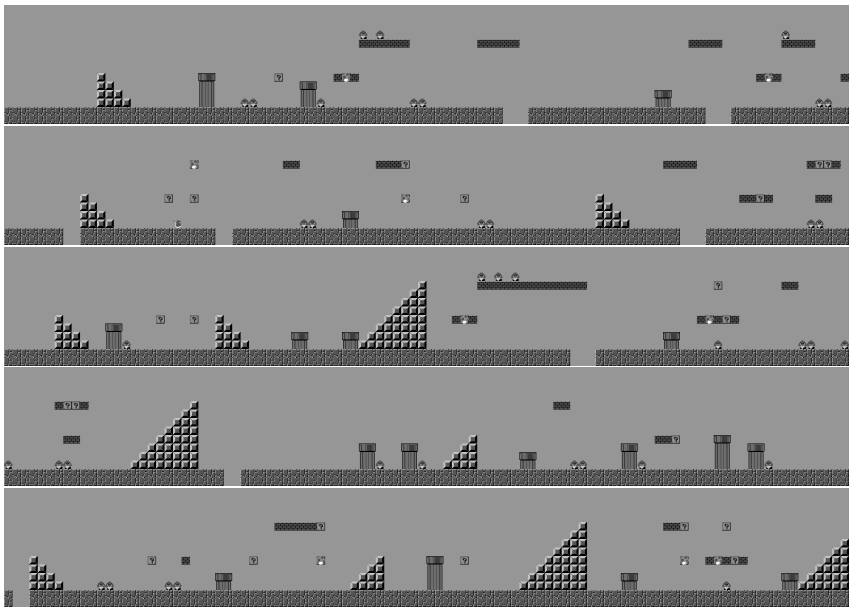


Figure 64: Trigram-based ($n = 3$) levels with SMB 1–1 as corpus.

12.4.2 *Effects of varying training data*

Of course, SMB contains a lot more content than just World 1–Level 1 (*1–1*). There are 32 levels across 8 worlds, some of which are very different from the others in style, e.g. water levels and boss fight levels. But the appearance of diversity is to some extent deceptive, as several levels are minor but clever variations of others. For instance, levels 1–3 and 5–3 are structurally identical, but with *Bullet Bill*s added to 5–3. Likewise, levels 5–1 and 7–1 are structurally similar, but 7–1 is more *Bullet Bill*-dense. And levels 1–1, 2–1, and 6–2 are all similar, and fairly similar to 1–2 and 4–2.

In order to investigate the results of training n-grams on more than one level, we created two new corpora: one based on levels 1–1 and 1–2 (see figure 65) and another based on levels 1–1, 1–2 and 2–1 (see figure 66). The combined corpora were created by simply concatenating levels. In figure 65,

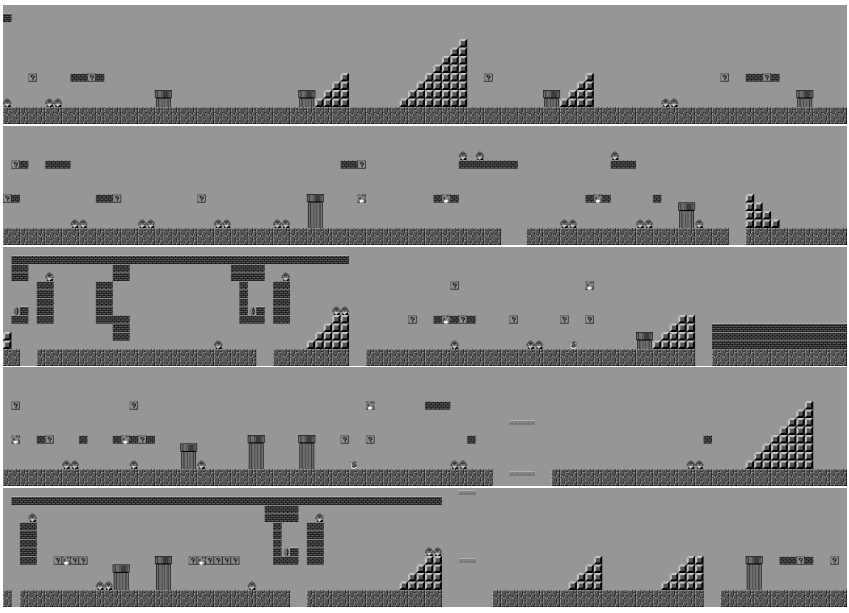


Figure 65: Trigram-based ($n = 3$) levels with SMB 1–1, 1–2 as corpus.

we can see the effect of training on a corpus consisting of multiple levels in subfigures 3 and 5, where the generated level abruptly changes style. Both begin with the style of level 1–2, and whereas the last one ends in the style of 1–1, the middle one returns to the style of 1–2 after generating a middle section in the style of 1–1.

Continuing with figure 66 we can see that the combination of different levels (e.g. all levels present in the middle and last one) as well as times when the generator sticks to one level style (e.g. the second example). By training on specific levels, the generator follows that particular style for that level, but the larger the corpus, the larger the variation.

As we extended our corpus to include all ground-based levels (no boss, no water and no mushroom/platform levels) each level's traits became part of the corpus and thus influenced the output. Unfortunately the structure of levels becomes clear when incorporating 15 levels (including under-ground

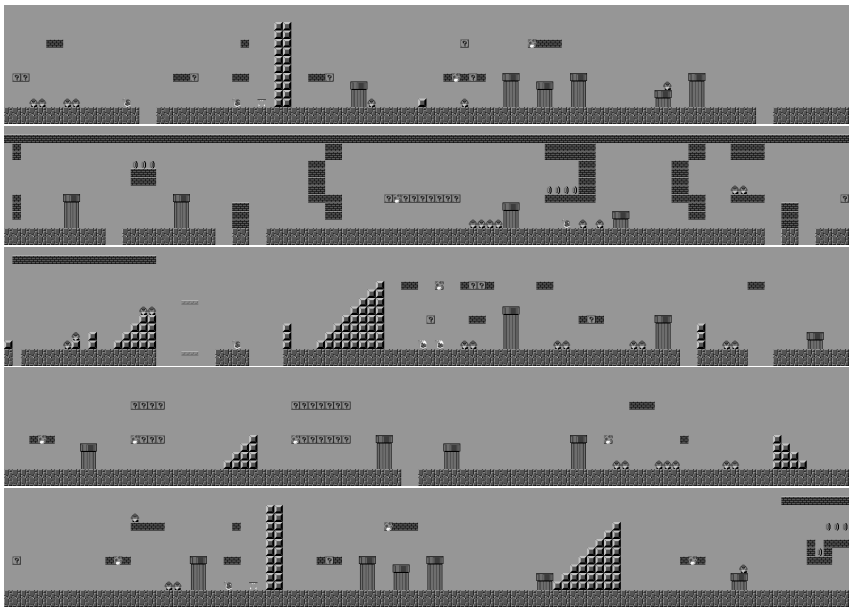
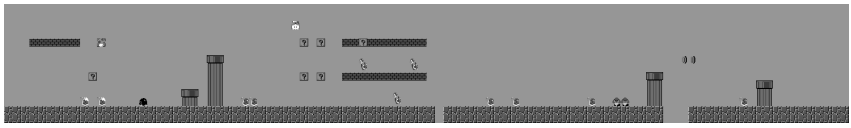


Figure 66: Trigram-based ($n = 3$) levels with SMB 1–1, 1–2 and 2–1 as corpus.

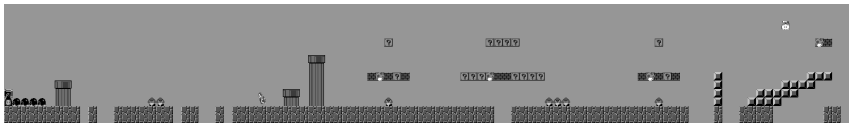
levels). Each level starts off with 16 "simple ground" slices with just ground and nothing interesting allowing the player to start in a safe spot. The effect on the generated levels is wide sections of space and nothing interesting from a play perspective. In order to generate interesting levels we shorten these safe-spots so that they do not become to influential in the corpus. We also remove the underground levels for the purpose of preserving the style of surface-levels.

12.4.3 *Expressive range*

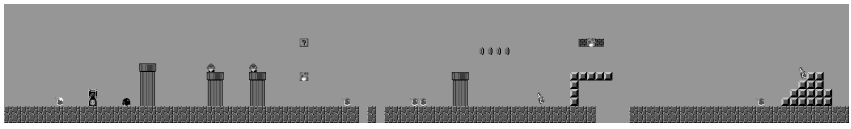
In order to show the diversity of the method we employ the concept of *expressive range* analysis [101, 202]. Expressive range analysis is a tool for characterising and exploring a PCG method by using a metric; essentially, a number of artifacts (in this case levels) are independently generated, and



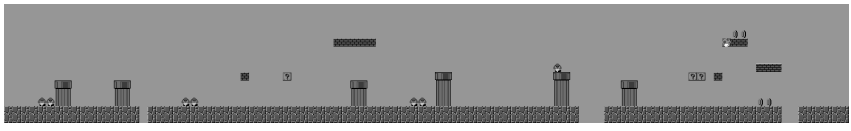
(a) Level: 704 Linearity +96 (MAX).



(b) Level 118: Linearity −16 (MIN).



(c) Level 50: Leniency +8 (MIN).



(d) Level 20: Leniency +44 (MAX).

Figure 67: ($n = 3$) levels with pruned corpus 2600 slices (15 levels from the original SMB with the first screen of each level removed).

plotted in the 2-dimensional space of two different metrics. We use the metrics Linearity and Leniency for expressive range comparison (see figure 67 and table 32). A level with a high linearity value forces the player to jump more often than a level with low linearity value. High Leniency means more enemies and gaps where the player may lose a life. An expressive range analysis of 1000 generated levels shows that the output of the n-gram level generator exhibits considerable diversity, at least in these two dimensions.

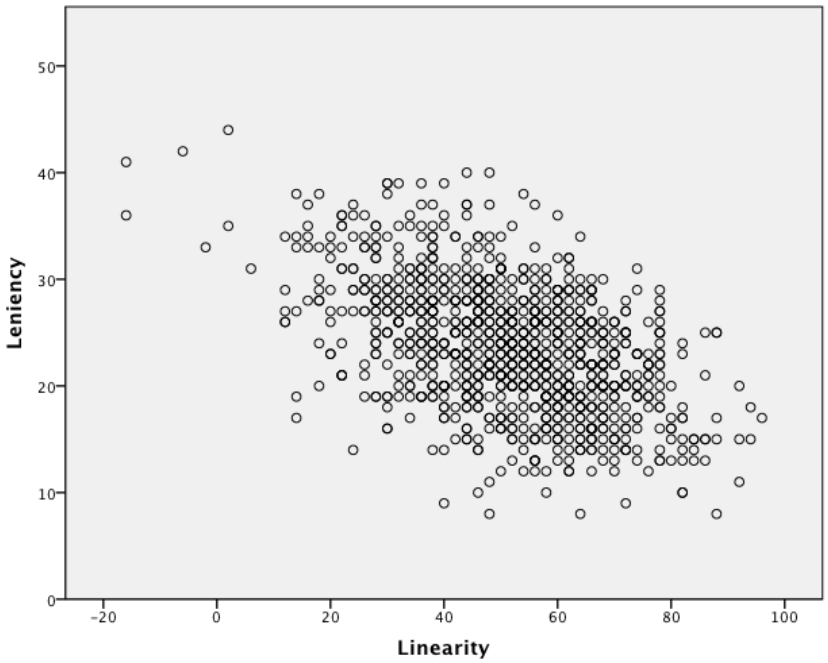


Figure 68: Leniency and Linearity for 1000 above ground pruned levels. Higher Leniency means more difficult. Higher Linearity means flatter levels.

Table 32: Linearity and Leniency.

n-gram callbacks per 1000 levels	252
Linearity Average	51.23
Linearity MIN	-16
Linearity MAX	96
Linearity DEV.	16.79
Leniency Average	23.704
Leniency MIN	8
Leniency MAX	44
Leniency STD.	6.01

12.5 LARGE SCALE COMPARISON

To answer the question of whether this method really allows us to copy style, we did a large scale statistical study of whether generated levels retain the style of those levels that go into their corpus. We chose to use the measures of linearity and leniency, discussed above, as measures of style. If the levels that are generated from a particular corpus have similar measures values for linearity and leniency as those in the corpus, we reason that the levels are similar in style in at least this respect.

We generated 1000 levels based on each original level (only one level in the corpus). We measured the linearity and leniency for each original level and compared that value to the calculated average value for each of the groups of the generated levels. In general, levels generated using n-grams have linearity and leniency values very close to those of the original levels (see Table 33). Exceptions do exist (World 3–Level 1 and 5–1 differ on leniency, and level 1–3 differs on linearity). For level 1–3 the difference may be related to the short original level (length 140), but the other two levels have just below average length. Level 5–1 is on the other hand rather spacious, which may account for the difference in leniency.

Table 33: *Line*arity & *Len*iency comparison between original & average value (1000 generated levels).

