Fausto José da Silva Valentim Mourato
Mestre em Engenharia Informática

Enhancing Automatic Level Generation for Platform Videogames

Dissertação para obtenção do Grau de
Doutor em Informática

Orientadores: Professor Doutor Manuel Próspero dos Santos,
Professor Associado, Universidade Nova de Lisboa

Professor Doutor Fernando Pedro Birra,
Professor Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Professor Doutor José Legatheaux Martins
Arguentes: Professor Doutor Rafael Bidarra
Professor Doutor Rui Prada
Vogais: Professor Doutor Pedro Faria Lopes
Professora Doutora Teresa Romão
Professor Doutor Fernando Birra

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2015

Enhancing automatic level generation for platform videogames

Copyright © Fausto José da Silva Valentim Mourato, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

To my wonderful daughter, Sofia!

Acknowledgement

I would like to thank all of those that by some means contributed to the development of this work.

To begin with, my gratitude goes to my supervisors. Once more, I had the honour of having professor Manuel Próspero dos Santos as supervisor, continuing the guidance and the transmission of knowledge that started ten years ago, always making me be aware of detail and aiming for perfection. I also had the pleasure of having the supervision of professor Fernando Birra, who embraced this work with all the energy and availability, always bringing interesting insights and relevant expertise. This synergetic co-supervision surely contributed to this work come to a good end and both helped me become a better researcher and academic.

My acknowledgements extend to the remaining of the thesis advisory committee, the professors Rui Prada, Teresa Romão, Nuno Correia and Joaquim Filipe. Their feedback and expert comments were relevant to direct this work at some of its important stages.

I also thank all my family for the unconditional support. My wife, Filipa, did everything possible to help and was always there in the most demanding moments. Moreover, my parents and sister offered the most diverse contributions along the way.

Furthermore, I thank all my friends, students and gamers around the world that played and/or divulged our prototypes, making our studies possible.

As a final point, it is also important to mention the different funding sources involved in this dissertation:

- The development of this work was partially subsided by *Fundação para a Ciência e Tecnologia* (Portuguese *Science and Technology Foundation*) within the scholarship reference SFRH/PROTEC/67497/2010. This grant was set initially under the PROTEC program, an initiative to promote PhD formation for the teaching staff of Polytechnic Institutes, in this case, *Instituto Politécnico de Setúbal (Polytechnic Institute of Setúbal).*
- The research has been enclosed by *Centro de Informática e Tecnologias da Informação* (*Centre for Informatics and Information Technologies*), which subsidised the work under the grant PEst-OE/EEI/UI0527/2011.
- This thesis benefited of tuition exemption in agreement to the cooperation protocol established between *Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa* (*Faculty of Sciences and Technology, New University of Lisbon*) and *Instituto Politécnico de Setúbal (Polytechnic Institute of Setúbal).*

Abstract

This dissertation addresses the challenge of improving automatic level generation processes for platform videogames. As Procedural Content Generation (PCG) techniques evolved from the creation of simple elements to the construction of complete levels and scenarios, the principles behind the generation algorithms became more ambitious and complex, representing features that beforehand were only possible with human design. PCG goes beyond the search for valid geometries that can be used as levels, where multiple challenges are represented in an adequate way. It is also a search for user-centred design content and the creativity sparks of humanly created content.

In order to improve the creativity capabilities of such generation algorithms, we conducted part of our research directed to the creation of new techniques using more ambitious design patterns. For this purpose, we have implemented two overall structure generation algorithms and created an additional adaptation algorithm. The later can transform simple branched paths into more compelling game challenges by adding items and other elements in specific places, such as gates and levers for their activation. Such approach is suitable to avoid excessive level linearity and to represent certain design patterns with additional content richness.

Moreover, content adaptation was transposed from general design domain to user-centred principles. In this particular case, we analysed success and failure patterns in action videogames and proposed a set of metrics to estimate difficulty, taking into account that each user has a different perception of that concept. This type of information serves the generation algorithms to make them more directed to the creation of personalised experiences.

Furthermore, the conducted research also aimed to the integration of different techniques into a common ground. For this purpose, we have developed a general framework to represent content of platform videogames, compatible with several titles within the genre. Our algorithms run over this framework, whereby they are generic and game independent. We defined a modular architecture for the generation process, using this framework to normalise the content that is shared by multiple modules. A level editor tool was also created, which allows human level design and the testing of automatic generation algorithms. An adapted version of the editor was implemented for the semi-automatic creation of levels, in which the designer may simply define the type of content that he/she desires, in the form of quests and missions, and the system creates a corresponding level structure. This materialises our idea of bridging human high-level design patterns with lower level automated generation algorithms.

Finally, we integrated the different contributions into a game prototype. This implementation allowed testing the different proposed approaches altogether, reinforcing the validity of the proposed architecture and framework. It also allowed performing a more complete gameplay data retrieval in order to strengthen and validate the proposed metrics regarding difficulty perceptions.

Keywords: Procedural Content Generation, Game Design Patterns, Platform Videogames, Human Factors.

Sumário

A presente dissertação visa a melhoria dos processos de geração automática de níveis para videojogos de plataformas. As técnicas de geração procedimental de conteúdo evoluíram da criação de simples elementos para a construção de cenários e níveis completos. Assim, os princípios associados aos algoritmos de geração tornaram-se mais ambiciosos e complexos, permitindo a inclusão de características que anteriormente só eram alcançáveis pela criação humana. Deste modo, a geração procedimental vai além da pesquisa automática por geometrias válidas que podem servir de base para um nível, no qual diversos desafios estão representados de forma coerente. Trata-se também de uma pesquisa por características de *design* direccionadas ao utilizador, bem como por detalhes de criatividade que tipicamente são associados à criação humana.

Com o intuito de melhorar as capacidades criativas dos algoritmos de geração, parte desta investigação foi direccionada para a criação de novas técnicas onde fossem aplicados padrões de *game design* mais ambiciosos. Neste sentido, foram implementados dois algoritmos para geração de estruturas globais e um algoritmo adicional para adaptação de conteúdo. Este último permite a transformação de simples caminhos ramificados em estruturas mais complexas de desafios, pela adição de elementos de jogo em locais estratégicos, tais como portões e alavancas para a sua activação. Esta abordagem permite evitar a criação de níveis demasiado lineares e potencia a representação de padrões adicionais com maior dinâmica de conteúdo.

Adicionalmente, a adaptação de conteúdo foi expandida do domínio geral de desenho para princípios orientados ao utilizador. Assim, foram analisados padrões de sucesso e insucesso em videojogos, a partir dos quais foi proposto um conjunto de métricas para estimativa de dificuldade, tendo em conta que se trata de uma percepção subjectiva, distinta entre os diversos jogadores. Este tipo de métrica permite aumentar a capacidade dos algoritmos de geração automática, tornando-os mais direccionados à criação de experiências de jogo personalizadas.

A investigação conduzida visou também a integração de diversas técnicas sobre uma plataforma comum. Com este objectivo, foi desenvolvida uma *framework* para representação do conteúdo de níveis de videojogos de plataformas. Os algoritmos de geração desenvolvidos assentam sobre esta *framework*, tendo uma abordagem genérica que é independente do videojogo considerado. Foi também proposta uma arquitectura modular para o processo de geração. Procedeu-se ainda à criação de uma ferramenta para edição de níveis. Posteriormente, implementou-se uma versão adaptada desse editor para permitir a edição semi-automática de níveis. Esta permite a um *designer* definir o conteúdo pretendido sob a forma de missões e objectivos, cabendo ao sistema gerar uma estrutura física correspondente. Tal abordagem vai ao encontro da ideia de interligar padrões de desenho de alto nível com algoritmos de geração de baixo nível.

Finalmente, um protótipo de jogo foi também implementado, integrando as diversas contribuições. Foi assim possível testar os vários algoritmos num jogo único, reforçar a validade da arquitectura e da *framework* propostas, bem como realizar uma recolha de dados adicional para reforçar e validar as métricas relativas a percepção de dificuldade que foram apresentadas.

Palavras-chave: Geração Procedimental de Conteúdo, Videojogos de Plataformas, Padrões de Desenho em Videojogos, Factores Humanos.

Contents

1. Introduction ... 1

1.1. Motivation ... 1

1.2. The Challenge ... 3

1.3. Research Goals ... 4

1.4. Methodology ... 5

1.5. Contributions ... 6

1.6. Publications ... 7

1.7. Outline ... 8

2. Procedural (Content) Generation ... 9

2.1. Introduction ... 9

2.2. Motivation ... 10

2.3. Taxonomies and Categorisations ... 12

2.3.1. Contrasting Perspectives ... 12

2.3.2. Requirements ... 13

2.3.3. Content Abstraction Levels ... 13

2.3.4. Groups of Algorithms ... 14

2.3.5. Algorithmic Approaches ... 15

2.4. PCG Examples in Platform Videogames ... 15

2.5. Challenges ... 16

2.6. Concluding Remarks ... 20

3. Platform Videogames ... 23

3.1. Introduction ... 23

3.2. Setting the Domain ... 23

3.2.1. Content Description ... 23

3.2.2. Subgenres ... 26

3.2.3. Categorisation ... 27

3.2.4. Study Set ... 28

3.3. Game Design ... 29

3.3.1. Platform Videogames and Storytelling Capabilities ... 29

3.3.2. Ludo-Narrative Analysis of *Platformers* ... 33

3.3.3. Level Design Patterns in Platform Videogames ... 35

3.3.4. Quest Design Patterns for Platform Videogames ... 39

3.3.5. Multiplayer Design Patterns 41
3.4. Concluding Remarks....42

4. Human Factors 45
4.1. Introduction 45
4.2. Playing Experience and Emotions 46
4.3. The Concept of Difficulty 48
4.3.1. Difficulty Measurements and Approaches in Other Videogame Genres....50
4.3.2. Dynamic Difficulty Adjustments....50
4.4. Difficulty Measurement in Platform Videogames 52
4.4.1. Overall Level Structure Difficulty 52
4.4.2. Independent Challenges....56
4.5. Concluding Remarks....64