Corpus	Lin. (SMB)	Lin. (gen.)	STD (gen.)	Len. (SMB)	Len. (gen.)	STD (gen.)
1-1	26.995	26.486	7.599	-8.295	-8.108	8.481
1-2	19.490	19.277	9.025	10.365	10.241	11.445
1-3	46.030	49.286	16.501	-1.623	-1.429	13.055
2-1	20.152	19.900	5.619	7.082	6.965	7.412
2-3	61.190	60.177	13.822	-35.778	-35.398	18.041
3-1	32.995	33.511	9.012	0.108	-0.532	8.971
3-2	16.385	16.176	7.506	1.455	1.4715	10.538
3-3	45.574	44.286	15.042	-3.983	-3.571	10.548
4-1	15.133	15.640	7.878	16.044	16.588	13.291
4-2	27.411	27.273	11.660	14.318	13.904	14.405
4-3	51.682	51.449	10.336	-22.575	-22.464	9.481
5-1	19.484	19.251	5.345	-1.538	-1.070	8.430
5-2	22.967	22.660	7.258	2.101	1.970	7.547
5-3	43.299	42.857	15.929	-2.731	-2.857	11.812
6-1	31.809	31.609	9.377	-17.194	-17.241	10.516
6-2	34.753	34.597	6.126	20.397	19.905	7.717
6-3	49.930	49.359	8.925	-9.496	-8.974	13.213
7-1	23.768	23.780	7.244	8.372	7.927	9.185
7-3	67.247	66.667	11.575	-40.883	-41.441	19.876
8-1	20.544	20.442	5.922	13.615	13.536	11.656
8-2	16.533	16.019	7.512	17.336	17.476	7.846
8-3	16.273	16.337	8.355	9.288	8.911	6.427

The Pearson correlation between linearity of source and generated levels is 0.9985; for leniency, the value is 0.9998. These very strong correlations confirm that the generated levels are indeed very similar to the source levels using this particular measure of style.

12.6 DISCUSSION

It is clear that several of the configurations of the n-gram level generator, in particular trigrams trained on one level or a few similar levels, generate playable and good-looking levels in the style of those that form the corpus. Thus, the method performs well according to the original criteria (it also takes mere milliseconds to generate a level). It is worth discussing what makes the method work, and how it could be used for other game content generation problems.

12.6.1 *The importance of the representation*

The success of the n-gram method in this study is partly because we managed to find a useful set of building blocks for the levels, or in other words a suitable "alphabet". The micro-patterns are few enough to allow meaningful n-gram training even with a small corpus of just a few levels, or in other words strings of just a few hundred characters' length.

A key part of the representation is that the level is seen as a single string, i.e. one-dimensionally. Several other ways of representing an SMB level for n-gram generation would be possible. For example, we could generate each row of blocks separately, with the alphabet consisting of individual blocks and each level consisting of 15 or 20 strings on top of each other. This would have the advantage of a small alphabet, but the considerable disadvantage of the different rows being completely disconnected and the level likely being unplayable. A similar effect would be expected if trying to generate columns rather than rows. While Snodgrass and Ontañón [210] use individual blocks as their alphabet, their use of a 2D Markov chain tech-

nique allows this as it takes both horizontal and vertical interactions into account [210].

One could also imagine using longer, larger building blocks, for example sequences that are 5-10 blocks wide, similar to the meso-patterns in our original pattern analysis of SMB levels. However, this would lead to much less perceived variety, as the individual building blocks would be easily identifiable as components of the level.

12.6.2 *Pruning the corpus*

In our case the corpus contains several longer sections where the only slice used is the simple ground, since almost all levels in SMB start with 16 slices (a whole screen) of safe area for the player to start in. These longer simple ground sections can skew the n-gram-generation to create uninteresting sections; since n-gram generation has no high-level context, these runs intended to appear at only the beginning of levels can also end up appearing in the middle of them. Similarly, the presence of either extremely common or extremely unique sequences may result in stereotyped structures being simply "copied" from the training corpus recognisably, limiting the variety (in the second example of figure 66, the "c"-like structure appears twice). If the corpus is unbalanced in such manner, we suggest pruning it to reach the desired effect. Alternatively, the generation algorithm could be modified through using a logarithmic transformation on the frequencies, so that less frequent slices are relatively more often chosen (there are many other possible smoothing methods and frequency transformations that can be tried).

12.6.3 *Linearity in game levels*

While the method presented here works well for SMB levels, it could be argued that its usefulness is limited to other side-scrolling platformers, or perhaps also similar games such as 2D scrolling shooters (e.g. *R-type*). However, this restriction is not quite as severe as it may appear, if we take into account games whose levels are structurally linear, even if they don't ap-

pear as literal left-to-right side-scrolling sequences. Many games that ostensibly feature 3D worlds with full 3D spatial movement are actually built on linear levels; examples include shooters such as *Halo* and *Call of Duty* (in campaign mode at least), racing games such as *Need for Speed* and *Forza Motorsport* and 3D endless runners such as *Temple Run* and *Canabalt*. Given the identification of a suitable alphabet, the n-gram method could be used as is. Branching paths could be handled by simply generating separate strings for the different paths following a branching point.

12.7 CONCLUSION

We have shown that n-grams, trained on a corpus consisting of one or several levels from the original *SMB* game, can be used to effectively generate levels that are similar in style to the level(s) used in the corpus. The method is fast and reliable, and gives a reasonable diversity in several dimensions. Using $n = 3$ gives markedly better results than $n = 2$ and particularly compared to $n = 1$ in terms of the visual appeal of the level, and most probably in terms of playability as well. Using a corpus consisting of several levels increases the variety among produced levels, but can lead to surprising shifts in style. It is also found that the generated levels are indeed (on average) very similar to the levels used to learn the n-grams, showed that the method accurately reproduces at least some aspects of style. This simple method has potential to be useful for a large number of games, and should be investigated further in other game domains.

NOTES

[24]From a more statistical viewpoint, counting n-grams gives us the maximum likelihood estimate of a $(n-1)$th order Markov model presumed to have generated the observed text. Generation then consists of sampling from this Markov model.

[25]For a survey of the uses of Markov modelling for generative music, see [9].

[26]This is a fairly simple back-off model, essentially a special case of the widely used Katz back-off model [114], with a back-off threshold of 0, and no discounting.

13

PAPER 7 – PATTERNS, DUNGEONS AND GENERATORS

Steve Dahlskog, Staffan Björk & Julian Togelius

ABSTRACT

This paper analyses dungeons, of the varieties commonly found in role-playing games, into several sets of design patterns at different levels of abstraction. The analysis focuses on mechanical patterns that could be either straightforwardly instantiated or recognized by a well-defined process. At the most concrete level a set of fundamental components were identified, followed by a long list of micro-patterns which can be directly instantiated. Shorter lists of meso- and macro-patterns, which can be identified mechanically, are also identified. The direct motivation for this analysis is to find building blocks and objectives for a search-based procedural dungeon generator, however we believe the analysis can be useful for understanding this class of game artifacts in general. In particular, the constraints on patterns being instantiable or recognizable leads to a stricter pattern analysis than many other attempts at analyzing game design.

PUBLISHED IN

Proceedings of the 10th International Conference on the Foundations of Digital Games

PATTERNS, DUNGEONS AND GENERATORS

13.1 INTRODUCTION

Design patterns have become an important tool for analyzing and reasoning about game design. They provide a relatively formal way of talking about game design, which is appealing particularly for those who want to automate the analysis and/or generation of game content. In recent research, a method has been devised for procedurally generating platform game levels based on design patterns [48, 50]. This method is based on the idea that design patterns can be ordered into different levels of abstraction, from smaller and more concrete patterns to larger and more abstract patterns. The larger patterns can be instantiated in multiple ways through different combinations of smaller patterns. Levels can then be generated through searching for combinations of smaller patterns that yield certain larger patterns.

This paper addresses the domain of dungeons, the type of levels with a spatial puzzle quality first introduced in *Dungeons & Dragons* [88] and typically found in "roguelikes" such as *Rogue* [232], *Moria* [118], and *Hack* [70], and computer role-playing games (RPGs) such as *Bard's Tale* [106] and *Ultima* [85]. We identify a relatively large set of design patterns at different sizes and levels of abstraction – micro-, meso- and macro-patterns. The motivation for carrying out this analysis is to find patterns that can be used for pattern-based or search-based dungeon generation and for this reason the granularity is finer and the level of detail of the pattern collection is higher than what is typical for pattern collections. We believe the pattern analysis carried out here has value for other purposes as well, such as understanding the design space and design affordances of dungeons as a game artifact.

13.2 RELATED WORK

Due to the approach chosen in this paper the related work is connected to several different research areas related to games. In the following section an overview of these will be presented together with examples of relevant games.

13.2.1 *Game Spaces and Dungeons*

Exploration or movement through spaces are common gameplay features in games. In line with this, Aarseth calls spatiality a defining element of games and argues for a possible use in classifying games according to how the space is implemented in the game [2]. Exploring the concept of Game spaces, Nitsche introduces three concepts relevant to this paper in the form of labyrinths (linear or unicursal), mazes (branching or multicursal) and arenas (open structures with areas of free movement bounded by surrounding enclosement) [161]. Similarly, Aarseth call games where an avatar has to be moved from a starting to end position "place-oriented" and identifies hubs, open landscapes, and uni- or multicursal structures as components of quest-based games [1].

A common game space in fantasy RPGs is the dungeon. The dungeon has been present in fantasy RPGs more or less from its introduction in *Dungeons & Dragons* [88] in 1974 until today. The following description is provided in a later edition of the game: "A Dungeon is a group of rooms and corridors in which monsters and treasures can be found." [89]. This definition is arguably rather open and could include other types of game spaces from other types of games, like bunkers, castles and other buildings in, for example First-Person Shooters. For the purposes of this paper, we accept this definition and the consequence that it might include similar spaces in other games, but we base our analysis only on games that are commonly agreed to fall into the RPG genre.

The ubiquity of dungeons in RPGs indicate that they solve recurring problems in game design, and provide a key part of the player experience. From

a player perspective, it is likely that the rather confined space of the dungeon provides interestingness by allowing exploration of a non-trivial layout of space, and excitement due to incorporated components such as enemies, traps, and treasures. The constraint on player movement can create additional tension and exploring dungeons may in many cases also impose a level of resource management (e.g. considering how much provisions and consumables should be brought or when attempt to resupply should be made). Similarly, it is likely from a designer's viewpoint that explicitly limiting the player's available choices and access to information helps structure the order in which players gains access to the game (and thus the story). By providing access to certain areas only in a specific order, it becomes easier to combine a storyline with the relatively free exploration of non-dungeon parts ("overworld") of many RPGs.

13.2.2 *Design Patterns*

Design patterns is the idea that certain design solutions can be described on an abstracted level so they can both be re-used in similar contexts and casual relations between them can be identified. They were originally developed for architecture by Alexander *et al.* to help end users take part in design processes [7] as part of a large movement focusing on understanding design methods (*cf.* [110]). The use of design patterns for understanding games was first introduced by Kreimeier [122] and then followed by Björk & Holopainen who developed a collection of approximately 300 design patterns [30]. Similar approaches include the 400 rules project [23] and the game ontology project [246]. This paper follows a convention to mark patterns through the use of SMALL CAPS.

While the collection developed by Björk & Holopainen include some patterns appropriate for analyzing the design of dungeons in RPGs, other more specific collections related to this have been developed. Hullett & Whitehead [102] explored the design of levels in First-Person Shooters and created a collection of 10 design patterns which have later been incorporated in the collection initiated by Björk & Holopainen [29]. Smith *et al.* analysed

20 games and the resulting design patterns were grouped into level and quest patterns in RPGs [207][27].

Gameplay design patterns have been combined with the Mechanics-Dynamics-Aesthetics framework [103] to explain how patterns dealing with concrete rules or game elements can cause dynamic behaviors and through this aesthetical experience. This has been used to both explore camaraderie [25] and pottering [133] in games and to compare similarities and difference between the game *X-COM: UFO Defense* [153] and its remake *XCOM: Enemy Unknown* [74, 40].

13.2.3 *Procedural Content Generation*

Procedural content generation (PCG) in games refers to methods for algorithmically creating game content (e.g. games, rules, worlds, levels, items, etc.). PCG might be used independently or to assist a human designer, with human design objectives and partial designs as input. While many early digital games featured some kind of PCG, it has only become an active area of research in academic settings within the last few years [193]. PCG has been used in many games of different genres, but is perhaps most central to games relying on runtime generation dungeons like *Diablo* [34], *Rogue* and *Ultima I*. Even as early as in 1979, methods for generating dungeons were present in both pen-and-paper RPGs (*Advanced Dungeons & Dragons* [87]) as well as in digital games (*Akalabeth: World of Doom* [84]).

A recent paper surveyed procedurally generated dungeons showing previous work from a technical perspective, amount of control over the generative process, the output and results. In the paper, the authors argue for researching dungeons due to its close relation to successful games [239]. Procedural dungeon generation has also been surveyed in the textbook on PCG in games [193]. A wide variety of different methods have been applied to generating dungeons. These include evolutionary algorithms [12, 13, 142, 237, 14], grammar expansion [62, 63], cellular automata [109], constraint solving [94, 199], and various ad-hoc methods such as dungeon diggers and binary space partitioning [193].