5. Architecture for a PCG System 67
5.1. Introduction 67
5.2. Architecture....67
5.3. Framework for Platform Games 68
5.3.1. Existing Approaches 69
5.3.2. Basic Organisation 70
5.3.3. Spatial Groups 71
5.3.4. Categories....72
5.3.5. Movements Aids 72
5.4. Level Editor 73
5.5. Prototype Implementation 79
5.6. Concluding Remarks....80

6. Using Graphs for Gameplay Representation.... 81
6.1. Introduction 81
6.2. Using Graphs to Represent Level Structures 82
6.3. Mapping Levels to Graph Representations 83
6.3.1. Graph Extraction Based on Pre-defined Rules....84
6.3.2. Graph Extraction from Example Levels....85
6.4. Graph Analysis 87
6.4.1. Initial Graph Processing 87
6.4.2. Path Extraction and Vertex Classification 89
6.5. Concluding Remarks....91

7. Automatic Level Generation Algorithms 93

7.1. Introduction 93

7.2. Related Work 93

7.3. Mapping Design Heuristics into a Genetic Algorithm 97

7.3.1. Level Representation 98

7.3.2. Heuristics for Level Evaluation 98

7.3.3. Operators 104

7.3.4. Closing Remarks 105

7.4. Graph-Based Adaptation Algorithm 107

7.4.1. Requisites 107

7.4.2. Implemented Adaptations 108

7.4.3. Examples and Results 110

7.4.4. Closing Remarks 115

7.5. Chunk Overlap Extractor and Generator 115

7.5.1. Algorithm 116

7.5.2. Output Correction 116

7.5.3. Closing Remarks 117

7.6. Algorithms' Taxonomy and Integration 119

7.6.1. Approaches 119

7.6.2. Output Type 120

7.6.3. Bridging Generation Algorithms with Quest Patterns 121

7.7. Semi-Automatic Level Generator 122

7.7.1. Approach 122

7.7.2. Interface and Main Features 122

7.7.3. Closing Remarks 125

7.8. Concluding Remarks 126

8. Results and Evaluation 129

8.1. Introduction 129

8.2. Experiment Details 129

8.3. Data Analysis 132

8.3.1. Players' Strategies and Behaviours 132

8.3.2. Difficulty Prediction 138

8.3.3. Level Evaluation 142

8.4. Concluding Remarks 145

9. Conclusions and Future Work **147**
9.1. Synthesis 147
9.2. Achievements 148
9.3. Further Developments 149
9.4. Concluding Statement 149

A. *Platform Level Editor* - Implementation Details **151**
A.1. Introduction 151
A.2. Generator Plugin Interface (C#) 151
A.3. Game Plugin Interface (C#) 152
A.4. Game Ontology XML – Example for *Prince of Persia* 152
A.5. Game Content XML – Example Level for *Prince of Persia* 155

References **159**

Cited Videogames **169**

Figures

Figure 1.1 – Genre classification for complexity and social interaction (Granic et al., 2014)..............3

Figure 2.1 – Screenshot of the videogame *Rogue*...10

Figure 2.2 – Screenshot of the videogame *Canabalt*..17

Figure 2.3 – Screenshot of the videogame *Spelunky*..17

Figure 2.5 – Screenshot of the videogame *Infinite Mario Bros*. 17

Figure 2.4 – Screenshot of the videogame *Cloudberry Kingdom*..17

Figure 3.1 – Ludo-narrative classification of an example set of platform videogames.......................34

Figure 3.2 – Cell structures for non-linear platform levels (Compton & Mateas, 2006). 36

Figure 3.3 – Design elements for the game *Super Mario Bros.* (Sorenson & Pasquier, 2010b), described as block, pipe, hole, staircase, platform and enemy. 38

Figure 4.1 – Example of common challenge representations in platform videogames. 53

Figure 4.2 – Example platform videogame situations. State chart for situation *a*. 53

Figure 4.3 – Example platform videogame situations. State chart for situation *b*. 54

Figure 4.4 – Example platform videogame situations. State chart for situation *c.1*.............................55

Figure 4.5 – Example platform videogame situations. State chart for situation *c.2*.............................55

Figure 4.6 – Example platform videogame situations. State chart for situation c, generalised. 56

Figure 4.7 – Principle of margin of error based on gap footprint (Sorenson & Pasquier, 2010a). 57

Figure 4.8 – Graphical representation of the concept of error margin. 58

Figure 4.9 – Jump classification regarding the placement of the destination point, P_1, in relation to the origin point, P_0. 58

Figure 4.10 – Maximum and minimum correction values to the trajectory to make a problematic trajectory possible...59

Figure 4.11 – Experimental measurement of percentage of failure in relation to the identified error margin of a challenge. 59

Figure 4.12 – Probabilities of failure in relation to the error margin considering different exponential values (K_d)...60

Figure 4.13 – Distribution on the jump origin point in a gap challenge. 62

Figure 4.14 – Boundaries to obtain the probability of the successful jumps..62

Figure 4.15 – Distribution over time for trials in a time-based challenge...63

Figure 4.16 – Distance and probabilities as a function of time for a moving platform challenge. 63

Figure 5.1 – Overview of the system architecture, composed by the following modules: Level Generation, Statistics and Profiling, Game Engine and System Manager and User Interface. 68

Figure 5.2 – Model of level framework proposed by Smith *et al.* (Smith et al., 2008).........................69

Figure 5.3 – Screenshot of the videogame *Super Mario Bros.* 71

Figure 5.4 – Example hierarchy for a simplified version of the videogame *Super Mario Bros.* 71

Figure 5.5 – Example of a resizable group and different possible concretisations. 71

Figure 5.6 – Example of a rule composed by the pattern (on the left) and the corresponding graph (on the right)....................73

Figure 5.7 – Screenshot of the implemented level editor using the proposed level representation framework, while editing a level for the videogame *Prince of Persia*. 74

Figure 5.8 – Setting the images that compose the bitmap representations within the *Platform Level Editor*. 75

Figure 5.9 – Setting the block hierarchy for the game description within the *Platform Level Editor*. ..75

Figure 5.10 – Setting the spatial groups for the game description within the *Platform Level Editor*. ...75

Figure 5.11 – Setting the categories for the game description within the *Platform Level Editor*. 76

Figure 5.12 – Setting the movement aids for the game description within the *Platform Level Editor*..76

Figure 5.13 – Configuration window for the definition of the automatic level generation pipeline. 78

Figure 5.14 – Example of automatic level generation with a sequential set of algorithms. At the top, the result of the first algorithm generates the base level structure. At the bottom, algorithm produces a decorated version of the previous level. 78

Figure 6.1 – A sample of the first level of the game *Prince of Persia* (on the left) and the corresponding graph (on the right). 84

Figure 6.2 – Example of two rules for graph construction in the game *XRick*, consisting of a pattern and the corresponding graph entry. 85

Figure 6.3 – Gameplay mapping of avatar positions. 86

Figure 6.4 – Obtained graph based on gameplay data....................86

Figure 6.5 – Example of five frequent situations for horizontal transitions....................87

Figure 6.6 – Vertex compression illustration for unidirectional passages. On the left, the original graph is presented and, on the right, the resulting graph is shown. 88

Figure 6.7 – Vertex compression illustration for bidirectional passages. On the left, the original graph is presented and, on the right, the resulting graph is shown. 88

Figure 6.8 – Example of a compressed graph. The compressed nodes are represented with dots. ..88

Figure 6.9 – An example of a potential additional graph compression in the game *Prince of Persia*. ..89

Figure 6.10 – An example of a potential graph compression case in the videogame *Infinite Tux*. 89

Figure 6.11 – Graph generated from the example level. 90

Figure 6.12 – Calculated tree with all possible paths from the beginning to the end of the level......91

Figure 7.1 – Rhythm-based level generation architecture (Smith & Whitehead, 2011). 94

Figure 7.2 – Depiction of the level exploration performed by the agents. 99

Figure 7.3 – Examples depicting the level exploration performed by the agents. 100

Figure 7.4 – Impact of the placement of the start and the end blocks within the levels regarding the existing paths and their length. 101

Figure 7.5 – A level obtained during the computation of the genetic algorithm, where cells marked as bad are identified with a cross. 102

Figure 7.6 – Examples of generated levels using different parameters regarding aesthetic balance. 103

Figure 7.7 – Example of the implemented crossover mechanism. 106

Figure 7.8 – Evolution of the levels' average fitness value. 107

Figure 7.9 – Examples of rules for level adaptation. The two rules on the left apply to the game *Prince of Persia* and the two rules on the right apply to *Infinite Mario Bros*. The two top rules intend to increase the level difficulty and the two bottom rules intend to decrease the level difficulty. 108

Figure 7.10 – Example of a gap in *Infinite Mario Bros.* and the possible level adaptation to reduce the gap size. 109

Figure 7.11 – Example of a detour adaptation rule in *Prince of Persia*. The rule on the left is applicable in the route to the dead-end and the rule on the right is applicable in the main path. 109

Figure 7.12 – Example of a situation that forces two players to follow distinct paths (the arrows denote the associations between trigger buttons and their respective gates, both marked with ellipses). 109