13.2.4 *Design Patterns used in PCG*

Given that design patterns are formalizations of game and level design into relatively simple and independent components, it stands to reason that they would be useful in content generators. A content generator is after all exactly a generative formal design theory. Thus, several recent experiments in procedural content generation use the language of design patterns to describe parts of the content generator or the artifacts it produces [71, 166].

Dahlskog *et al.* have taken the metaphor further and developed a search-based level generator based on design patterns on different levels of abstraction [48, 49]. The prototype implementation, which generates levels for *Super Mario Bros* (SMB) [158], uses patterns on the micro-, meso- and macro-levels. Micro-patterns are simple vertical slices of *SMB* levels, meso-patterns are larger features such as enemy hordes or "pipe valleys", and macro-patterns are sequences of meso-patterns. The patterns at each level of abstraction are composed of multiple patterns of a lower level of abstraction. This configurations allows levels to be created through searching the space of sequences of micro-patterns for occurrences of macro-patterns. In effect, micro-patterns are used as building blocks and meso- and macro-patterns as objectives.

13.3 CLASSIFICATION OF DUNGEONS

In order to argue for what a dungeon looks like we surveyed a large number of dungeons in a set of RPGs and gathered empirical data by brute force incremental analysis similar to [31]. The approach focused on identifying patterns based on game space and game mechanics related to dungeons in each game. The approach included actual game playing, primarily on emulators, supported with strategy guides, maps and Youtube clips to minimize playtime outside the actual dungeons (i.e. minimal time was committed to exploring story or solving puzzles). We picked games from an exhaustive source [22] and correlated this source with the game magazine *Computer Gaming World*'s lists of RPGs [184] to have a rich data set. All in

Table 34: Fundamental Components

TILE	The basic unit of space in a dungeon. Individual TILES have Boolean attributes associated with them: *Passable* and *Seethru*.
LEVEL	A rectangular space of TILE.
WALL TILE	The base classification of TILES in a LEVEL. TILES belonging to this category are not *Passable* and not *Seethru*. The BASE CONTENT of a TILE in *Rogue* is "Rock".
GROUND TILE	A classification for TILES that are *Passable* and *Seethru*.
ITEM	A game object that can be in a TILE but which can also be picked up, carried, and dropped in other TILES.
AGENT	A game object that can perform actions, e.g. moving and attacking, and is located on a specific TILE. The player's avatar is an AGENT as are monsters. They typically hinder other from entering the TILE they are in, i.e. they temporary remove the attribute *Passable* from the TILE they are in.
LINE-OF-SIGHT(*A*,*B*)	A Boolean function returning if all TILES on a straight line between point *A* and *B* in a LEVEL are *Seethru*.
TRAVERSABLE (*A*,*B*)	A Boolean function returning if a player can move between point *A* and *B* in a LEVEL. A ROUTE is a traversal solution and the length of different ROUTES may be needed for some design patterns.
SEQUENCED (*A*,*B*)	A Boolean function returning if point *A* must be visited before point *B* in a LEVEL.

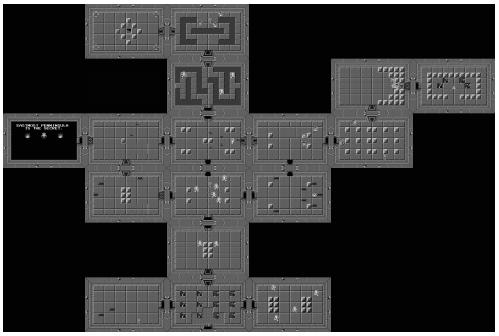


Figure 69: A dungeon in The Legend of Zelda (*Connected Rooms*).

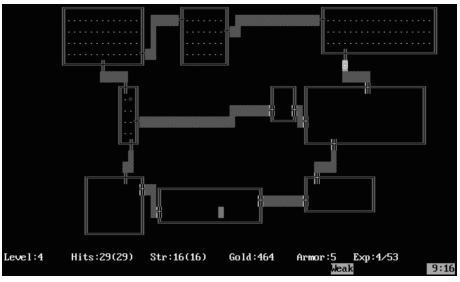


Figure 70: A dungeon in Rogue (*Rooms & Corridors*).

all, 91 games released between 1975-1993 were analyzed in detail but several more modern games will be used as examples in the text. The motivation for focusing on our analysis on the period between 1975 and 1993 is due to resource limitations as well as these early games have a stronger focus on actual dungeons whereas modern games have the possibility of adding open world-like areas with the effect that the player spend more time there than in dungeons (*c.f. The Elder Scrolls V: Skyrim* [26]). We intend to add more games to the list in the future as we extend the project with content generation.

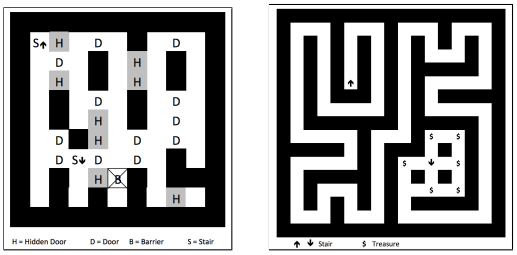


Figure 71: Dungeons in Ultima I (*Maze*) and Ultima II (*Labyrinth*).

Figure 72: A dungeon in Diablo (*Open area*).

In essence, the surveyed games showed similarities on the account of topology and the different dungeons could be classified and grouped into five different types. Commonly the topology could be described as scarce or dense depending on how much traversable game space the dungeon layout contains (dense dungeons have more traversable space).

1. *Connected Rooms* is a type of dungeon that is most common in classic text-based adventure games. The player moves from interesting sites (rooms) without explicit corridors, pathways or tunnels to the next site. Games like *Colossal Cave Adventure* [47] and *Zork* [124, 104] but also *The Legend of Zelda* [160] (see Fig. 69) have such dungeons; they are efficiently mapped with graphs.

2. A *Rooms & Corridors* dungeon is often scarce with a small set of rooms connected with non-branching corridors. Typical examples of

this type of dungeons are present in *Rogue* (see Fig. 70) and roguelike games from the 1980-ies. Corridors are functional game space and events (combat) can take place there. If the game allows the player to dig through walls, graphs will not be sufficient to map the game space and 2D-matrices are needed.

3. *Labyrinths* are unicursal structures with a single path leading through the dungeon. Earlier games like *Ultima II* [86] (see Fig. 71). These dungeons are often dense and demand 2D-matrices for mapping.

4. *Mazes* are multicursal layouts with multiple paths leading through the dungeon. Both earlier and later games have this kind of topology. It is often dense and games like *Akalabeth: World of Doom* and *Ultima I* have these kinds of dungeons (see Fig. 71).

5. *Open area*-dungeons consists of extremely open space (for a dungeon) with obstacles (e.g. thin walls) that hinders free maneuvering, but in comparison with the other types of dungeons, the tactical maneuvering have greater importance. Corridors are uncommon and if one considers the traversable space of these dungeons they are very dense. Games like *Telengard* [123] and *Diablo* (see Fig. 72) are typical for this kind of dungeons.

13.4 PATTERNS

After surveying the 91 games, the next step was to identify design patterns within these related to the level, or dungeon, design. However, the domain of the study needed to be delimited to a manageable size and an emphasis was placed on the *Rooms & Corridors*, *Labyrinths*, and *Mazes* classifications. More specifically, we decided on studying dungeons as they can appear in two-dimensional games where movement primarily takes place horizontally, and is characterized by exploration in constrained spaces and progression. Further, we restricted ourselves to games where the dungeons can be specified as combinations of multiple tiles, though movement between the

tiles may or may not be pseudo-continuous. This definition includes dungeons in classic CRPGs such as early games in the *Legend of Zelda* series and *Final Fantasy* [215] series as well as roguelikes like *Rogue*, *Nethack* [219] and *Diablo*. We are excluding games with a large amount of vertical moment, in particular platformers like *Super Mario Bros*, and games which prominently feature three-dimensional exploration like *Tomb Raider* [46]. Furthermore, we are looking at patterns that exist in multiple games, not just one or a few. Even so, all the 91 games analyzed continued to be used as sources when possible.

Design patterns provide abstract descriptions of solutions to common problems. Although this abstraction can easily be contextualized by human designers given a specific design problem, the same does not apply to systems that procedurally generate solutions. The strategy we applied to bridge this issue was to introduce several layers of interrelated patterns where the lowest levels have a low degree of abstraction and therefore can be easily instantiated or recognized by an algorithm, making the patterns useful as part of a procedural dungeon generator. Previously Dahlskog & Togelius [48, 49] and Ferreira & Toledo [71] have done so in other domains. As a result, the pattern analysis is here in some senses more rigorous and formal than similar pattern analyses found elsewhere, but also more limited. In relation to the different levels of the Mechanics-Dynamics-Aesthetics (MDA) framework [103], the focus was squarely on mechanical patterns, omitting dynamical or aesthetic ones.

As the immediate motivation for this study is to find a set of patterns that can be used for content generation, we needed to place constraints on the patterns we found. A first constraint we set up was to use a set of fundamental components and that patterns should be described in terms of these components and other patterns. This implied a hierarchy of patterns which lead to a classification of patterns as either micro-patterns, meso-patterns or macro-patterns similar to what was done earlier by Dahlskog *et al.* [48]. The relation between these is that meso-patterns can be built out of configurations of multiple micro-patterns, and macro-patterns can be built out of combinations of micro- and meso-patterns. Micro-patterns have the

further constraint that they should be *mechanically instantiable*: it should be possible for a constructive algorithm to instantiate any of those patterns at any given position in a tile by simply "dropping it in" without analyzing more than the immediately neighboring tiles.

If should be noted that the collection presented here is non-exhaustive. The patterns presented are however sufficient to create the dungeons with all the main features found in RPG dungeons. When reading the tables, note that most entities are classified as being more specific versions of other entities and this is indicated by listing the more general entity in parenthesis after an entity name.

13.4.1 *Fundamental Components*

For the context of this paper, two Boolean attributes are relevant for TILES: *Passable* and *Seethru*. *Passable* TILES can be both entered and left while *Passable* are those that can be seen through as well as allow ranged attacked to pass through them. These and several other basic concepts can be found in table 34.

13.4.2 *Micro-patterns*

The Micro-patterns introduced here make use of *types* that describes categories of Micro-patterns. For example, all patterns classified as *Space* create areas where players can move and individual Micro-Patterns make use of this type to describe how they can be combined with other patterns to create larger areas where players can move around. A list of identified micro-patters can be found in table 35–37.

13.4.3 *Meso-patterns*

While Meso-patterns are also mechanical in the sense that they can be observed from static instances of a game in progress (and thus identified by a static evaluation function), they are abstract in the sense that their existence

Table 35: Micro-patterns part 1.

SPACE	A SPACE is a group of connected TILES that share the same attributes and optionally other design features such as topological properties or gameplay functionality. The simplest SPACE is simply one TILE.
CORRIDOR (Space)	A CORRIDOR is a series of horizontal or vertical GROUND TILE. The end points of a CORRIDOR can be connected to other *Spaces*. In *Rogue* CORRIDORS connect ROOMS but in other games like *Ultima I* (see Fig. 71) CORRIDORS is the main spatial component and ROOMS are missing.
CONNECTOR (Space)	A CONNECTOR is a 1 GROUND TILE long CORRIDOR that is used to let passages in dungeons turn or allow intersections by being connected to other *Spaces*.
ROOM (Space)	A ROOM consists of several GROUND TILES but wider than a CORRIDOR and allow for more freedom in movement.
DOOR (Space)	The DOOR is a barrier that has two states; "open" or "closed". Open doors are *Passable* and *Seethru* while closed door are neither. Connection to other *Spaces* are typically in a north-sound or west-east direction. In *Diablo* ENEMIES does not notice the player character through DOORS. Some DOORS have a connected key that allow for a third state "locked". Some games have "breakable" DOORS which puts the door in a constant "open" state.
HIDDEN DOOR (Space)	HIDDEN DOOR is a DOOR that functions like a WALL until it has been revealed.
KEY (Item) [207]	A *Key* is a *Item* allowing the ability of altering any or a specific DOOR's state regarding being "locked".

Table 36: Micro-patterns part 2.

PROPS (Space)	PROPS are GROUND TILES with decorations but no extra functionality. Doors in *Rogue* are PROPS!
OBSTACLES (Item)	OBSTACLES occupy TILES and hinder AGENTS from entering them. They can be modeled as ITEMS that cannot be picked up and make TILES temporary lose any *Passable* attribute like AGENTS but can also be modeled as AGENTS that cannot perform actions. They may be destroyable or movable. Examples include boulders in *Rogue*.
INSTALLATIONS (Props)	INSTALLATIONS are PROPS (or OBSTACLES) that allow actions, typically by the players' avatars when they are next to them, but INSTALLATIONS can also provide actions to other AGENTS or be activated by the game system. Example includes fountains in roguelikes and mana pools in *Diablo*.
CONTAINERS (Installation) [207]	CONTAINERS allow caches of ITEMS to be accessed from the TILES they are placed in. While they typically are INSTALLATIONS they can also be ITEMS. Like DOORS, CONTAINERS may be locked so possession of KEYS or abilities to pick locks may be need to have access to the ITEMS inside CONTAINERS. *The Legend of Zelda: A Link to the Past* [159] has *Chests* that need KEYS to be unlocked while *Nethack* provides a multitude of ways to open locked *Chests*, none which involve KEYS.