Figure 7.13 – Example of a tuned level for the game *Prince of Persia* (changes are marked with ellipses and arrows denote the associations between trigger buttons and the respective gates). 111

Figure 7.14 – Graph generated from the example level. 112

Figure 7.15 – Calculated tree with all possible paths from the beginning to the end of the level. The value *SDif* represents the estimated difficulty and *Len* refers to the segment length in cells. 112

Figure 7.16 – Example of a set of computed modifications. 113

Figure 7.17 – Example of a tuned level for the game *Prince of Persia* (changes are marked with ellipses and arrows denote the associations between trigger buttons and the respective gates). 113

Figure 7.18 – Example of a tuned level for our prototype *Tux Likes You* (changes are marked with ellipses). 114

Figure 7.19 – Example of a tuned level for the game *XRick* (changes are marked with ellipses). 114

Figure 7.20 – Example of a level with two independent segments, adjusted individually for two players of different skills. 115

Figure 7.21 – Possible example set of levels for *Infinite Tux* in the context of the chunk overlap generation algorithm (top) and identified transition points (bottom). 116

Figure 7.22 – Example of an automatically generated level using the chunk overlap algorithm. Transitions between levels are highlighted. 117

Figure 7.23 – Example of a level situation with an implicit gap. 118

Figure 7.24 – Example of two arbitrary level parts with common columns where shifting is allowed. Equal columns are marked with rectangles. 118

Figure 7.25 – Example of a wrongly created level using basic overlapping, as the level is not traversable. The transitions between chunks are marked with strong lines. 118

Figure 7.26 – Interface of the implemented semi-automatic level generator based on quests. 123

Figure 7.27 – Configurations for different patterns, namely *secret zone* of *collectible powers*, *multiple checkpoint* with *switch/door* and *hostile zone* to *defeat enemies*. 124

Figure 7.28 – Alternative geometries generated for a *checkpoint* pattern of the type *milestone*, considering the game *Prince of Persia*. 124

Figure 7.29 – Alternative geometries generated for a *multiple checkpoint* pattern of the type *switch/door*, considering the game *Prince of Persia*. 125

Figure 7.30 – Alternative configuration of a certain level part, for the game *Infinite Tux*, configured for different difficulty settings. 126

Figure 8.1 – Screenshot of our prototype game, *The Platformer*. 130

Figure 8.2 – Screenshot of our prototype, *The Platformer*. The character is running in the scenario. 130

Figure 8.3 – Screenshot of our prototype, *The Platformer*. The character is jumping across a gap... 130

Figure 8.4 – Screenshot of our prototype, *The Platformer*. The character is climbing to a platform above. 130

Figure 8.5 – Screenshot of our prototype, *The Platformer*. The character is facing a closed gate. 130

Figure 8.6 – Screenshot of our prototype, *The Platformer*. The character is stepping over a button that opens a gate. 131

Figure 8.7 – Screenshot of our prototype, *The Platformer*. The character is entering the exit door, finishing the level. 131

Figure 8.8 – Screenshot of our prototype, *The Platformer*. The player is attempting a jump over a gap. 133

Figure 8.9 – Screenshot of our prototype, *The Platformer*. The player is trying to move to the right avoiding the blade. 133

Figure 8.10 – Screenshot of our prototype, *The Platformer*. The player is jumping to a moving platform. 133

Figure 8.11 – Screenshot of our prototype, *The Platformer*. The player is jumping between platforms while avoiding a blade trap. 133

Figure 8.12 – Frequencies for the distance in relation to the platform edge in the jump origin, when jumping between static platforms. 134

Figure 8.13 – Mean of the distance to the gap's edge in relation to the player's ranking for a spatial challenge. 134

Figure 8.14 – Standard deviation of the distance to the gap's edge in relation to the player's ranking for a spatial challenge. 135

Figure 8.15 – Frequencies for the chosen instant to cross a timed-based challenge within one period. 135

Figure 8.16 – Means of the chosen instant to cross the time-based challenge, in relation to the player's ranking. 136

Figure 8.17 – Standard deviations of the chosen instant to cross the time-based challenge, in relation to the player's ranking. 136

Figure 8.18 – Frequencies for the distance in relation to the platform edge in the jump origin, when jumping to a moving platform. 137

Figure 8.19 – Frequencies of moving platform positions for jumps towards those platforms. 138

Figure 8.20 – Comparison of the percentages of failure using a gap challenge individually and in combination with a time-based challenge. 139

Figure 8.21 – Comparison of the percentages of failure using a time-based challenge individually and in combination with a spatial challenge. 139

Figure 8.22 – Probabilities of success (estimated and measured) for time-based challenges........... 140

Figure 8.23 – Measured probabilities of success for gaps with different values of x and y. 142

Figure 8.24 – Estimated probabilities of success for gaps with different values of x and y (at the left, estimator based on error margins and, at the right, estimator based on the distribution model). .. 142

Tables

Table 3.1 – Sub-genre classification for platform videogames according to the emphasis on movement, confrontations and interactions. 28

Table 4.1 – Estimated difficulty for different levels of the game *Super Mario Bros.,* considering alternative numbers of retries. 60

Table 4.2 – Measured and predicted percentages of success in a test level of the game *Little Big Planet.* 61

Table 8.1 – Probabilities of success (estimated and measured) for time-based challenges. 140

Table 8.2 – Probabilities of failure (estimated and measured) for a gap challenge. 141

Table 8.3 – Level evaluation for different authoring approaches and profiles 143

Table 8.4 – Users' speculations regarding the level creation method (humanly created or procedurally generated). 144

Table 8.5 – Level creation times using the *Platform Level Editor* and the *Quest Editor* (mean – μ, and standard deviation – σ). 145

Acronyms

AI	Artificial Intelligence
CG	Computer Graphics
DDA	Dynamic Difficulty Adjustments
FPS	First-Person Shooter
GA	Genetic Algorithm
GOP	Game Ontology Project
HCI	Human-Computer Interaction
NPC	Non-Player/Non-Playable Character
ORE	Occupancy Regulated Extension
PCG	Procedural Content Generation
RPG	Role Playing Game
RRT	Rapidly-exploring Random Tree
UGC	User-Generated Content
XML	Extensive Mark-up Language

1. Introduction

"If every act of intelligence is an equilibrium between assimilation and accommodation, while imitation is a continuation of accommodation for its own sake, it may be said conversely that play is essentially assimilation, or the primacy of assimilation over accommodation."

(Piaget, 1962)

1.1. Motivation

Videogames are increasingly present in the quotidian life. Daily, around the world, millions of people use computational devices for entertainment purposes and, specifically, to play games. The Entertainment Software Association[1] (ESA) claims that forty two percent of Americans play videogames more than three hours per week and that in 2014 consumers spent over twenty two billion dollars on computer games, hardware and accessories. These numbers are just a few indicators in a considerable wider range of statistics that prove a notion that is gradually established in everybody's minds: videogames are part of our lives. Possibly, the emphasis on the entertainment side of technology is just a direct consequence of the wide spread of that same technology. The search for entertainment is a natural reflex of intelligent beings, who play also as a learning mechanism, especially in the younger years. Thus, the science behind videogames and its study is not only about the technical features for the creation of this specific type of software products. It is also a matter of promoting new experiences, exploring the creative side of our minds and the expansion of learning opportunities.

Therefore, videogames gained relevance in academic and research context, especially in the last decade. One internal evidence is the work promoted by the Portuguese Society for Videogame Sciences (*Sociedade Portuguesa de Ciências dos Videojogos*)[2]. Their objective is to promote the topic within a scientific background in cooperation with multiple universities, supporting various initiatives in this context, such as conferences and workshops, among others. Abroad, it is also possible to encounter similar examples, such as the Society for the Advancement of the Science of Digital Games[3], established in the United States of America. Some research groups are reputable in this topic. Also in the United States of America, the main references on the subject are the Centre for Games and Playable Media[4], at the University of California, Santa Cruz and the Massachusetts Institute of Technology Game Lab[5]. Another example, in this case in Europe, is the Centre for Computer Games Research at the IT University of Copenhagen[6]. Lastly, one interesting community to refer, with a more informal background and with more emphasis on the industrial side of game development, is entitled *GameDev.net*[7]. They provide several articles, tutorials and other useful resources regarding the subject.

[1] http://www.theesa.com
[2] http://www.spcvideojogos.org
[3] http://www.sasdg.org
[4] http://games.soe.ucsc.edu
[5] http://gamelab.mit.edu
[6] http://game.itu.dk
[7] http://www.gamedev.net

Procedural Content Generation (PCG) refers to the creation of resources for a videogame or other similar interactive application by implementing one or more algorithms for the creation process, in opposition to manually creating that same content using editing tools. This approach allows independent developers and small companies to overcome the issue of lack of resources to design game environments from scratch. The increase of mobile and casual gaming expanded the videogame market, which brought additional opportunities to that type of companies. Besides, automatic generation processes promote content adaptation to the user profile, allowing the creation of levels to fit certain preferences, the user skills or other features.

PCG is an active research topic in videogames science, although it is likewise associated with several other computer related groups of interest, such as Human-Computer Interaction (HCI) and Artificial Intelligence (AI). A heterogeneous informal on-line community (*PCGWiki*[1]) was created with intensive discussions on this topic, reinforcing the frequent exchange of ideas on the matter.

The usage of random factors in PCG algorithms allows generating multiple alternatives for a particular type of content. Even though this is not restrictive, a common usage consists in the generation of graphical content. Regarding interactive applications, such as videogames, it is commonly applied to generate object models. For instance, modelling a forest is a time-consuming task than can be rapidly accomplished using procedural generation as an alternative, since there are several studies and techniques to model trees procedurally (Habel, Kusternig, & Wimmer, 2009; Weber & Penn, 1995). One subject inside this topic that has been debated recently concerns to the automatic and semi-automatic generation of videogame levels, in extension to the creation of specific parts of the scenarios that represent those levels.