Table 37: Micro-patterns part 3.

STAIRS (Installation)	STAIRS are INSTALLATIONS that allow movement away from the current LEVEL. This is typically to another LEVEL which requires the position of a entry point on that LEVEL. When movement back is possible the natural design solution is to have STAIRS on the entry point which leads back to the original STAIRS TILE. If this is not possible, a PROP indicating the entry point can provide diegetic consistency [29]. In several games this vertical movement affects the difficulty or strength of the opposing ENEMY.
IMPASSABLE SPACE (Space)	An IMPASSABLE SPACE a group of connected TILES that are not *Passable* but *Seethru* and thereby allows LINE-OF-SIGHT through them. In *Diablo* the lava lakes works as IMPASSABLE TERRAIN.
TRAPS (Installations) [207]	TRAPS are typically hidden and perform a one-time attack on AGENTS entering the TILE they are placed in. Traps are typically hidden until activated or revealed, and may be disarmed when revealed. They are typically modeled as INSTALLATIONS with automatic activation but could also be created as AGENTS that cannot move. Examples of TRAPS include bear traps in *Rogue* and pits in *Nethack*.
ENEMIES (Agents)	ENEMIES are simply AGENTS whose primary behavior is to attack players' avatars. A main dichotomy regarding ENEMIES are if they can move or not.
SPAWN POINTS (Installation) [207]	SPAWN POINTS are INSTALLATIONS from which ENEMIES appear. Generators in *Gauntlet* [15] are SPAWN POINTS.
PORTALS (Installations) [207]	PORTALS allow instantaneous movement between to spatially separated points in a level. They can be modeled as STAIRS which may be either PROPS or OBSTACLES and can either work only in direction or both directions. The *Bard's Tale* games include PORTALS in their dungeon design.

relies on combinations of Micro-patterns. Many of the patterns mentioned in this section (and the next) have already been documented as patterns earlier (*cf.* [29, 207, 102]), although in some cases under other names. However, those descriptions were not specific enough to be the basis for an intended dungeon generator so alternative versions are presented here. Our list of identified meso-patterns can be found in table 38 and 39.

13.4.4 *Macro-patterns*

The highest level of abstraction used in this collection are Macro-Patterns. These are pattern defined through the use of the fundamental components and lower-level patterns, typically focusing on longer periods of gameplay or — somewhat paradoxically — specific gameplay aspects that depend on a combination of circumstances. Like in the case of Meso-Patterns, they have already been identified in a variety of other games. Table 40 contains our list of macro-patterns.

13.5 DISCUSSION AND CONCLUSIONS

The patterns presented above are only a sample of the possible patterns for procedurally generating dungeons. Many more, e.g. ARENAS, BOSS MONSTER DUNGEON, MULTI-LEVEL DUNGEON, and SECRET AREAS, could have been included, though that would require a longer paper format. However, one can ask what the benefits of the presented framework here provides given that several of the games examined already create procedurally generated dungeons. However, the pattern-based content generation approach allows us to build generators that respect design constraints and implement patterns on different levels, unlike the relatively unstructured output of many dungeon generators.

The pattern collection provides an abstract model of level design in "dungeon crawl" RPGs. These patterns were constructed after reviewing 91 games using dungeon so they reflect actual design practice of game designers. As such they present a model usable for both designers and researchers,

Table 38: Meso-Patterns part 1

Choke Points (Space) [29, 102, 207]	Choke Points are Tiles that are the only connections between two different parts of a Level, or in other words: a Tile *C* is a Choke Point if there exists Tiles *A* and *B* so that Traversable(*A*,*B*) is true but all Routes require passing through *C*. Choke Points can be created from Doors or Portals and allow sequences of Sequenced relations to be constructed; a Choke Points *C* between *A* and *B* enforce Sequenced(*A*,*C*) and Sequenced(*C*,*B*). Choke Points larger than Tiles can be constructed through placing Spaces such as Rooms behind other Choke Points.
Special Rooms (Room)	These are Room created together with specific content such as Items or Enemies in them as well as possibly having restrictions on access to them. *Shops* in *Nethack* is an example of a Special Room; it is only accessible through a Choke Point, includes a *Shopkeeper* and have Installations on all Tiles to handle buying and selling Items. *Beehives* is another example from the same game which populate a Room with Enemies in the form of *Killer Bees* and *Queen Bees* together with *Royal Jelly* Items.
Dead Ends (Space)	Dead Ends are locations from which players must move through previously explored areas. The simplest form of Dead End is constructed by placing a Choke Point between a Ground Tile and the rest of a Level. However, other Spaces such as Corridors or Rooms can also be the Dead Ends after a Choke Point. This pattern is subjective to the amount of gameplay that is provided behind the Choke Point, no or little gameplay can make a larger Space into Dead Ends while a small Space with rich gameplay is less of a Dead End (cf. the *Shop* in Special Rooms pattern.)

Table 39: Meso-Patterns part 2

CONDITIONAL PASSAGEWAYS (Choke Point) [29, 207]	CONDITIONAL PASSAGEWAYS are TILES that are only *Passable* when a character has a special skill or *Item*. They need to be CHOKE POINTS to avoid functionally becoming equivalent to OBSTACLES. The possibility of CONDITIONAL PASSAGEWAYS necessitate the consideration of a CONDITIONALLY TRAVERSABLE(*A*,*B*) function. The presence of CONDITIONAL PASSAGEWAYS can make CHOKE POINTS directional in that they only are CHOKE POINTS when moving from *A* to *B* but not when moving from *B* and *A*. CONDITIONAL PASSAGEWAYS can be created through the use of DOORS and KEYS but *Pokémon* [81] provides an example of another solution: a Bicycle is needed to go on "Cycling Road" (Kanto and Sinnoh regions) and Seaside Cycling Road.
ONE-WAY TRAVEL (Route) [29]	This is design solution which makes a ROUTE between *A* and *B* so that TRAVERSABLE(*A*,*B*) is true while TRAVERSABLE(*B*,*A*) is false. This is typically done through PORTALS or STAIRS but the introduction of one-way DOORS is another possibility.
FLANKING ROUTES (Route) [29, 102]	These ROUTES offer alternatives to what is perceived as the most direct ROUTE between SPACE *A* and *B*. They can be created through first creating a ROUTE that is TRAVERSABLE(*A*,*B*) and then create another ROUTE which is less obvious. This latter feature can be achieved by making the ROUTE longer, hiding it through the use of SECRET DOORS, or making it CONDITIONALLY TRAVERSABLE(*A*,*B*) through the use of CONDITIONAL PASSAGEWAYS.

Table 40: Macro-Patterns

QUICK RETURNS [29]	QUICK RETURNS intend to let players explore a part of a LEVEL but offer a quick way of returning to previously explored parts after reaching a certain point. This can be modeled by designing so that TILES *A* and *B* are TRAVERSABLE(*A*,*B*) (or CONDITIONALLY TRAVERSABLE(*A*,*B*)) with a certain minimum length but the solution for TRAVERSABLE(*B*,*A*)) is shorter. This is typically achieved through constructing TRAVERSABLE(*A*,*B*) through a number of CHOKE POINTS but providing ONE-WAY TRAVEL from *B* to *A* or a CONDITIONAL PASSAGEWAY at *B* that activates a shorter CONDITIONALLY TRAVERSABLE(*B*,*A*) than *A* had to *B*. The Portals to the town in *Diablo* and the Castle of Ordeals are examples of QUICK RETURNS.
BACKTRACKING LEVELS [29]	Somewhat misnamed as the pattern can be applied to parts of LEVELS, BACKTRACKING LEVELS denote design solutions where players need to move from TILE *A* to *B* and then return following basically the same ROUTE. BACKTRACKING LEVELS can be constructed from inserting a chain of CHOKE POINTS between *A* to *B* and making *B* part of a DEAD END. BACKTRACKING LEVELS can also be applied to chains of LEVELS, the goal of *Rogue* is to descend through LEVELS until one finds the amulet of Yendor and then ascend back up to the starting point.
SNIPER LOCATIONS [29, 102]	Places advantageous to making ranged attacks against ENEMIES classify as SNIPER LOCATIONS. These can most easily be created by having two SPACES connected by an IMPASSABLE SPACE. However, SNIPER LOCATIONS should be relatively safe also in that ENEMIES cannot quickly reach them. The use of SECRET DOORS, CONDITIONAL PASSAGEWAY or ROUTES of a certain minimum length between the two SPACES can achieve this.

and can support either manually, procedurally generated dungeon designs or a mixed-method approach. Although the pattern collection does not yet reach the level of abstraction used by Aarseth or Nitsche (i.e. [1] and [161]), we believe further work can define labyrinths, mazes, and hubs as patterns and potentially as part of systems for PCG dungeons. A first indication of how such patterns would look can be gleamed from the LABYRINTH and HUB-AND-SPOKE patterns by Smith *et al.* [207].

While the meso- and macro-level patterns have been described so that implementing them should be fairly unproblematic, in several cases alternatives have been provided. This shows how patterns can be implemented in different ways to achieve the same gameplay functionality and offers designers choices to use the most appropriate solutions.

One of the pattern presented, SNIPER LOCATIONS, was actually not found in the examined games. It was included to show how the pattern collection can be extended to support additional types of gameplay (which in this case already exists in other genres) through introducing patterns building on already existing patterns. While this can also be done through adding new fundamental components or micro-patterns, this has a larger risk of fundamentally changing the design so resulting gameplay no longer is seen as being part of the genre.

On a more theoretical level, the model of using Micro-, Meso-, and Macro-patterns show how game design practice can be analyzed in greater detail and be described to a level where different levels of abstract on the "mechanical" level of design patterns can be implemented in code for dungeon RPGs. In future work we aim at showing how Meso-patterns can be given the requirement to be *mechanically recognizable*, *i.e.* it should be possible for an algorithm to recognize all instances of any given meso-pattern in a dungeon through a direct evaluation function. We hope that, despite the limited scope, the added rigor might be useful outside of procedural content generation as well since it shows how the knowledge contained in many previously identified patterns can be rephrased to be usuable more directly in implementation.

Concluding, the collection of patterns presented here provide an overview of level design for dungeons to a level of granularity that supports the design of PCG dungeon systems. While implementation of such a system is the next step in our work, we believe this collection has a value in describing dungeon level design through a tiered model that can also support manual construction of dungeons as well as provide a tool for further analysis of dungeon level designs.

NOTES

[27] http://rpgpatterns.soe.ucsc.edu/

14

PAPER 8 – PLAYER EXPERIENCE EVALUATION OF LEVEL GENERATORS IN THE MARIO AI FRAMEWORK

Steve Dahlskog, Julian Togelius & Paul Davidsson

ABSTRACT

This paper describes a player experience-focused evaluation of several level generators for the Mario AI Framework. Two generators which were recently devised by the authors, one based on n-grams and one based on the design pattern recognition together with search-based generation, are compared with several existing generators from previous studies. Results show that both of the more recent generators produce significantly more enjoyable and Mario-like levels than those produced by previous generators.

SUBMITTED

PLAYER EXPERIENCE EVALUATION OF LEVEL GENERATORS IN THE MARIO AI FRAMEWORK

14.1 INTRODUCTION

Over the last couple of years many methods for automatically generating game levels have been suggested and based on these methods several generators have been developed. In most cases, multiple methods are applicable to the same content generation problem, raising the question which method to choose for tackling that problem. The answer to this question depends on the particular goals one may have for the generation of game content. Moreover, different games have different designs and may differ a lot from each other. For some games, variation of layout is important and for some games the challenge is more important. While many methods can be used to compare and characterize level generators, ultimately one wants to know which level generators are likely to be most favored by actual players of a game. In order to do this, user studies need to be carried out.

In this paper we describe a user study that evaluates and compares five different generators for the popular platform game *Super Mario Bros.* (SMB) [158]. The generators are evaluated by the users according to four criteria; how entertaining the generated level is to play, how challenging it is, how well-made it is, and finally how SMB-like it is.

In the next section, some related work is reviewed. This is followed by a description of the generators used in the experiment. We then describe the experimental set-up and the results from the experiments. Finally, an analysis of the results and a summary of the conclusions are provided.

14.2 BACKGROUND

14.2.1 *Related work*

Game level generation is an area of Procedural Content Generation (PCG) which has over the last couple of years gained an interest in the research community. The research focus has been on different AI-methods to generate several different types of game content including games, game worlds, game levels, items, quests, rules, and textures. The methods are diverse, covering different approaches like grammars [186], agents [58], evolutionary computation [95], answer set programming [199], constraint solving [208] and cellular automata [109].

Automatic level generation is not just an area for academic study, it has a long history in the game industry where games like *Beneath Apple Manor* (1978) [243], *Akalabeth: World of Doom* (1979) [84] and *Rogue* (1980) [232] demonstrates the earliest examples of procedurally generated game levels. The games are set in *dungeons*, where the player explores rooms and corridors to find treasures and combat monsters.

In Game AI and PCG research another influential game has been the center of attention, at least according to the amount of papers written and generators developed, namely the seminal platform game *Super Mario Bros.* [158]. In academic Game AI, the Mario AI framework, developed from *Infinite Mario Bros* (IMB) [172], has been been used in the Mario AI competition both for NPC behaviour [223] and level generation [190]. Several different PCG approaches have been developed for IMB, including a data-driven model that predict different dimensions of player experience [169], design grammar and grammatical evolution [186], design patterns [50], n-grams [52], and constraint solving combined with a mixed-initiative tool [208].

In relation to content generation a set of papers have looked into user studies for evaluation. Dahlskog and Togelius [49] presented a user study comparing three different approaches for their pattern-based approach. Shaker et al. [189] developed an adaptive system that directly asks players

their preferences and the paper describes an experiment where users compares an adaptive with a non-adaptive system. Bakkes et al. [20] present a system that balances the challenges in IMB levels together with a user evaluation.

Another type of evaluation that is commonly used for IMB levels is (computational) metrics-based evaluation, in particular the concept expressive range, initially suggested by Smith and Whitehead [202] is used to measure two key metrics; linearity and leniency. Other metrics based on player experience was suggested by Shaker et al. [186]. Later, Horn et al. [101], compared seven different generators and the original SMB levels with the use of six different expressivity metrics. Out of the seven different generators three generators; Notch, P-Notch and ORE are incorporated into our study.

Of particular relevance to the work presented in this paper, is the study conducted by Mariño et al. [135], who combined computational metrics together with a user-study to compare four generators. Two novel generators were introduced in that paper; the user study saw 37 users playing 1 training level and 4 levels from the Notch, ORE, HCTA+P, and HCTA+R generators.The users grade Enjoyment, Visual Aesthetics and Difficulty on a 7-likert scale for each level where a low value indicates more enjoyable, have better visual aesthetics and more challenging. Unlike Mariño et al. [135], we use preference-based questionnaires, a different set of aesthetic dimensions and a partially different set of level generators.

14.2.2 *Purpose*

Given the large amount of methods for automatic generation of levels, comparative and user experience-based evaluations is of great relevance. Most published studies concern level generation methods, some focus on metrics but few a based on user studies for evaluation purposes and therefore motivates user studies. A possible outcome is the possibility of identifying areas for further study, and to find strengths and weaknesses of a particular method or generator.

Therefore the purpose of the paper is twofold, primarily to evaluate two generators [50, 52] which both use different approaches to analyzing the SMB levels and synthesizing the results of analysis to create new levels, and secondly to relate level generators to each other. In this paper the comparison is made between n-gram, MLLG, ORE, Notch and P-Notch.

14.3 GENERATORS

We have evaluated five different generators: the n-gram Generator (n-**gram**), the Multi-level Level Generator (**MLLG**), the Occupancy-Regulated Extension (**ORE**) generator, the Notch Level Generator (**Notch**) and finally, the Parameterized Notch Level Generator (**P-Notch**), which are described below (see Fig. 73 for examples of levels).The motivation for choosing each generator is as follows; 1) n-gram and MLLG are new and have only been evaluated according to metrics [52, 50], 2) ORE and Notch are used in other metrics-based evaluations [101] and user studies [135], and finally 3) ORE and P-Notch are different. In the study we use the exact same levels that are used by Horn et al. [101].

14.3.1 *n-gram Generator*

The n**-gram** generator [52] applies an n-gram model, an idea that can be traced back to information theory [194, 195] to mimicking the original SMB-game. Dahlskog et al. [52] created training data for their model by slicing up the original game's levels in vertical slices (1 tile wide). The generation of a level then basically works as stringing slices into sequences according to a probabilistic language model that predicts the next slice in a level sequence following a $(n-1)$-order *Markov* model. In this setup for the experiment we use levels that are generated with *tri-grams*.

14.3.2 *Multi-level Level Generator*

The **Multi-level Level Generator** [50] is a pattern-based approach that combines game design patterns [30, 48] and evolutionary computation. The approach uses a representation where human created content (i.e. levels taken from the original game SMB) is cut to vertical slices called *micro-patterns* and then a specific order of these micro-patterns are searched for to make up *meso-pattersn* and thus new levels. The MLLG approach is an extension to pattern-based approaches [49, 51] where the fitness function takes into account the order of the meso-patterns to follow a macro-pattern disposition from levels in the original game. The advantage of this multiple patterns layers is that repeated patterns are avoided.

14.3.3 *Occupancy-Regulated Extension*

The **Occupancy-Regulated Extension** generator (ORE) was initially designed to capture the aspect of human creativity to produce interesting level designs [139] and was submitted to the Level Generation Track of the IEEE CIS-sponsored 2010 Mario AI Championship [190]. The approach includes a general geometry assembly algorithm with human-design-based level authoring where a set of modules of human-made level designs (*chunks*) are pieced together to form new levels. In the initial approach a library of 40 chunks was used to generate IMB levels. ORE is suitable for procedural generation with *mixed-initiative* (*cf.* [208]) since it can use different weights for chunks and different chunk libraries together with a post-processing approach.

14.3.4 *Notch*

The Notch generator (Notch) is the standard generator that comes with the Mario-AI framework since it was part of the original IMB [172]. It generates levels from left to right and it uses simple checks to make sure the levels are playable. It places different components to the levels according to prob-

abilities of different components like number of gaps and the length of the gaps.

14.3.5 *Parameterized Notch*

The Parameterized Notch generator (P-Notch) is described in [185]. This is the standard Notch generator described above, which has been augmented to take several parameters relating to the number and placement of gaps, enemies and items. The levels generated here are based on a systematic varying of the parameters to sample the parameter space evenly; the same settings are used as in [101].

14.4 EXPERIMENT SET-UP

We gathered 1000 generated levels from each generator and placed them in a pool of levels and randomly picked pairs with the only constraint that the same level could not be picked both times for a pairwise comparison. This way, no two participants were likely to play the same levels; the order of presentation of the levels (beyond the first level) were in no way influenced by the type of level generator. We have 32 users playing 1 training level and 20 levels divided on pair-wise comparison resulting in 10 comparisons on four aspects; how challenging it is, how well-made it is, and finally how SMB-like it is. Every participant in the study received 10 pairs (20 levels in total) together with a test level (the first level of the SMB [158] game with one modification – Piranha Plants are present in the pipes of that level to ensure that the level uses the complete vocabulary of level elements available in the Mario AI Framewok). The test level was a mandatory pre-trial-run of the Mario AI Framework where the player could get familiarised with the specifics of the special version of the game. The parameters for for the following generators: ORE, Notch and P-Notch are exactly the same as in [101]. For n-gram it is the same as in [52] ($n = 3$) and for MLLG the same random seed as in [50].

14.4.1 *Users*

We recruited users by asking students present in two different computer labs. All users stated they had previous experience of playing SMB-games by filling out the survey form.One female and 31 male users with an age span from 22 to 35 years (average age 24 years, σ 3.45 years and median 23 years). The users answered the survey questions in Swedish. The survey questions and answers have been translated to English for the purpose of this article.

14.4.2 *Equipment*

The tests were conducted on standardised desktop PCs in a traditional set-up of a university computer lab, on the users own laptop or the evaluators' laptop when the other possibilities were not available. The framework has very low computational requirements, and therefore the particular hardware used is not likely to have any effect on level generation or gameplay.

14.4.3 *Levels*

In order to anonymise the generators and their respective levels, all levels were named "a" or "b" and each pair was numbered from 0 to 9. The data to identify each level, pair and set of pairs were kept on a different computer than the one the levels were tested on to certify that neither users or the authors could check which generator or level were tested.

14.4.4 *Questionnaires*

The questionnaire was *self-administered* but with a test leader present to answer questions and to provide help with the system evaluated. The questionnaire consisted of three parts: instructions, general questions and evaluation questions. The second part contained questions about the user (name, age, average amount of time playing games and which Mario games the

user had played previously from a list of 14 of the platform games with a check-box next to each title.Previously, computational metrics have looked into aspects like how hard the levels were (Leniency) and how much vertical variation the level had (Linearity) [202]. In a user study the users were asked to grade *Enjoyment*, *Visual Aesthetics* and *Difficulty* [135] for IMB-levels. In this paper we are interested in knowing more about the qualities of the new generators and therefore we have chosen to ask about entertainment value, how challenging, how well made and how Mario-like the levels are. Enjoyment and entertaining may overlap from a player perspective but we chose entertaining to include more aspects of game play experience. The third part consisted of 4 questions per pair:

- Which level is the most entertaining one?
- Which level is the most challenging one?
- Which level is the most well made one?
- Which level is the most Mario-like?

The answers consisted of 4 check-boxes for the answers: "Level A", "Level B", "Both A and B", and finally, "None of them".

14.5 RESULTS AND ANALYSIS

32 persons accepted to participate in the experiment[28]. In table 41 the aspect of most entertaining levels by row against column is shown (example: in row 1 the n-gram is seen as most entertaining against MLLG 11 times whereas MLLG is seen as most entertaining against n-gram 13 times).

In the tables, we have highlighted significant results as follows: * to indicate at the 99.9% confidence level, ** equals 99.5%, *** equals 99%, and † equals 95%.

For each of the evaluation questions (which level is most) 1) Entertaining, 2) Challenging, 3) Well made and 4) Mario-like we performed a Binomial test for each pair of compared generators thus allowing the testing of the hypothesis $f = g$ (f and g being different generators) when we determine the

Table 41: Most entertaining levels by row against levels column.

	n-gram	MLLG	ORE	Notch	P-Notch
n-gram	-	11/13	9/12	26/3*	22/3*
MLLG		-	17/3***	14/7	18/0*
ORE			-	16/3***	23/1*
Notch				-	8/1†
P-Notch					-

Table 42: Most challenging levels by row against levels column.

	n-gram	MLLG	ORE	Notch	P-Notch
n-gram	-	10 /14	13/12	26/3*	26/3*
MLLG		-	15/7	18/3**	19/1*
ORE			-	16/4***	22/1*
Notch				-	8/4
P-Notch					-

Table 43: Most well-made levels by row against levels column.

	n-gram	MLLG	ORE	Notch	P-Notch
n-gram	-	7/17†	10/16	22/2*	21/5**
MLLG		-	16/2*	12/8	17/1*
ORE			-	16/5†	18/3**
Notch				-	5/2
P-Notch					-

Table 44: Most Mario-like levels by row against levels column.

	n-gram	MLLG	ORE	Notch	P-Notch
n-gram	-	12/8	19/8†	23/3*	18/8†
MLLG		-	12/4†	10/6	14/5†
ORE			-	12/9	13/4†
Notch				-	8/2
P-Notch					-

Table 45: Compared with the same generator part 1.

Most entertaining levels		
	Both	None
n-gram	5	0
MLLG	2	1
ORE	3	1
Notch	3	1
P-Notch	1	9
Most challenging levels		
	Both	None
n-gram	3	0
MLLG	3	2
ORE	4	1
Notch	2	2
P-Notch	1	9

Table 46: Compared with the same generator part 2.

Most well-made levels		
	Both	None
n-gram	4	0
MLLG	6	4
ORE	1	4
Notch	3	0
P-Notch	2	10
Most Mario-like levels		
	Both	None
n-gram	1	3
MLLG	7	3
ORE	2	5
Notch	3	3
P-Notch	1	10

respondents' preference when trying out two generators i.e. it will indicate whether there is a significant preference between the two generators.

14.5.1 *Entertaining*

The Linear Level Generator [52] was considered more entertaining than the Notch and P-Notch to the 99% confidence level.

The Multi-level Level Generator [50] was considered more entertaining than the ORE (at the 99% confidence level) and P-Notch (at the 99.9% confidence level) but not (!) really against NLG (significant difference only at 85% confidence level).

The ORE generator [139] was considered more entertaining than Notch and P-Notch (at the 99% and 99.9% confidence level respectively). The comparisons between Notch and P-Notch favours Notch (at the 95% confidence level) but the few concrete comparisons with an outcome indicate that the users think that they are very similar in regards to the entertainment aspect.

14.5.2 *Challenging*

The Linear Level Generator was considered more challenging than the Notch and P-Notch with a result indicating a significant difference in preference at the 99.9% confidence level.

The MLLG was preferred against ORE, Notch and P-Notch with a results indicating a significant difference in preference above 90% confidence level (90%, 99.5% and 99.9% respectively). ORE was also considered more challenging than Notch and P-Notch above the 99% confidence level (99% and 99.9% respectively).

14.5.3 *Well made*

The MLLG is considered producing well made levels than n-gram generator, ORE and P-Notch (95%, 99.9% and 99.9%), whereas n-gram generator is

outperforming Notch and P-Notch (99.9% and 99.5%). ORE 95% and 99.5% against Notch and P-Notch.

14.5.4 *Mario-like*

n-gram generator is considered more Mario-like than ORE (95%) Notch (99.9%) and P-Notch (95%). MLLG is considered more Mario-like than ORE (95%) and P-Notch (95%) but not more than Notch, however all comparison pairs sums up to fewer than 20 clear comparisons which in it self indicate that there are some users finding them similar in that aspect. ORE seem more Mario-like only against P-Notch (95%) but also with fewer than 20 clear comparisons.