This dissertation focuses on the automatic level generation for platform videogames. In this specific genre, the player controls a character, having as the main objective to reach a certain place in the scenario, by moving that character over entities that are usually referred as platforms, hence its designation. Therefore, the major interaction is to control the movement of a particular character, the player's avatar, which commonly has the ability of jumping. It is also frequent to have unreal overrepresented movement and jumping capabilities associated to the cartoonish aspect of these games.

Platform videogames, often referred as *platformers*, were particularly popular in the 1980's, where several titles were released such as *Super Mario Bros.* and *Sonic the Hedgehog.* Nonetheless, this genre is not lost in time. The characters used in these videogames are still popular nowadays and both these titles have recent remakes, namely *Sonic 4* by *Sega* and *New Super Mario Bros. U* by *Nintendo*, respectively, with improved graphics although keeping the original concept mostly intact. The simplicity of controls and interactions contrasts with contemporary approaches for videogames that have more multifaceted interactions that sometimes result in significant complexity for which certain users do not have physical and/or emotional availability. Nevertheless, simplicity does not mean lack of challenge. In fact, *platformers* follow one simple guideline: *easy to play but hard to master.* Moreover, platform videogames are one of the most balanced game genres regarding level complexity and social interaction. This can be observed in the conceptual map that overviews the most common game genres that is presented in Figure 1.1 (Granic, Lobel, & Engels, 2014).

In addition, game designers expanded the genre to take advantage of the Internet to allow the users to expand their creativity. For instance, the game *Little Big Planet*, released in 2008 as an exclusive for the *Playstation 3* system, allows the players to create and share new levels with other players. By the

[1] http://pcg.wikidot.com

date of this document, the *Little Big Planet* online community presented over nine million levels created by the users, available to play without costs. A considerable part of those levels shows professional quality.

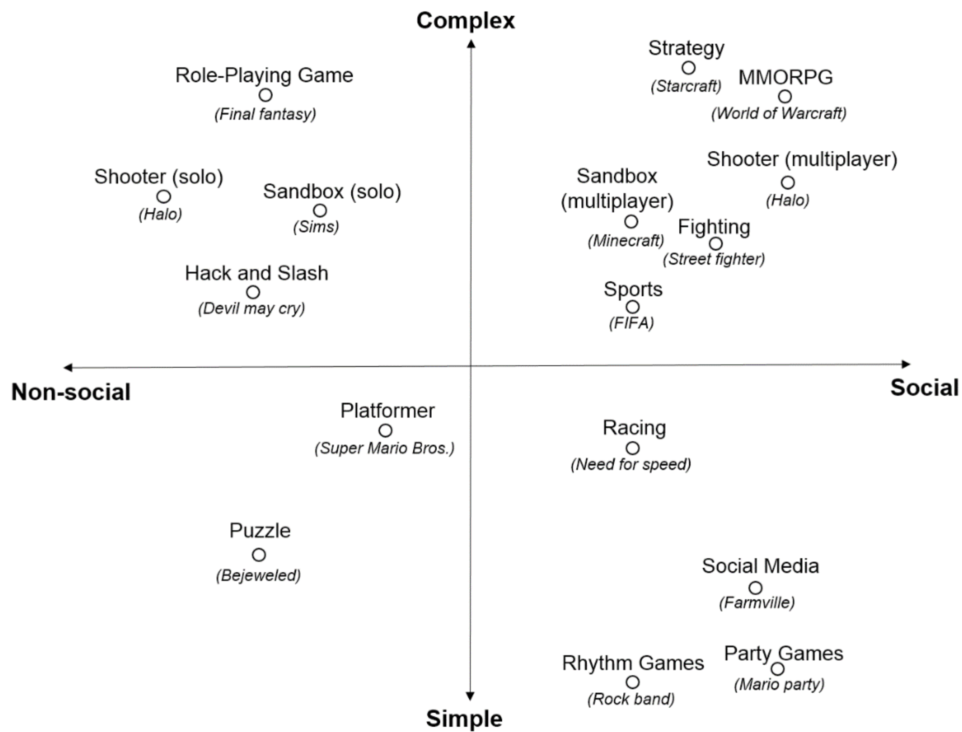


Figure 1.1 – Genre classification for complexity and social interaction (Granic et al., 2014).

A second version of *Little Big Planet* was released in 2011 and a third one in 2014, confirming that this type of games is still popular and has a large set of players. These sequels present minor updates regarding gameplay, which reinforce that the existence of a numerous amount of levels is not a threat to designed level sets. The market has space for new content created by game designers, even though the players have access to the referred numerous alternative community levels. Moreover, a considerable part of the users has interest in brands and franchises. For this reason, *Little Big Planet* provides level packs that can be bought, featuring characters and environments from other themes, such as the videogame *Metal Gear Solid* and the movie *Pirates of the Caribbean*. Therefore, in most cases, User-Generated Content (UGC) should not be seen as a replacement or a threat to the core content that is provided in the standard game versions.

In the same manner, PCG does not have to mandatorily imply the *"death of the level designer"* as it has been suggested (Andrew Doull, 2008). All these different approaches are interesting alternatives in the process of level design, which are not mandatorily exclusive. Some players want specific masterpieces from certain authors, which can be complemented with UGC to promote creativity and content variety. PCG can also encourage content diversity and increase the replay value of a videogame but some advantages are exclusive of this approach. The inclusion of procedural content allows its adaptation, for instance, to match a certain difficulty perception. To conclude, PCG can be used in semi-automatic procedures to speed up the human creation process, showing once again that the multiple design alternatives are complementary.

1.2. The Challenge

Platform videogames bring some interesting challenges to PCG. On the one hand, the geometry that these games use in the scenarios is considerably simple comparing to other videogames. Elements

are usually elementary polygons with a small set of edges and vertices, considering a two-dimensional representation. On the other hand, and even though those models are simple, it is not trivial to understand and define an algorithm to automatically generate levels that can be played. In addition, the result should not only be a playable level but also something that is visually coherent and aesthetic. More than a restrained search, this represents a case of virtual creativity. Furthermore, the output is very sensitive because small changes can transform a good output, which is a well-balanced level, in a useless output, which is an impossible level.

Although it may not seem intuitive at first sight, it is a harder task to generate a platform level than a composed landscape. Passing from procedurally generating a common environment to generating a game space brings the concept of difficulty and its inherent importance. An environment has to be generated with a set of rules that keep the physical validity of the output. In a game space, besides the physical restrictions, the output must represent challenges. In particular, small changes do not cause major problems in the generation of a regular environment but, in a game space, a small change can destroy a challenge, making it impossible. For instance, generating a terrain consists, typically, in displacing vertices and smoothing values. If a user makes random changes in that environment in a later stage, such as making a hill a little higher or digging a small valley, the output is still physically valid. However, in a game space, such as a platform videogame, changing the position of a platform, even slightly, can simply make a gap impossible to overcome and thus the output becomes invalid. Therefore, in game space generation, it is important to take into account the sensitivity of the produced levels. In particular, this has to do with the perceptions of difficulty that are associated with the represented challenges. Consequently, as we are dealing with perceptions, it is important to study such type of human factors. Usability is not a new term in Computer Science, and has been studied for more than twenty years, especially by the hands of Nielsen and Norman (Nielsen, 1993; Norman, 2002). In small terms, the concept represents the easiness of use of a certain object or system as well as its learnability. Still, not all usability principles apply to computer games, once the provided tasks are not intended to become trivial. Interacting with a simple graphical application is different from interacting with a videogame.

1.3. Research Goals

The main aim of this investigation is the general enhancement of procedural level generation for platform videogames. We established a set of goals in order to pursue that objective, namely:

- **Development of global level generation techniques.** The most popular approaches to procedurally create levels for platform games consist in the composition of a sequence of challenges, following a rhythmic structure (Smith, Treanor, Whitehead, & Mateas, 2009), or using pre-authored level parts, defined as chunks (Mawhorter & Mateas, 2010). Whereas such approaches are an interesting foundation to this subject, the results have limitations regarding the overall meaning of the levels. For this purpose, one important goal of this dissertation is the creation of context-aware generation algorithms with a more global analysis of the level content, such as the path branching, among others.
- **Creation of composed challenges.** Besides the previously identified lack of an overall analysis, the existing approaches focus on the creation of challenges composed by a single action, such as jumping over a gap or avoiding a trap. This work also aims to create challenges with the composition of multiple tasks, such as triggering a certain entity to open a passage or grabbing certain items that are required in later stages of the levels, among other possible types of composed challenges.

- **Expansion of the existing approaches to estimate difficulty.** While difficulty has been estimated in different manners in some game genres, this type of prediction in platform games was particularly incomplete. A goal of this work is the establishment of more concrete metrics to analyse the probability of failure in different types of challenges according to their features.
- **Inclusion of story related content in the generation process.** The evolution of automatic level generation should present an integration among different layers of abstraction within game content. This work also presents research regarding the possible alternatives for such type of integration, in particular in bridging story related concepts with level generators.

1.4. Methodology

The work hereby described was decomposed into three main research phases, which are explained next.

Phase 1

The first phase started under the premise that the existing techniques for the automatic level generation for platform videogames were limited, focusing mainly the creation of valid scenarios composed by a sequence of steps, without an overall view about the content.