14.5.5 *Intra-generator comparisons*

In table 45 and 46 the n-gram generator gets "both" more than "none" in 3 out 4 cases. Whereas P-Notch gets "none" in all cases including the highest numbers.

14.6 DISCUSSION

Looking at the aspect of how the generators seem entertaining to the users, the Notch and P-Notch generators tend to provide less entertainment value in comparison to the n-gram, MLLG and ORE generators. When Notch and P-Notch are compared to each other, the results indicate that they are often perceived as equally (both/none) entertaining, challenging, well-made and Mario-like but also that users prefer Notch to P-Notch. If we include the internal comparison P-Notch more often get the verdict "none" (see table 45 and 46) on all judged aspects. Notch, P-Notch and ORE has been compared previously [101] and P-Notch provides fewer threats (high leniency) and many overlapping platforms (high density) for the player.ORE and Notch has been compared both by users and by computational metrics in [135] where the users found Notch more enjoyable and to have better visual aes-

thetics but ORE being more difficult. The difference here could partly be due that [135] use Likert scale ratings, whereas we use preference indications, which have been found to be more reliable [244]. However, the metrics calculated by Mariño et al. [135] showed ORE-generated levels being less linear and more dense and that Notch was slightly more lenient.

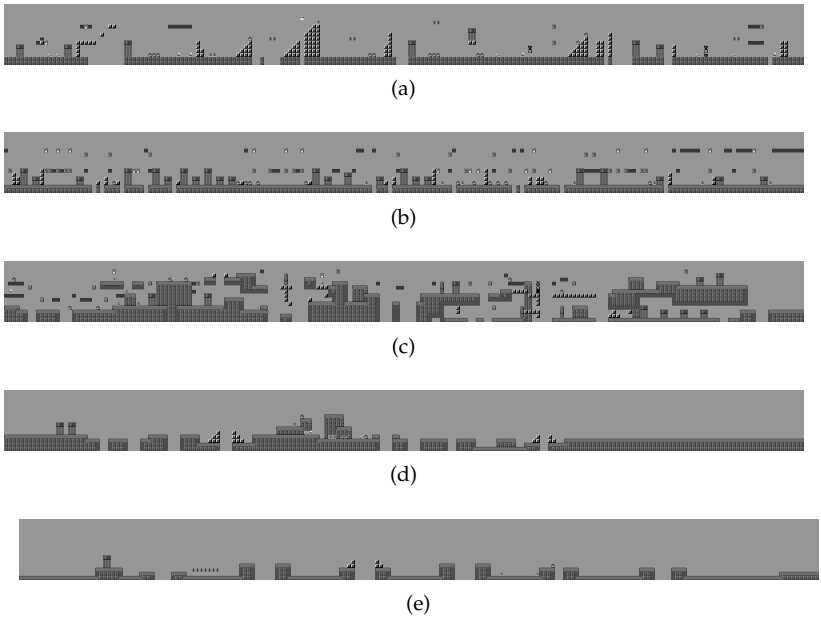
With respect to the intra-generator comparisons, it was often hard for the users to distinguish between levels of the same generator. Exceptions from this were the following comparisons: n-gram (entertaining, challenging levels, well-made), Notch (well-made) and P-Notch (all aspects). However, too few comparisons were made to allow for any strong conclusions.

It stands to reason that the n-gram generator, which mostly copies the micro-structure of the original Super Mario Bros levels, produces the most Mario-like levels of all generators. Given the close resemblance to the original Mario levels, one could have expected that this generator's output would be perceived as less entertaining. However, the results do not appear to bear this out, as n-gram-generated levels were seen as similarly entertaining to ORE- and MLLG-generated levels. This points out that this kind of data-driven level generation, where level generators are trained on a corpus of level content and reproduce some of the structure of those levels, is a promising avenue to pursue further, including using this approach in other domains. For ORE-generated levels we can see that it seem to be perceived to be better than Notch and P-Notch on all the evaluated aspects except for not being decisively more Mario-like than Notch (an aspect shared with MLLG). The intra-generator comparison between Notch and P-Notch indicate that parameter settings is important when tuning the generator, something that further studies could look into.

14.7 CONCLUSION

This paper described the methods and results for an evaluation of the player experience of levels generated by several different level generators for the Mario AI Framework. Two of those generators were previously proposed by the authors. It was found that the new generators, the n-gram generator

and the multilevel pattern-based generator, were both significantly more entertaining and Mario-like than both the default Notch and parameterized Notch generator, as well as the ORE generator. These results validate the two new proposed methods, which both implement different approaches to analyzing the original Super Mario Bros levels and synthesizing the analysis results into new levels. More generally, it suggests that such analysis-based generation could be useful in other procedural content generation domains.



(a)

(b)

(c)

(d)

(e)

Figure 73: Example of levels: a) n-gram, b) MLLG, c) ORE, d) Notch and e) P-Notch.

NOTES

[28]One person did only answer three out of four questions for two pairs (Mario-like for the first pair and entertaining for the 6th pair, both these questions were comparisons between the same generator but different levels). All other participants answered all questions.

BIBLIOGRAPHY

[1] Espen Aarseth. From Hunt the Wumpus to Everquest: Introduction to Quest Theory. In *Proceedings of the 4th International Conference on Entertainment Computing*, ICEC'05, pages 496–506, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29034-6, 978-3-540-29034-6. doi: 10.1007/11558651_48.

[2] Espen Aarseth. Allegories of space: The question of spatiality in computer games. In Friedrich von Borries, Steffen P. Walz, and Matthias Böttger, editors, *Space Time Play: Synergies Between Computer Games, Architecture and Urbanism: the Next Level*, pages 44–55. BirkHäuser, 2007.

[3] Acornsoft. Elite. [Digital game], 1984.

[4] Kirsten Acuna. 'Grand Theft Auto V' Cost More To Make Than Nearly Every Hollywood Blockbuster Ever Made. [WWW], September 2013. URL `http://www.businessinsider.com/gta-v-cost-more-than-nearly-every-hollywood-blockbuster-2013-9?IR=T`.

[5] E. Adams and J. Dormans. *Game Mechanics: Advanced Game Design*. Voices That Matter. Pearson Education, Limited, 2012. ISBN 9780321820273.

[6] Tarn Adams. Slaves to Armok: God of Blood Chapter II: Dwarf Fortress. [Digital game], August 2006.

[7] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, U.S.A., 1977.

[8] Valter Alves and Licinio Roque. Design Patterns in Games: the case for Sound Design. In *Proceedings of the Second Workshop on Design Patterns in Games*, DPG '13, May 2013.

[9] Charles Ames. The Markov process as a compositional model: A survey and tutorial. *Leonardo*, 22(2):175–187, 1989.

[10] Erik Andersen. Optimizing Adaptivity in Educational Games. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 279–281, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1333-9. doi: 10.1145/2282338.2282398.

[11] James Arnold and Rob Alexander. Testing Autonomous Robot Control Software Using Procedural Content Generation. In *Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security*, volume 8153 of *SAFECOMP 2013*, pages 33–44, New York, NY, USA, 2013. Springer-Verlag New York, Inc. ISBN 978-3-642-40792-5. doi: 10.1007/978-3-642-40793-2_4.

[12] D. Ashlock, C. Lee, and C. McGuinness. Simultaneous Dual Level Creation for Games. *Computational Intelligence Magazine, IEEE*, 6(2): 26–37, May 2011. ISSN 1556-603X. doi: 10.1109/MCI.2011.940622.

[13] D. Ashlock, C. Lee, and C. McGuinness. Search-Based Procedural Generation of Maze-Like Levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, 2011. ISSN 1943-068X. doi: 10.1109/TCIAIG.2011.2138707.

[14] Daniel Ashlock and Cameron McGuinness. Automatic Generation of Fantasy Role-playing Modules. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games*. IEEE, August 2014.

[15] Atari Games. Gauntlet. [Digital game], 1985.

[16] Atari Inc. Pong. [Digital game], 1972.

[17] Atari Inc. Asteroids. [Arcade game], 1979.

[18] Avalanche Studios. Just Cause. [Digital game], March 2006.

[19] Avalanche Studios. Just Cause 2. [Digital game], March 2010.

[20] S. Bakkes, S. Whiteson, G. Li, G. V. Vişniuc, E. Charitos, N. Heijne, and A. Swellengrebel. Challenge balancing for personalised game spaces. In *IEEE Games Media Entertainment 2014*, Oct 2014. doi: 10.1109/GEM.2014.7047971.

[21] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-510-X.

[22] Matt Barton. *Dungeons and Desktops: The History of Computer Role-playing Games*. A K Peters Ltd, 2008. ISBN 1568814119.

[23] H. Barwood and N. Falstein. 400 Rules Project. Web page, February 2015. URL `http://gameonwebdesign.com/400project.html`.

[24] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. Technical Report AIM 239, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1972. URL `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-239.pdf`.

[25] Karl Bergström, Staffan Björk, and Sus Lundgren. Exploring aesthetical gameplay design patterns: Camaraderie in four games. In Artur Lugmayr, Heljä Franssila, Olli Sotamaa, Christian Safran, and Timo Aaltonen, editors, *Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments*, MindTrek '10, pages 17–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0011-7. doi: 10.1145/1930488.1930493.

[26] Bethesda Game Studios. The Elder Scrolls V: Skyrim. [Digital game], 2013.

[27] Bethesda Softworks. The Elder Scrolls II: Daggerfall. [Digital game], August 1996.

[28] BioWare. Neverwinter Nights. [Digital game], 2002.

[29] S. Björk. Game Design Patterns 2.0. Web page, March 2013. URL http://gdp2.tii.se/.

[30] S. Björk and J. Holopainen. *Patterns in Game Design*. Charles River Media game development series. Charles River Media, 2005. ISBN 9781584503545.

[31] Staffan Björk, Sus Lundgren, and Jussi Holopainen. Game Design Patterns. In *Proceedings of the 2003 DiGRA International Conference: Level Up*, 2003.

[32] Blizzard Entertainment. StarCraft. [Digital game], March 1998.

[33] Blizzard Entertainment. World of Warcraft. [Digital game], March 2004.

[34] Blizzard North. Diablo. [Digital game], 1996.

[35] Barry W. Boehm. Understanding and controlling software costs. *Journal of Parametrics*, 8(1):32–68, 1988. doi: 10.1080/10157891.1988.10472819.

[36] H. J. Bremermann. Optimization Through Evolution and Recombination. In M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, editors, *Self-Organizing Systems*. Spartan Books, 1962.

[37] Cameron Browne. Elegance in Game Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):229–240, 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2197621.

[38] Cameron Browne and Frederic Maire. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2041928.

[39] Edmund K. Burke, James P. Newall, and Rupert F. Weare. Initialization Strategies and Diversity in Evolutionary Timetabling. *Evolutionary Computation*, 6(1):81–103, March 1998. ISSN 1063-6560. doi: 10.1162/evco.1998.6.1.81.

[40] Alessandro Canossa, Staffan Björk, and Mark J. Nelson. X-COM: UFO Defense vs. XCOM: Enemy Unknown— using gameplay design patterns to understand game remakes. In *Proceedings of the Ninth International Conference on the Foundations of Digital Games*, 2014.

[41] Daniel Cermak-Sassenrath. Experiences with design patterns for old-school action games. In *Proceedings of The 8th Australasian Conference on Interactive Entertainment: Playing the System*, IE '12, pages 14:1–14:9, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1410-7. doi: 10.1145/2336727.2336741.

[42] K. Compton and M. Mateas. Procedural Level Design for Platform Games. In *Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference*, 2006.

[43] Kate Compton, Joe Osborn, and Michael Mateas. Generative Methods. In *Proceedings of the 2013 Workshop on Procedural Content Generation in Games*, 2013.

[44] Michael Cook and Simon Colton. Ludus Ex Machina: Building A 3D Game Designer That Competes Alongside Humans. In Simon Colton, Dan Ventura, Nada Lavrač, and Michael Cook, editors, *Proceedings of the Fifth International Conference on Computational Creativity*, pages 54–62, Ljubljana, Slovenia, June 2014.

[45] Michael Cook, Simon Colton, and Alison Pease. Aesthetic Considerations for Automated Platformer Design. In *AIIDE*, 2012.

[46] Core Design. Tomb Raider. [Digital game], 1996.

[47] William Crowther. Colossal Cave Adventure. [Digital game], 1976.

[48] Steve Dahlskog and Julian Togelius. Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, DPG '12, pages 1:1–1:8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1854-9. doi: 10.1145/2427116.2427117.

[49] Steve Dahlskog and Julian Togelius. Patterns as Objectives for Level Generation. In *Proceedings of the Second Workshop on Design Patterns in Games*, DPG '13, May 2013.

[50] Steve Dahlskog and Julian Togelius. A Multi-level Level Generator. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games*, pages 389–396. IEEE, August 2014.

[51] Steve Dahlskog and Julian Togelius. Procedural Content Generation Using Patterns as Objectives. In Antonio M. Mora Anna I. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, volume 8602 2014 of *Lecture Notes in Computer Science*, pages 325–336. Springer-Verlag, 2014.

[52] Steve Dahlskog, Julian Togelius, and Mark J. Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference*, pages 200–206, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3006-0. doi: 10.1145/2676467.2676506.

[53] Steve Dahlskog, Staffan Björk, and Julian Togelius. Patterns, Dungeons and Generators. In *Proceedings of the 10th International Conference on Foundations of Digital Games*, FDG '15, 2015.

[54] Isaac M. Dart, Gabriele De Rossi, and Julian Togelius. SpeedRock: procedural rocks through grammars and evolution. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 8:1–8:4, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0872-4. doi: 10.1145/2000919.2000927.

[55] H. Desurvire, M. Caplan, and J. Toth. Using Heuristics to Evaluate the Playability of Games. In *CHI 2004 Extended Abstracts on Human Factors in Computing Systems*, April 2004.

[56] EdsgerW. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer New York, 1982. ISBN 978-1-4612-5697-7. doi: 10.1007/978-1-4612-5695-3_12.

[57] Dajana Dimovska, Douglas Wilson, Kennett Wong, Lars Bojsen-Moeller, Lau Korsgaard, Mads Lyngvig, and Robin Di Capua. Dark Room Sex Game. [Digital game], 2008.

[58] Jonathon Doran and Ian Parberry. Controlled Procedural Terrain Generation Using Software Agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 2010.

[59] Jonathon Doran and Ian Parberry. Towards procedural quest generation: A structural analysis of RPG quests. Technical Report LARC–2010–02, Laboratory for Recreational Computing, Dept. of Computer Science & Engineering, Univ. of North Texas, May 2010.

[60] Jonathon Doran and Ian Parberry. A prototype quest generator based on a structural analysis of quests from four mmorpgs. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0872-4. doi: 10.1145/2000919.2000920.

[61] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, 2011. ISSN 1943-068X. doi: 10.1109/TCIAIG.2011.2149523.

[62] Joris Dormans. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1–1:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814257.

[63] Joris Dormans and Stefan Leijnen. Combinatorial and exploratory creativity in procedural content generation. In *Proceedings of the 2013 Workshop on Procedural Content Generation in Games*, 2013.

[64] Christopher Dristig Stenström and Staffan Björk. Understanding Combat Design in Computer Role-Playing Games. In *Proceedings of the Second Workshop on Design Patterns in Games*, DPG '13, May 2013.

[65] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2 edition, 2015. ISBN 978-3-662-44874-8.

[66] Entertainment Software Association. Essential facts about the U.S. computer and video game industry, 2013. URL http://www.theesa.com/facts/pdfs/ESA_EF_2013.pdf.

[67] Entertainment Software Association. Essential facts about the computer and video game industry, 2015. URL http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf.

[68] Evolutionary Games. Galactic Arms Race. [Digital game], 2010.

[69] Farbrausch. .kkrieger. [Digital game], 2004.

[70] Jay Fenlason, Kenny Woodland, Mike Thome, Jonathan Payne, Andries Brouwer, and Don Kneller. Hack. [Digital game], 1982-1985.

[71] Lucas Ferreira and Claudio Toledo. A search-based approach for generating angry birds levels. In *Proceedings of the 9th IEEE International Conference on Computational Intelligence in Games*, 2014. doi: 10.1109/CIG.2014.6932912.

[72] Firaxis Games. Sid Meier's Alpha Centauri. [Digital game], 1999.

[73] Firaxis Games. Civilization IV (PC game), 2005.

[74] Firaxis Games. XCOM: Enemy Unknown. [Digital game], 2012.

[75] Firebird. The Sentinel. [Digital game], 1986.

[76] Gerhard Fischer, Kumiyo Nakakoji, Jonathan Ostwald, Gerry Stahl, and Tamara Sumner. Embedding computer-based critics in the contexts of design. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 157–164, New York, NY, USA, 1993. ACM. ISBN 0-89791-575-5. doi: 10.1145/169059.169133.

[77] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966.

[78] Tracy Fullerton. *Game Design Workshop - A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, New York, U.S.A., second edition, 2008.

[79] Tracy Fullerton. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. A K Peters/CRC Press, New York, U.S.A., third edition, 2014.

[80] S. Gallagher and S. H. Park. Innovation and Competition in Standard-Based Industries: A Historical Analysis of the U.S. Home Video Game Market. *IEEE Transactions on Engineering Management*, 49(1), February 2002.

[81] Game Freak. Pokémon Red/Blue Version. [Digital game], 1996.

[82] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, U.S.A., 1994.

[83] Gareth Bourn. No Man's Sky. [Digital game], June 2016.

[84] Richard Garriott. Akalabeth: World of Doom. [Digital game], 1979.

[85] Richard Garriott. Ultima. [Digital game], 1981.

[86] Richard Garriott. Ultima II: The Revenge of the Enchantress. [Digital game], 1982.

[87] Gary Gygax. Dungeon Masters Guide (sic!). [Role-playing game], 1979.

[88] Gary Gygax and Dave Arneson. Dungeons & Dragons. [Role-playing game], 1974.

[89] Gary Gygax and Dave Arneson and Frank Mentzer. Dungeons & Dragons Set 1: Basic Rules. [Role-playing game], 1983.

[90] Gearbox Software. Borderlands. [Digital game], 2009.

[91] Gearbox Software. Borderlands 2. [Digital game], 2012.

[92] G.A. Gorry and M.S. Scott-Morton. A Framework for Management Information Systems. *Sloan Management Review*, 13(1):55–70, 1971.

[93] J. Gregory. *Game Engine Architecture*. Taylor & Francis, 2nd edition, 20014. ISBN 9781466560017.

[94] K. Hartsook, A. Zook, S. Das, and M.O. Riedl. Toward supporting stories with procedurally generated game worlds. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 297–304, Aug 2011. doi: 10.1109/CIG.2011.6032020.

[95] Erin J. Hastings, Ratan K. Guha, and Kenneth O. Stanley. Evolving content in the Galactic Arms Race video game. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, pages 241–248. IEEE, 2009. ISBN 978-1-4244-4814-2.

[96] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural Content Generation for Games: A Survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, 2013. ISSN 1551-6857. doi: 10.1145/2422956.2422957.

[97] R. Hevner, A., T. March, S., J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28:75–105, 2004.

[98] Lejaren A. Hiller and Robert A. Baker. Computer Cantata: An investigation of compositional procedure. *Perspectives of New Music*, 3:62–90, 1964.

[99] John H. Holland. Genetic Algorithms and the Optimal Allocation of Trials. *SIAM Journal on Computing*, 2(2):88–105, 06 1973.

[100] John H. Holland. Erratum: Genetic Algorithms and the Optimal Allocation of Trials. *SIAM Journal on Computing*, 3(4):326, 12 1974.

[101] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. A Comparative Evaluation of Procedural Level Generators in the Mario AI Framework. In *Proceedings of the 9th International Conference on Foundations of Digital Games*, FDG '14, 2014.

[102] Kenneth Hullett and Jim Whitehead. Design Patterns in FPS Levels. In *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 78–85, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-937-4. doi: 10.1145/1822348.1822359.

[103] Robin Hunicke, Marc Leblanc, and Robert Zubek. MDA: A Formal Approach to Game Design and Game Research. In *In Proceedings of the Challenges in Games AI Workshop, Nineteenth National Conference of Artificial Intelligence*, pages 1–5. AAAI Digital Library, 2004.

[104] Infocom. Zork 1. [Digital game], 1980.

[105] Interactive Data Visualization, Inc. Speedtree. [Software], 2011.

[106] Interplay Productions. Tales of the Unknown, Volume I: The Bard's Tale. [Digital game], 1985.

[107] Introversion Software. Darwinia. [Digital game], March 2005.

[108] Blake Ives, Scott Hamilton, and Gordon B. Davis. A framework for research in computer-based management information systems. *Management Science*, 26(9):910–934, 1980. doi: 10.1287/mnsc.26.9.910.

[109] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 10:1–10:4, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814266.

[110] J.C. Jones. *Design Methods*. Architecture Series. Wiley, 1992. ISBN 9780471284963.

[111] Anna Jordanous. Evaluating evaluation: Assessing progress in computational creativity research. In *Proceedings of the second international conference on computational creativity (ICCC-11). Mexico City, Mexico*, pages 102–107, 2011.

[112] D. Jurafsky and J.H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall series in artificial intelligence. Pearson Prentice Hall, 2nd edition, 2008. ISBN 9780131873216.

[113] S. Karakovskiy and J. Togelius. The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2188528.

[114] Slava M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401, 1987.

[115] Aphra Kerr. *The Business and Culture of Digital Games: Gamework and Gameplay*. Sage Publications, Inc., 2006. ISBN 9781412900478.

[116] 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, 2012. doi: 10.1109/CIG.2012.6374174.

[117] Rilla Khaled, Mark J. Nelson, and Pippin Barr. Design Metaphors for Procedural Content Generation in Games. In *Proceedings of the 2013 ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1509–1518, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466201.

[118] Robert Alan Koeneke and Jimmey Wayne Todd. Moria. [Digital game], 1994.

[119] R. Koster. *Theory of Fun for Game Design*. O'Reilly Media, 2004. ISBN 144931497X.

[120] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

[121] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11189-6.

[122] Brend Kreimeier. The case for game design patterns. 2002. URL `http://www.gamasutra.com/features/20020313/kreimeier_00.htm`.

[123] Daniel Lawrence. Telengard. [Digital game], 1982.

[124] P.D. Lebling, M.S. Blank, and T.A Anderson. Special Feature Zork: A Computerized Fantasy Simulation Game. *Computer*, 12(4):51–59, April 1979. ISSN 0018-9162. doi: 10.1109/MC.1979.1658697.

[125] Kwong-Sak Leung and Yong Liang. Adaptive elitist-population based genetic algorithm for multimodal function optimization. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *GECCO*, volume 2723 of *Lecture Notes in Computer Science*, pages 1160–1171. Springer, 2003. ISBN 3-540-40602-6.

[126] Chris Lewis, Noah Wardrip-Fruin, and Jim Whitehead. Motivational game design patterns of 'ville games. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 172–179, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1333-9. doi: 10.1145/2282338.2282373.

[127] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul MB Vitányi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, 2004.

[128] Antonios Liapis and Georgios N. Yannakakis. Refining the paradigm of sketching in ai-based level design. In *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 2015.

[129] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Generating map sketches for strategy games. In *Proceedings of Applications of Evolutionary Computation*, volume 7835, LNCS, pages 264–273. Springer, 2013.

[130] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the 8th Conference on the Foundations of Digital Games*, pages 213–220, 2013.

[131] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Towards a generic method of evaluating game levels. In *Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 2013.

[132] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Designer modeling for sentient sketchbook. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2014.

[133] Sus Lundgren and Staffan Björk. Neither playing nor gaming: pottering in games. In Magy Seif El-Nasr, Mia Consalvo, and Steven K. Feiner, editors, *FDG*, pages 113–120. ACM, 2012. ISBN 978-1-4503-1333-9.

[134] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4): 251–266, 1995. ISSN 0167-9236. doi: 10.1016/0167-9236(94)00041-2.

[135] Julian Mariño, Willian Reis, and Levi Lelis. An Empirical Evaluation of Evaluation Metrics of Procedurally Generated Mario Levels. In Arnav Jhala and Nathan Sturtevant, editors, *AIIDE*, pages 44–50. The AAAI Press, 2015.

[136] Glenn A. Martin and Charles E. Hughes. A Scenario Generation Framework for Automating Instructional Support in Scenario-based Training. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 35:1–35:6, San Diego, CA, USA, 2010. Society for Computer Simulation International. ISBN 978-1-4503-0069-8. doi: 10.1145/1878537.1878574.

[137] Glenn A. Martin, Charles E. Hughes, Sae Schatz, and Denise Nicholson. The Use of Functional L-systems for Scenario Generation in Serious Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 6:1–6:5, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814262.

[138] Richard O. Mason and Ian I. Mitroff. A program for research on management information systems. *Management Science*, 19(5):475–487, 1973. doi: 10.1287/mnsc.19.5.475.

[139] Peter Mawhorter and Michael Mateas. Procedural Level Generation Using Occupancy-Regulated Extension. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, 2010.

[140] Maxis. Spore. [Digital game], 2008.

[141] Cameron McGuinness and Daniel Ashlock. Decomposing the level generation problem with tiles. In *IEEE Congress on Evolutionary Computation*, pages 849–856. IEEE, 2011.

[142] Cameron McGuinness and Daniel Ashlock. Decomposing the level generation problem with tiles. In *IEEE Congress on Evolutionary Computation*, pages 849–856. IEEE, 2011.

[143] Morgan McGuire and Odest Chadwicke Jenkins. *Creating Games: Mechanics, Content, and Technology*. CRC Press, 2008. ISBN 9781439865927.

[144] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative Design Patterns for Computer

Role-Playing Games. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 88–99, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2131-2. doi: 10.1109/ASE.2004.63.

[145] Matthew McNaughton, James Redford, Jonathan Schaeffer, and Duane Szafron. Patttern-Based AI Scripting Using ScriptEase. In Yang Xiang and Brahim Chaib-draa, editors, *Advances in Artificial Intelligence*, volume 2671 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40300-5. doi: 10.1007/3-540-44886-1_6.

[146] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*. Springer-Verlag, London, UK, UK, 1996. ISBN 3-540-60676-9.