Following the main premise, the first efforts regarding the generation algorithms consisted in exploring alternatives that could include global concepts. The main result regarding this aspect is a level generator based on game design heuristics, mapped into an evolutionary computational model. The first incursions in the concept of difficulty were also performed by analysing gameplay data and mapping it probabilistically. Besides, we have identified the need to merge the concepts of different videogames into a common ground, in order to promote comparisons among different titles. Thus, we defined a framework to represent content and levels for platform videogames.

In order to establish a base for further developments, an architecture for a complete procedural content generation system was defined, decomposed into distinct modules regarding the different aspects to consider, namely: generation algorithms, profiling and game engine. The referred framework for content description promoted the integration of the various modules.

Regarding the generation processes, this first phase allowed to identify that the existing algorithms for level creation could be improved if they could reflect gaming situations where the user has to perform composed tasks, such as, for instance, finding an object to interact in another part of the map.

Phase 2

In the second phase, the proposed architecture was put into practise with one initial prototype, entitled *Tux Likes You*, an adapted version of the game *Infinite Tux*, tailored to the referred architecture. Primarily, with that implementation, it was possible to test the feasibility of the idea. Moreover, we have done additional studies regarding generation algorithms. Specifically, some tests regarding the automatic level generation based on sample levels were performed.

With the purpose of achieving more composite challenges in the generated levels, we implemented an adaptation algorithm. It consists of an observation of previously created level structures and the adaptation of the existing paths to include additional gaming entities that force the user to gather objects and trigger events, among other modifications. Naturally, composed goals are only possible in videogames where those types of concepts are represented. The main results were obtained using

the game *Prince of Persia*. Furthermore, this technique lays strongly in the usage of graphs to describe the paths inside a level.

The conceptual idea of platform videogame was also explored. This genre was analysed regarding its main features and the main ideas behind the minds of game designers were dissected. We have analysed how these games include story and narrative elements, how they are transformed into challenges to the user via missions and quests and, finally, how they are connected to the level structure and the paths that are represented.

Phase 3

The third phase was intended as a confirmation of the obtained results. Specifically, it was designed to reinforce the relationship between higher-level concepts, such as the description of quests and missions in a level, and the generation algorithms that provide geometries where those quests and missions occur. A prototype of a semi-automatic generation tool was created with an author-centric approach, in which a designer can define the sequence of quests that he/she desires and the lower abstraction algorithms create an adequate level geometry for such structure. The studies regarding difficulty measurements have also been expanded to cover a wider range of challenging situations. The first phase already included some notions about this topic, but the usage of the implemented architecture encouraged this additional data gathering. The different prototypes implemented during our experiments reinforce the potential of the global architecture and the viability of such approach for small development teams. As a final point, an evaluation assessment was performed. We implemented a game prototype from scratch using the proposed framework and using the different approaches that were explored during this research. With this final experiment, it was possible to analyse the applicability of the proposed algorithms and approaches, and to expand the tests regarding difficulty estimators.

1.5. Contributions

The research hereby presented consists in the accumulated expertise that can be summarised in the following contributions to the scientific community:

- A better understanding about the experience of playing a videogame, in particular *platformers*, with strong emphasis on perceiving the notion of difficulty.
- A proposal for difficulty metrics in a platform videogame, established by a relationship to the probability of failure, based on geometry features and user parameterisation.
- The study of level design patterns for platform videogames and their association to the storytelling capabilities of the genre.
- A proposal for a generic level representation framework, adapting the principles proposed in related literature but with emphasis on geometry, which allows to represent and analyse levels within a common structure.
- The design of a general architecture for a PCG system with a proof of feasibility of the proposed approach with an effective system implementation.
- A proposal for new possible approaches to automatically generate platform levels, namely by mapping design heuristics into genetic algorithms and with the definition of a novel mechanism for implicit chunk identification.
- A tagging mechanism for graph representations of game levels, which provides semantic information about the level structure.

- A content adaptation algorithm that adapts the level structure to include composite challenges of non-linear gameplay, recurring to the graph tagging mechanism referred in the preceding point.
- A semi-automatic generator mapping different quest patterns with generation algorithms, which allows creating different level structures for a user-defined sequence of quests.

1.6. Publications

During this project, part of the obtained results was materialised into the following publications, grouped in their respective areas of interest:

Human factors

- *Measuring Difficulty in Platform Videogames* (Mourato & Próspero dos Santos, 2010), where we present our first insights about the concept of difficulty in platform videogames and how it can be predicted.
- *Difficulty in Action Based Challenges: Success Prediction, Players' Strategies and Profiling* (Mourato, Birra, & Próspero dos Santos, 2014), in which we extend our studies regarding difficulty with the inclusion of different types of challenges and applying a more extensive data analysis.

Generation algorithms

- *Automatic Level Generation for Platform Videogames Using Genetic Algorithms* (Mourato, Próspero dos Santos, & Birra, 2011), which contains an approach for automatic level generation using evolutionary computation to map basic design heuristics.
- *Enhancing Level Difficulty and Additional Content in Platform Videogames Through Graph Analysis* (Mourato, Birra, & Próspero dos Santos, 2012a), an article that presents an adaptation algorithm that is able to tune an existing draft geometry to include composed challenges and to perform difficulty adjustments.
- *Using Graph-based Analysis to Enhance Level Generation for Platform Videogames* (Mourato, Birra, & Próspero dos Santos, 2013b), an extension to the previous article, presenting with more detail the aspects regarding the graph analysis that are performed in order to understand the level structures and its possible modifications.

System architecture

- *Integrated System for Automatic Platform Game Level Creation with Difficulty and Content Adaptation* (Mourato, Birra, & Próspero dos Santos, 2012b), which contains an initial overview of the system architecture and the framework that has been used in this project.
- *Sistema Integrado de Geração Automática de Conteúdo para Videojogos de Plataformas* (Integrated System for Automatic Content Generation in Platform Videogames) (Mourato, Birra, & Próspero dos Santos, 2012c), a more detailed overview of the referred architecture for an automatic level generation system and the implemented framework for level representation.

Design patterns

- *The Challenge of Automatic Level Generation for Platform Videogames Based on Stories and Quests* (Mourato, Birra, & Próspero dos Santos, 2013a), a study about the storytelling capabilities of platform videogames and the initial insights of how to use the structures of a quest as a base to semi-automatic level generation.

1.7. Outline

Regarding its structure, this document is composed of eight more chapters following this introductory statement, structured as follows:

- **Chapter 2 – Procedural (Content) Generation –** reviews the main concepts behind the procedural and automatic creation of multimedia content and presents the main approaches in this topic. Its usage in computer games is explored, the possible taxonomies are analysed and the main challenges are identified.
- **Chapter 3 – Platform Videogames –** digs in the concept of platform videogame in order to extract the main notions of this type of games. The definition of this genre is clarified with the identification of the main features of these games and how they have been used to transmit story related content.
- **Chapter 4 – Human Factors –** explores different aspects related to human factors in gameplay, focusing on difficulty in *platformers* and taking into account its possible interest for automatic level generation. A set of metrics to predict difficulty are presented by mathematically relating the notion of difficulty to the geometric features of different challenges.
- **Chapter 5 – System Architecture –** presents a proposal to structure a procedural content generation system and the main components it may comprise. This architecture is supported by a representation scheme that is compatible with distinct games within the respective genre.
- **Chapter 6 – Using Graphs for Gameplay Representation –** describes our studies regarding the usage of graphs to analyse the structure of a level in a platform videogame, which is useful to some automatic generation algorithms.
- **Chapter 7 – Automatic Level Generation Algorithms –** explores the possible approaches to the automatic level generation for platform videogames, inspired by the studies that are presented in chapters 2, 3, 4 and 6, and using the system architecture that is presented in chapter 5.
- **Chapter 8 – Results and Evaluation –** describes an evaluation study using the previously presented principles and approaches. For this purpose, a novel game has been designed laying on the architecture presented in chapter 5. The level generation is accomplished using the techniques presented in chapter 7 and applying the difficulty metrics presented in chapter 4.
- **Chapter 9 – Conclusions and Future Work –** summarises the content in the most relevant conclusions and presents a set of ideas for further contributions to this area.

2. Procedural (Content) Generation

"We take for granted the ability of computer games to present players with engaging content, whereas demand for new and even player-customised content keeps increasing while manual content production is already expensive and unscalable."

(Hendrikx, Meijer, Van Der Velden, & Iosup, 2013)

2.1. Introduction

Procedural Generation is the common designation of the programmatic processes that generate any type of content. For instance, an image can be created procedurally by mapping the colour of each pixel to the result of a mathematical function that depends on the coordinates of those same pixels. Changing coefficients in that function produces distinct images. As an alternative, those same pixels can be obtained using random coefficients instead of the previous deterministic approach. In fact, this alternative is the main idea behind one of the pioneering techniques for the procedural image synthesiser that is nowadays designated as *Perlin noise* (Perlin, 1985). Despite the common usage in graphics, this type of generation can also be applied to other media. For instance, the result of any function over time, either with or without randomised factors, can be mapped into the amplitude and/or the frequency of a wave to generate sound. The challenge in the preceding cases of image and sound, and broadly in the topic of procedural generation, is the creation of valid results besides random noise without a particular meaning. With a more ambitious view, this type of generation can be an even more complex process, represented as a computer program that implements certain reasoning, heuristics or any other principles, in order to achieve valid results for the most varied types of content.

The expansion of the previous term to *Procedural Content Generation*, normally shortened to the acronym PCG, is typically used in the context of videogames and other interactive application when the creation process goes beyond the scope of mere aesthetics and has a certain influence over the global experience. Although there is no official definition for this term, it is broadly accepted that PCG is the "algorithmic creation of game content with limited or indirect user input" (Togelius, Kastbjerg, Schedl, & Yannakakis, 2011). In addition, regarding the output, PCG is stated as the procedural generation that "affects the gameplay significantly", as suggested in the *PCG Wiki*.

One of the first usages of PCG can be found about thirty years ago, in a videogame entitled *Rogue*. This type of generation was used to randomly create differently sized rooms, connected with corridors. By this time, the main used concept was not much more than pure randomness in order to achieve different results at each run. In particular, random sized rectangular rooms were generated and then connected with corridors, created randomly as well, resulting in scenarios as the one presented in Figure 2.1. This simple idea has inspired several other further implementations for the generation of game scenarios in which is common to apply the designation *Rogue-like* game.

In the remaining of this chapter, we will explore the main ideas of PCG and some of the challenges that we can find in this topic. We will start by presenting the main advantages and motivations behind procedural generation, in section 2.2. Then, in section 2.3, we will try to clarify some of the subjective

notions of what is and what is not PCG, and the classifications that one might have for different approaches. Furthermore, in section 2.4, we will show some examples of PCG in *platformers*. Moreover, the main challenges regarding research within this topic will be explored in section 2.5. To finish, the concluding remarks about this chapter will be presented in section 2.6.

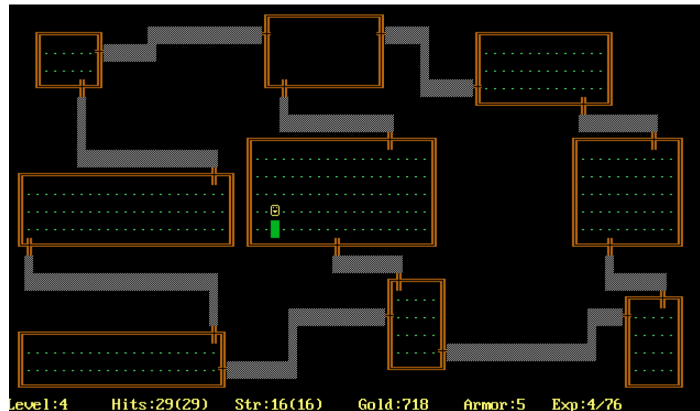


Figure 2.1 – Screenshot of the videogame *Rogue.*

2.2. Motivation

This broad initial view of the subject presents procedural generation as an alternative to represent and generate resources in any application, in opposition to modelling the content with a certain set of tools. Accordingly, one may wonder what benefits this may bring. The main advantages that are often associated to this approach are presented next in this section, centred in the development of videogames and the respective gains it may bring to both game developers and players.

To begin with, one aspect to take into account is the benefit that this approach brings to the replay value in videogames (often informally referred as *replayability*). The replay value refers to the motivation of a user to play a certain game all over again. Generating levels with automated principles gives the user the possibility of playing different levels every time he/she plays, with new challenges. Even though an infinite set of levels does not mean directly that the replay value is higher, a wider variety of alternatives is a natural way to avoid monotony in playing the same content multiple times. Still, it is important to reinforce that is up to the game creators to take advantage of what PCG can bring within this aspect. In addition, as referred in the introduction, this does not mean that manually created levels are useless and it is always better to create them procedurally. This should be seen as an alternative that presents some advantages that one can have in a videogame. For instance, automatically generated levels can be used in-between humanly created levels where the designers define the main story elements. Actually, story related aspects can also be used interestingly in the topic of PCG, as we will see with more detail along the remaining of this document, especially in section 3.3.

Furthermore, PCG allows avoiding predictability and the pre-establishments of tactics, an alternative view over the referred concept of replay value. In a non-PCG game, with fully authored content, as the user plays the game he/she will have to replay small parts after failing certain challenges. As every user plays those same parts, most of them will face difficulties in the same specific situations. Over time, players will tend to define strategies based on the specific situations that they are facing, instead of reasoning with abstract concepts. More than facing challenges, the players will start to follow recipes to overcome the design of the quests and will share those instructions among them. Frequently, this distorts the concept of the game and the spirit in which the missions were designed, and it even incites the players to take advantage of design flaws that meanwhile have been detected in

some challenges. The usage of procedural techniques is a solution to overcome this issue by performing small modifications in the generation process, providing similar yet different representations for one certain situation, thus creating distinct challenges that cannot be easily decomposed into a systematic solution. For instance, the videogame *Left 4 Dead* presents scenarios without predefined spawn points for the enemies. This simple detail increases the replay value, as the player cannot anticipate their position. With this feature, the surprise element is always present.

As referred in the introduction, the videogame *Little Big Planet* is a successful case of manually created levels, in particular with user creation. Still, some new ideas may arise from the fusion of the two approaches. Likewise, the community-based solution of this game follows another advantage of PCG, which is related to the needed human resources for level creation. In general, modelling graphical content is a time-consuming task. Therefore, one key idea of PCG is to reduce this time by allowing a small set of parameters, rules or principles to produce a wide set of results.

Another aspect commonly related to PCG is data compression. Depending on the type of application, a procedural level can be represented with a small set of parameters. A simple example is the usage of a simple random seed to generate the whole environment based on that value. This means that the game has a persistent scenario for a theoretical infinite world. This is the approach that was used in the videogame *Elite*, a spatial themed videogame in which the creation of the universe, composed by a set of planets with certain features, is performed algorithmically using an initial set of parameters. In a slightly different matter, one interesting case of storage compression associated to procedural generation is a game entitled *.kkrieger*, a demonstration developed essentially with procedural content. The use of procedural meshes and textures allowed representing a game with contemporary graphics with less than one hundred kilobytes.

Moreover, one specific unique advantage of PCG is personalisation. In the creation process, game designers have to create content in a generically acceptable base. Some mechanisms are implemented to overcome the differences among players, namely those that relate to the game difficulty. Specifically, some level parts can be defined with multiple variants to match different difficulty settings. Those can consist of small differences in the placement of opponents, traps or other entities, depending on the game features. Naturally, as this design process is based on manual procedures, it is impossible to tune the levels differently to every distinct player individually. However, if that content is generated procedurally, it is then possible to use the player's profile as input in the algorithms to make the levels more suitable to his/her skills, needs, desires or limitations. Those can be of different kind, from physical restrictions, such as colour blindness, to players' skills, as the typical contrast between the expert gamers and the occasional players. Considering that playing a game is also a learning process (Koster, 2004), this type of adaptation may be useful to promote ideal learning curves. Several aspects regarding personalisation on PCG will be explored later in this document, especially in chapter 4, where the concept of difficulty is analysed with more detail.

Lastly, we believe that PCG can promote the creation of new game mechanics. We have denoted that some of the advantages that can be found in automatically generating game spaces are not immediate and depend on their effective usage in gameplay mechanics by the game designers. For instance, a game where the user can simply play random levels in every gaming session is not an interesting challenge by itself. Since all levels are different, the players' scores are not likely to be compared, which does not promote competition among players. However, if every time that a level is generated for one player it is also sent to a certain group of random players, this creates competition groups for each level that is created and presents an alternative competitive mechanism that does not exist in non-PCG games. In conclusion, PCG can provide an ever-ending set of different levels that can be used in several alternative manners to promote distinct and innovative gaming mechanics.

2.3. Taxonomies and Categorisations

PCG can serve different purposes, focus on distinct aspects inside a videogame and, naturally, lay on different techniques depending on the objective. It is thus important to understand the main features that have to be considered when recalling PCG and the possible approaches that are used for the existing purposes.

2.3.1. Contrasting Perspectives

An initial taxonomy for the different approaches in PCG was proposed by Togelius *et al.* (Togelius, Yannakakis, Stanley, & Browne, 2010), which focuses on the five different perspectives that are presented next:

- **Offline vs. online.** This first distinction aims to divide the procedural generation processes in those that are used in the design phase in order to produce the game content and those that are used in the players' side, when they are playing the game. The given example of *Elite* is one case of offline PCG, in which the algorithms are used essentially for data compression and to speed up the production process. At runtime, the implemented routines simply decompress previously represented information. As offline techniques aim mainly for compression, we will direct our main attention to online approaches, in generation processes that are able to constantly produce new content and possibly taking the profile of the player as input for the generation process.
- **Necessary vs. optional content.** The authors presented the relevance of distinguishing PCG algorithms regarding their influence in gameplay, namely if the content that is being generated is necessary or optional. Their distinction presents these opposite sides stating that "necessary content is required by the players to progress in the game" in opposition to the content or game spaces that "the player can choose to avoid". For the creation of a complete level structure, mandatory and optional content must be coherent altogether. During our work, we have explored techniques for both settings. Later, in chapter 7, we will present our approaches for both situations and show how distinct algorithms for the two situations can be integrated in order to produce a whole level from scratch.
- **Random seeds vs. parameter vectors.** This distinction focuses on one important aspect that is commonly associated to PCG, which is randomness, namely evoking the control over the results of that same randomness. The minimum control over it is to define a random seed, which enables the generation of pseudo-random yet deterministic sequence of numbers, and use those numbers in the decision process along the algorithms. More control can be achieved by defining a vector of parameters that somehow changes the odds in every pseudo-random decision along the processes. These parameters can be anything that makes sense within the context of the generated content.
- **Stochastic vs. deterministic generation.** Still regarding randomness, the generation processes can be divided in stochastic and deterministic. As this definition exists outside PCG, it is essentially important to emphasise that the authors state that, in this context, one should not assume that a random seed means determinism, otherwise every PCG would be considered deterministic.
- **Constructive vs. generate-and-test.** The final distinction focuses on the algorithm strategy to produce the results. A constructive approach describes an algorithm that creates the content step-by-step, expanding the result at each iteration, ensuring validity in each expansion. In the case of the generate-and-test approach, the validity is only required in the end of the

process. Meanwhile, in transitional iterations, the results are analysed and some content parts are adapted to be used in further steps.