[147] MicroProse. Civilization. [Digital game], 1991.

[148] MicroProse. Sid Meier's Colonization. [Digital game], 1994.

[149] Ian Millington and John Funge. *Artificial Intelligence for Games*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009. ISBN 0123747317, 9780123747310.

[150] Mojang. Minecraft. [Digital game], May 2009.

[151] N. Montfort and I. Bogost. *Racing the Beam: The Atari Video Computer System*. Platform Studies. MIT Press, 2009. ISBN 9780262012577.

[152] Richard Moss. 7 uses of procedural generation that all developers should study. [Web page], January 2016. URL `http://www.gamasutra.com/view/news/262869/7_uses_of_procedural_generation_that_all_developers_should_study.php`.

[153] Mythos Games. UFO: Enemy Unknown (marketed as X-COM: UFO Defense in NA). [Digital game], 1994.

[154] Mark Nelson and Adam Smith. Asp with applications to mazes and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[155] Allen Newell and Herbert A Simon. Computer science as Empirical Inquiry: Symbols and Search. *Communications of the ACM*, 19(3):113–126, 1976.

[156] Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer, 2009.

[157] Nintendo. Donkey Kong. [Arcade game], 1981.

[158] Nintendo. Super Mario Bros. [Digital game], 1985.

[159] Nintendo EAD. The Legend of Zelda: A Link to the Past. [Digital game], 1991.

[160] Nintendo R&D4. The Legend of Zelda. [Digital game], 1986.

[161] M. Nitsche. *Video Game Spaces: Image, Play, and Structure in 3D Worlds*. Game studies. MIT Press, Cambridge, MA, U.S.A., 2009. ISBN 9780262141017.

[162] Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, U.S.A., 2002.

[163] Carl Magnus Olsson, Staffan Björk, and Steve Dahlskog. The conceptual relationship model: Understanding patterns and mechanics in game design. In *DiGRA 2014 Conference*, 2014.

[164] C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, K. Waugh, M. Carbonaro, and J. Siegel. A Pattern Catalog For Computer Role Playing Games. In *Proceedings of GAMEON North America*, pages 33–38, 2005.

[165] Ben Paechter, R.C. Rankin, Andrew Cumming, and Terence C. Fogarty. Timetabling the classes of an entire university with an evolutionary algorithm. In AgostonE. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature — PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 865–874. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-65078-2. doi: 10.1007/BFb0056928.

[166] Alex Pantaleev. In search of patterns: Disrupting rpg classes through procedural content generation. In *Proceedings of the 2012 Workshop on Procedural Content Generation in Games*, pages 57–61, May 2012.

[167] Alison Pease, Daniel Winterstein, and Simon Colton. Evaluating Machine Creativity. In *Workshop on Creative Systems, 4th Internation Conference on Case Based Reasoning*, pages 129–137, 2001.

[168] C. Pedersen, J. Togelius, and G.N. Yannakakis. Modeling player experience in super mario bros. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 132–139, Sept 2009. doi: 10.1109/CIG.2009.5286482.

[169] C. Pedersen, J. Togelius, and G.N. Yannakakis. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):54–67, March 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2043950.

[170] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, December 2007. ISSN 0742-1222. doi: 10.2753/MIS0742-1222240302.

[171] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3): 287–296, July 1985. ISSN 0097-8930. doi: 10.1145/325165.325247.

[172] Markus Persson. Infinite Mario Bros. [Digital game], 2008.

[173] Rhianna Pratchett. GAMERS IN THE UK - digital play, digital lifestyles., December 2005.

[174] Mike Preuss, Antonios Liapis, and Julian Togelius. Searching for good and diverse game levels. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2014.

[175] Re-Logic. Terraria. [Digital game], 2011.

[176] Chris Reade. *Elements of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0-201-12915-9.

[177] Ingo Rechenberg. *Evolutionsstrategie, Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973.

[178] Rockstar North. Grand Theft Auto V. [Digital game], 2013.

[179] David F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, Inc., New York, NY, USA, 1985. ISBN 0-07-053534-5.

[180] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 9780136042594.

[181] Katie Salen and Eric Zimmerman. *Rules of Play*. MIT Press, Cambridge, MA, U.S.A., 2004.

[182] Adam Saltsman. Canabalt (PC game). [Digital game], 2009. URL `http://www.adamatomic.com/canabalt/`.

[183] Schwefel, Hans-Paul. *Evolution and Optimum Seeking*. Wiley, 1995.

[184] Scorpia. Scorpia's Role-Playing Game Survey. *Computer Gaming World*, 87:16–27, 107–109, 1991.

[185] N. Shaker, G. N. Yannakakis, and J. Togelius. Feature analysis for modeling game content quality. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 126–133, Aug 2011. doi: 10.1109/CIG.2011.6031998.

[186] N. Shaker, M. Nicolau, G.N. Yannakakis, J. Togelius, and M. O'Neill. Evolving Personalized Content for Super Mario Bros Using Grammatical Evolution. In *IEEE Conference on Computational Intelligence and Games*, pages 304–311. IEEE, 2012. ISBN 978-1-4673-1193-9. doi: 10.1109/CIG.2012.6374170.

[187] N. Shaker, G.N. Yannakakis, and J. Togelius. Crowdsourcing the aesthetics of platform games. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(3):276–290, 2013. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2231413.

[188] Noor Shaker. *Towards Player-Driven Procedural Content Generation*. PhD thesis, ITU Copenhagen, 2013.

[189] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards Automatic Personalized Content Generation for Platform Games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. The AAAI Press, 2010.

[190] Noor Shaker, Julian Togelius, Georgios N. Yannakakis, Ben George Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter A. Mawhorter, Glen Takahashi, Gillian Smith, and Robin Baumgarten. The 2010 mario ai championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4):332–347, 2011.

[191] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'Neill. Evolving levels for Super Mario Bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 304–311. IEEE, 2012.

[192] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. Constructive generation methods for dungeons and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[193] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[194] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[195] Claude E. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30(1):50–64, 1951.

[196] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, U.S.A., 3rd ed. edition, 1996.

[197] Ruben Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 2:1–2:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814258.

[198] Adam M. Smith and Michael Mateas. Computational caricatures: Probing the game design process with AI. In *Artificial Intelligence in the Game Design Process, Papers from the 2011 AIIDE Workshop, Stanford, California, USA, October 11, 2011*, volume WS-11-19 of *AAAI Workshops*. AAAI, 2011.

[199] Adam M. Smith and Michael Mateas. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[200] Gillian Smith. *Expressive Design Tools: Procedural Content Generation for Game Design*. PhD thesis, UC Santa Cruz, Santa Cruz, CA, June 2012.

[201] Gillian Smith. Understanding Procedural Content Generation: A Design-centric Analysis of the Role of PCG in Games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 917–926, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557341.

[202] Gillian Smith and Jim Whitehead. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 4:1–4:7, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814260.

[203] Gillian Smith, Mee Cha, and Jim Whitehead. A Framework for Analysis of 2D Platformer Levels. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 75–80, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-173-6. doi: 10.1145/1401843.1401858.

[204] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 175–182, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-437-9. doi: 10.1145/1536513.1536548.

[205] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-937-4. doi: 10.1145/1822348.1822376.

[206] Gillian Smith, Ryan Anderson, Brian Kopleck, Zach Lindblad, Lauren Scott, Adam Wardell, Jim Whitehead, and Michael Mateas. Situating Quests: Design Patterns for Quest and Level Design in Role-Playing Games. In *Proceedings of the 4th international conference on Interactive Digital Storytelling*, ICIDS'11, pages 326–329, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25288-4.

[207] Gillian Smith, Ryan Anderson, Brian Kopleck, Zach Lindblad, Lauren Scott, Adam Wardell, Jim Whitehead, and Michael Mateas. Situating Quests: Design Patterns for Quest and Level Design in Role-Playing Games. In Mei Si, David Thue, Elisabeth André, James C. Lester, Joshua Tanenbaum, and Veronica Zammitto, editors, *ICIDS*, volume

7069 of *LNCS*, pages 326–329, Berlin / Heidelberg, 2011. Springer. ISBN 978-3-642-25288-4. doi: 10.1007/978-3-642-25289-1_40.

[208] Gillian Smith, Jim Whitehead, and Michael 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, 2011.

[209] Gillian Smith, Jim Whitehead, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha. Launchpad: A Rhythm-Based Level Generator for 2-D Platformers. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(1):1–16, 2011.

[210] Sam Snodgrass and Santiago Ontañón. Generating maps using Markov chains. In *Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence and Game Aesthetics*, pages 25–28, 2013.

[211] Sam Snodgrass and Santiago Ontañón. Experiments in map generation using markov chains. In *Proceedings of the 9th International Conference on Foundations of Digital Games*, FDG '14, 2014.

[212] Sonic Team. Sonic the Hedgehog. [Digital game], 1991.

[213] Spectrum Strategy Consultants. From exuberant youth to sustainable maturity - Competitiveness analysis of the UK games software sector. Consultant report, DTI - Department of Trade and Industry, U.K., 2002.

[214] Francis Spufford. *Backroom Boys – The Secret Return of the British Boffin*. Faber and Faber Limited, Croydon, U.K., 2003.

[215] Square. Final fantasy. [Digital game], 1987.

[216] Statista. Global PC and console games revenue in 2014 and 2019 (in billion U.S. dollars). [WWW], December 2015. URL `http://www.statista.com/statistics/237187/global-video-games-revenue/`.

[217] Steve Russell et al. Spacewar! [Digital game], 1962.

[218] Subset Games. FTL: Faster Than Light. [Digital game], September 2012.

[219] The NetHack DevTeam. NetHack. [Digital game], 1987.

[220] Tommy Thompson. The Fine Line Between Rehash and Sequel: Design Patterns of the Super Mario Series. In *Proceedings of the Forth Workshop on Design Patterns in Games*, DPG '15. FDG, 2015.

[221] J. Togelius and J. Schmidhuber. An Experiment in Automatic Game Design. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 111–118, 2008. doi: 10.1109/CIG.2008.5035629.

[222] J. Togelius, R. De Nardi, and S.M. Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 252–259, 2007. doi: 10.1109/CIG.2007.368106.

[223] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 Mario AI Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2010.

[224] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N. Yannakakis. Multiobjective Exploration of the StarCraft map space. In Georgios N. Yannakakis and Julian Togelius, editors, *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 265–272. IEEE, 2010. ISBN 978-1-4244-6295-7.

[225] Julian Togelius, Mike Preuss, and Georgios N. Yannakakis. Towards multiobjective procedural map generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814259.

[226] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In

Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I, EvoApplicatons'10, pages 141–150, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12238-8, 978-3-642-12238-5. doi: 10.1007/978-3-642-12239-2_15.

[227] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is Procedural Content Generation?: Mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0872-4. doi: 10.1145/2000919.2000922.

[228] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011. ISSN 1943-068X. doi: 10.1109/TCIAIG.2011.2148116.

[229] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. Procedural content generation: Goals, challenges and actionable steps. In *Dagstuhl Seminar 12191: Artificial and Computational Intelligence in Games*. Dagstuhl, 2013.

[230] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N. Yannakakis. The Mario AI Championship 2009-2012. *AI Magazine*, 34 (3):89–92, 2013.

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

[232] Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lane. Rogue. [Digital game], 1980.

[233] Triumph Studios. Age of Wonders: Shadow Magic. [Digital game], 2003.

[234] A.M. Turing. Intelligent machinery. In D. C. Ince, editor, *Collected Works of A.M. Turing*. Elsevier Science, 1992.

[235] Turtle Rock Studios. Left 4 Dead. [Digital game], 2008.

[236] Ubisoft Montreal. Far Cry 2. [Digital game], November 2008.

[237] Valtchan Valtchanov and Joseph Alexander Brown. Evolving Dungeon Crawler Levels with Relative Placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 27–35, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1084-0. doi: 10.1145/2347583.2347587.

[238] Valve Corporation. Alien Swarm. [Digital game], 2010.

[239] 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, March 2014. ISSN 1943-068X. doi: 10.1109/TCIAIG.2013.2290371.

[240] Ron Weber. Toward a theory of artifacts: a paradigmatic base for information systems research. *Journal of Information Systems*, 1(2):3–19, 1987.

[241] William Higinbotham. Tennis for Two. [Analog game], October 1958.

[242] World Machine Software, LLC. World Machine. [Software], 2011.

[243] Don Worth. Beneath apple manor. [Digital game], 1978.

[244] Georgios N Yannakakis and Héctor P Martínez. Ratings are overrated! *Frontiers in ICT*, 2:13, 2015.

[245] Georgios N. Yannakakis, Antonios Liapis, and Constantine Alexopoulos. Mixed-initiative co-creativity. In *Proceedings of the 9th Conference on the Foundations of Digital Games*, 2014.

[246] José P. Zagal, Michael Mateas, Clara Fernández-vara, Brian Hochhalter, and Nolan Lichti. Towards an ontological language for game analysis. In *Proceedings of International DiGRA Conference*, pages 3–14, 2005.

ISBN 978-91-7104-684-0 (print)
ISBN 978-91-7104-685-7 (pdf)

MALMÖ UNIVERSITY
205 06 MALMÖ, SWEDEN
WWW.MAH.SE