2.3.2. Requirements

From a complementary point of view, Shaker *et al.* (Shaker, Togelius, & Nelson, 2014) identified the main groups in which the possible requirements of a PCG system can be described, presented as follows:

- **Speed**. As performance is a commonly analysed feature in almost every system, the generation speed of a PCG algorithm is an important requirement to define. The generation process can be practically instantaneous or take a significant amount of time. The time requirements have to do directly with the final use of its results. Recalling the previous taxonomy, content can be generated either online or offline. Naturally, online algorithms have to perform faster while the requirements for offline algorithms are usually inferior.
- **Reliability**. This feature addresses the validity of the created output and how invalid cases are tackled. Typically, higher reliability requirements are expected for the elements that have more relevance to gameplay. However, even though this feature is presented as one requisite to establish for the algorithms, it is also relevant to analyse how such algorithms are used within a generation pipeline, as it may interfere directly with the effective reliability of such algorithm. For instance, a certain algorithm that does not ensure validity in the output can be used in a reliable generation process if used within a generate-and-test loop, massively generating and testing new content in an additional step, until certain output criteria is met. As long as this example case does not interfere with speed requisites, the full process must be considered reliable.
- **Controllability**. This aspect refers to allowing a human to specify some features of the generated content. There are several associated aspects, from aesthetic elements to mood related content. One interesting aspect to control in the generation of a level is the sense of difficulty in that same level. We will approach that topic specifically in chapter 4.
- **Expressivity**. Also referred as diversity, this feature has to do with the capability of producing varied results in its domain, ideally being able to cover all possible valid representations for the application in which it is being used.
- **Creativity**. This feature can be referred as believability and it consists in stating how similar to a human designer an algorithm is. In practice, this can be seen as a Turing test for the capability of level design.

2.3.3. Content Abstraction Levels

With another perspective, Hendrikx *et al.* (Hendrikx et al., 2013) presented a survey on the topic, focusing on the domain of videogames. The authors observed game content in different abstraction layers, as presented next:

- **Game bits.** These are elementary units of the game content that do not have a particular engagement role when interpreted by themselves. In this category, it is possible to include the automatic generation of textures (Ebert, Musgrave, Peachey, Perlin, & Worley, 2002), sounds (Nierhaus, 2009), nature elements (Re, Abad, Camahort, & Juan, 2009) and buildings (R. M. Smelik, Tutenel, Bidarra, & Benes, 2014).
- **Game space**. This group refers to the environments and scenarios where the game takes place. Game spaces are correct compositions of models using different types of game bits.

For further study in this specific topic, we point to the survey of Kelly and McCabe about city generations methods (Kelly & McCabe, 2006) and the terrain modelling overview presented by Smelik *et al.* (R. Smelik, Kraker, Groenewegen, Tutenel, & Bidarra, 2009).

- **Game systems**. This category refers to game spaces containing specific complex models to make them more realistic. It includes the generation of nature rules, urban environments, transports networks and all types of social interactions. In this last feature, an interesting original work is worth mention, entitled *Façade* (Mateas & Stern, 2003a, 2003b, 2004, 2005), an interactive drama videogame in which the user plays by interacting within a couple's discussion about their life, without any specific predefined goal.

- **Game scenarios**. This group expands the general dynamisms presented in the previous category, with emphasis on the specific set of events that describes the game and the corresponding story. It includes the main structure of a level and their respective puzzles (Colton, 2002) and stories (Ashmore & Nitsche, 2007).

- **Game design**. This higher abstraction level refers to the creation of the specific features that transform a simple scenario into a gaming environment, representing certain challenges. This includes generating the interconnection of settings, stories and themes (Brathwaite & Schreiber, 2008) in order to define the game background, as well as the system design that comprises the game rules.

- **Derived content**. This final group refers to the content that the game produces as a result of gameplay and that may be reproduced in any type of media, namely leader boards and mechanisms for news and broadcasts. While it is easy to understand the principles of a high-scores table, in a game where the rules have been created procedurally, the challenges are how to measure those scores, how they can be transformed into achievements, trophies or other rankings, and how the related important events should be announced and divulged.

2.3.4. Groups of Algorithms

Besides the presented decomposition of game content into multiple abstraction levels, Hendrikx *et al.* (Hendrikx et al., 2013) have also identified six main groups to separate the algorithms, which are described next:

- **Pseudo-random number generators.** This first group of algorithms is not likely to be used individually but is expected to be part of every PCG system. It refers to all algorithms that aim to generate sequences of numbers that behave in an apparent random manner.

- **Generative grammars.** This group encloses the approaches based on the concept of grammar, a set of rules that produces or identifies the possible words in a language (Chomsky, 1956). Regarding PCG, words have a more ample meaning, relating to every possible valid entity in a certain universe. Different approaches based on generative grammars are typically used in PCG, such as L-systems (Lindenmayer, 1968) and shape grammars (Özkar & Stiny, 2009; Stiny & Gips, 1971), among others.

- **Image filtering**. This category groups the algorithms that perform the operations of procedural generation based on a two-dimensional grid, interpreted as a rasterised image composed by pixels (Gonzalez & Woods, 2007).

- **Spatial algorithms**. This group refers to the generation processes that analyse space features by decomposing that same space. The most common approaches included in this group are

tiling and layering (Lagae, Kaplan, Fu, Ostromoukhov, & Deussen, 2008), grid subdivision (Fournier, Fussell, & Carpenter, 1982; Miller, 1986) and fractals (Mandelbrot, 1977).

- **Modelling and simulations of complex systems**. This group refers to the generation algorithms that are represented as systems that somehow resemble existing rules in nature. For instance, a terrain can be created based on erosion principles using tensor fields (Chiba, Muraoka, & Fujita, 1998).
- **Artificial intelligence**. This group directs the topic to the approaches inspired by this traditional research field, including genetic algorithms (Mitchell, 1998), artificial neural networks (Hassoun, 2003) and constraint satisfaction problems (Tsang, 1993). In our research, we have used genetic algorithms as a mechanism to generate levels, as we will see later in this document, in section 7.3.

2.3.5. Algorithmic Approaches

As a final point, Smith (Smith, 2014b) observed the state of the art of PCG and presented an analytical framework to understand the role of PCG in games from a design point of view. In this context, she proposed a distinction into five different approaches as follows:

- **Optimisation**. In this approach, the design process works as a search within the domain of possible solutions, in which the output has a set of heuristics for its evaluation.
- **Constraint satisfaction**. This type of methodology involves a strong set of rules that validate the result and a procedure to find solutions within that set of rules.
- **Grammars**. Their implementations consist in the definition of rules that are able to build the level gradually.
- **Constructive**. This type of generators works by expanding an initial level base, as grammars do, but the construction process is not defined in the formal mechanism of rules. Instead, it consists of assembling pieces from an existing predefined set of building chunks.
- **Content selection**. This approach can be seen as an adaptation of the constructive approach, but in which the building chunks are especially large (an effective level segment) and their composition is rather trivial. As it is arguable that this can be considered an approach by itself, it is also controversial that this type of generation can be considered PCG (Togelius et al., 2011).

Through this section, we have seen that there are multiple approaches and techniques in the main topic of PCG. The different surveys used as main references explored the differences of alternative approaches from distinct points of view. To conclude, an additional survey is worth mention in this topic for complementary analysis, in which Cooper *et al.* (Cooper, El Rhalibi, Merabti, Wetherall, & Rhalibi, 2010) described some of the pointed features along multiple abstraction levels, integrated within a system for the development of general world scenarios.

2.4. PCG Examples in Platform Videogames

Along this chapter, we have unveiled the common approaches in the topic of PCG and, meanwhile, we have observed some example applications in commercial videogames. We now direct the attention to its specific usage in platform videogames by presenting examples of commercial games using PCG within this genre.

- ***Canabalt*** (Figure 2.2) is a game inspired by the *platformer* genre, developed for the *Experimental Gameplay Project*[1] community contest themed *minimal*, available to play for free online[2]. While scenarios present the common notion of platform videogame, the gameplay in this game consists in a reduced control interface with only one possible action: jumping. This likely sub-genre of platform videogames has become popular in recent years with the increase of casual gameplay in mobile devices, where we can also point to similar example games such as *I Must Run* and *Run Like Hell*. In chapter 3, we will explore the *platformer* genre and discuss whether these games should be considered *platformers* or not, and its implications in the specific topic of PCG.

- ***Spelunky*** (Figure 2.3) is another free downloadable game[3] that is particularly interesting to consider in this work because, in some aspects, it consists of a platform game with procedural level generation based on simple principles and heuristics. Besides, the source code was released, which allows a deeper look at those principles. The approach was interesting enough to grow recent attention in academic research. For instance, Scales and Thompson (Scales & Thompson, 2014) implemented an AI toolset for this game, which might serve researchers to study possible alternatives regarding artificial players or level generation.

- ***Infinite Mario Bros.*** (Figure 2.4) is a remake of the classical videogame *Super Mario Bros.* and is a frequent framework to test generation algorithms (Togelius, Karakovskiy, & Baumgarten, 2010; Shaker et al., 2011) and intelligent agents to control the avatar (Karakovskiy & Togelius, 2012; Togelius, Karakovskiy, Koutník, & Schmidhuber, 2009). We will describe some of those studies during this document. Recently, an adapted version of the engine with different resources has also been released under the designation *Infinite Tux*.

- ***Cloudberry Kingdom*** (Figure 2.5) is a *platformer* where small levels are generated procedurally at runtime. As stated, PCG can be used as way to promote new gameplay approaches and this videogame is an example of such case. While the existence of continuous randomly generated small levels without a story does not represent an interesting game, the creators of *Cloudberry Kingdom* implemented alternative game modes such as, for instance, passing as much levels as possible within a certain time limit, as a way to make the game interesting.

These examples show the potential of procedural generation for platform games. As those attempts have been mainly done without scientific context by independent developers, some ideas are still *ad-hoc* principles and, in some cases, technical details are absent. However, they provide a good inspiration for scientific research.

2.5. Challenges

Before proceeding to the implementation of any type of PCG system, it is important to understand the main contemporary lines of work in this topic and where the main contributions are being addressed. Togelius *et al.* (Togelius, Justinussen, & Hartzen, 2012) identified the main desirable properties for the outcome of a PCG system, with the following order of increasing importance:

- **Consistency**, meaning that, in first place, a level must meet certain lexical and syntactical rules. In other words, the generated entities and their usage must make sense in the game's domain.

[1] http://experimentalgameplay.com/
[2] http://adamatomic.com/canabalt/
[3] http://www.spelunkyworld.com/

Figure 2.2 – Screenshot of the videogame *Canabalt.*

Figure 2.3 – Screenshot of the videogame *Spelunky.*

Figure 2.4 – Screenshot of the videogame *Infinite Mario Bros.*

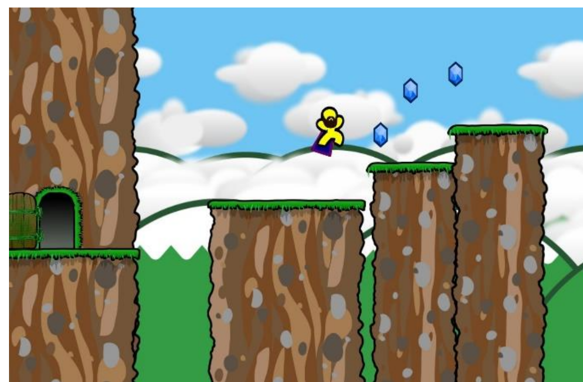


Figure 2.5 – Screenshot of the videogame *Cloud-berry Kingdom*

- **Playability**, which implies that the created scenarios must allow the game rules to take place. In a platform game, this means that the user has to be able to move the avatar and to perform the various additional actions that are available in the game.

- **Winnability**, a feature that extends the previous principles by guaranteeing that the winning conditions are represented and that they are accomplishable. For instance, in a platform videogame, a level can be considered playable if it comprises a path from the starting point to the ending position of that same level. The existence of a gap between platforms that it is physically impossible to jump through, placed in a mandatory part of scenario, makes this property unfulfilled.

- **Fairness**, referring to the adequacy of the implemented reward system. In a single player perspective, accomplishing certain tasks should provide rewards because of those successful actions. From a multiplayer point of view, this has to do with the similarity of conditions in which each player competes.

- **Challenge**, extending the concept of fairness to a more abstract notion in which the levels should not only represent cases of rewarded tasks but also those tasks should not be trivial or excessively complex. A level is created to be a test to certain skills in an adequate manner and the rewards should be proportional to the challenge that is represented.

- **Interest**, which is the final goal of any type of generated content. This is an even more abstract concept than the challenge itself. Levels should not only provide challenging tasks but should also provide those tasks in an interesting manner.

In fully automated processes without human supervision, the first three aspects are natural requirements, as they represent validity conditions regarding the output. However, it is in the last features where we can encounter the real interest of PCG and the effective applicability of the concepts. In order to make a game fair, it is important to understand how the various game entities interfere in the experience of gameplay. This feature is directly related to the following requisite of challenge, which emphasises the need of identifying the represented difficulty, a question that is addressed further in this document, in chapter 4. Finally, making a game interesting is more than making good challenges. Those challenges have to transmit concepts, ideas, metaphors, stories or other types of content richness. We will explore the concept of quest in section 3.3 and present an approach to use such principle as part of a semi-automatic generation process in section 7.7.

Regarding the possible lines of research for PCG in games, we can point to the following five main directions, as presented in the referred survey of Hendrikx *et al.* (Hendrikx et al., 2013):

- **Generation of content at the top of the content pyramid**. As bottom abstraction content has a lower radius of influence over the content of a game, the algorithms for that level tend to have fewer restrictions. For that reason, this type of algorithms has been researched with higher emphasis on the last years. Currently, as research in that type of algorithms progressed, the interest on automatic generating content at higher levels increased as well. Thus, it is important to understand how to automatically create those higher abstraction concepts and/or associate them with the existing concrete algorithms.
- **Development of more detailed generators.** Game content is increasingly becoming more detailed regarding different features of realism. For instance, buildings in games that were developed twenty-year ago were mere cubes with simple façades while nowadays they present architectural and semantic details. Whereas content detail increases, new automatic generation algorithms need to address those details as well in order to keep up to the human creation capabilities.
- **Generation of "missing" game bits**. It is always possible to think of additional small details of game content in which PCG has not been used yet. Likewise, one can think of new content to include in videogames. Recalling the last point about the increase of detail in videogames, it is natural to think that if detail increases in bottom level content, upper level content will also become more detailed, which sometimes will result in the inclusion of new bottom level content that has not been considered before.
- **Handle of detail-performance trade-off.** This aspect points the importance of the research about generating content at scale. While processing power increases, some of the generation algorithms that were previously considered exclusively to offline usage start to be thought as plausible to be used online, either by scaling them in, with multi-core support, or scaling them out, taking advantage of parallel computing.
- **Evaluation of generated content.** As long as games are still played by humans, there will always be human factors to analyse in gameplay. The traditional game creation process consisted in humans creating experiences for other humans, which by itself is still an open area of research, because those experiences are subjective and thus hard to quantify. Generating content brings all those questions about experience and adds challenges in the mimic of the design process.

Our studies regarding generation algorithms can be considered as being directed to the generation of content at upper abstraction levels of the content pyramid. As stated, the ultimate goal of PCG is to make content interesting, as games should include creativity and a motivating background. With that

goal in mind, part of this research was directed to automatically generate game spaces and game scenarios, as we will further see in chapter 7. Besides, this research presents advances in the evaluation of generated content. Accordingly, we will draw some attention to the concept of difficulty later in chapter 4. Moreover, in chapter 8, we will analyse the quality of user created content in comparison to procedurally generated content.

Finally, it is important to point to a more general overview to the future of PCG research. Togelius *et al.* (Togelius et al., 2013) identified the main long term directions, described as the following three main groups:

- **Multi-level, multi-content PCG** refers to the creation generation algorithms that are capable of addressing the multiple levels of detail within game content. In some manners, this research direction is related to the referred goal of generating content at the top of the content pyramid and the integration with lower abstraction concepts. This line of work is yet to produce effective results, as there is no such system yet, even though some examples point in that direction. Currently, one good example of a videogame in the route towards this principle is *Dwarf Fortress*, probably "the most complex videogame ever made" (Tyson, 2012). In this game, the generated details go from the map landscape to the individual fingers, teeth, and organs of the creatures that inhabit that landscape. Furthermore, all those features have influence over gameplay. However, despite the extensive level of detail and the complexity of the simulation system, the procedural generation part is quite simplistic, mostly not more than simple controlled randomness, presenting little variation. In academia, two main works are worth mention that also point to this research direction. The first, brought by Hartsook *et al.* (Hartsook, Zook, Das, & Riedl, 2011), presents the usage of PCG within some of the existing abstraction layers of game content. In particular, the implemented system works with a top-down approach, starting by generating a story, followed by the generation of the map structure and the contained elements. The second example is the work of Smelik *et al.* (R. Smelik, Tutenel, Kraker, & Bidarra, 2011), in which the authors present *SketchaWorld*, a system that generates virtual environments based on primal designer description about the content. The most interesting part regarding PCG is the usage of semantics that relate the existing entities in the map among themselves. With this, "a local change to the landscape affects all terrain features in that area, each to an extent that is defined by its particular semantics".
- **PCG-based game design** is still an abstract idea that consists in using PCG as a core element of gameplay in which "exploration of an infinite range of content is a central part of the gameplay" (Togelius et al., 2013). It is seen as such usage of PCG in which the game could not exist without it. Two main examples that barely show what this concept aims to be are the games *Galactic Arms Race* (Hastings, Guha, & Stanley, 2009) and *Petalz* (Risi, Lehman, D'Ambrosio, Hall, & Stanley, 2012). In the first, the "players pilot space ships and fight enemies to acquire unique particle system weapons that are evolved by the game", according to the identified preferences throughout gameplay. In the second, the players have seeds that generate plants, which can be shared with other players or combined with other seeds to generate new morphologies. Plants and seeds are then part of a social market in which value is implied to the result of PCG algorithms.
- **Generating complete games** refers to the creation of the game content and the game rules where that content is used, as well as the complete game interface and interaction mechanisms. Likewise, this is a long-term goal far from being achieved. The main progresses are

still directed only to the automatic generation of game rules, where we can refer some interesting examples. Togelius and Schmidhuber (Togelius & Schmidhuber, 2008) generated different games based on an overall mechanism consisting in the movement of entities within a two-dimensional environment. The authors used an implementation built on artificial neural networks that progress with evolutionary algorithms. A similar work also based on evolutionary computation has been presented by Cook and Colton (Cook & Colton, 2011), aiming to the creation of simple arcade games. Another similar approach is the work of Treanor *et al.* (Treanor, Blackford, Mateas, & Bogost, 2012). More than just generating content, this work consists in generating game variants for *Pac-Man-like* games. A game contains an agent and some entities in a discrete gridded game space, where some parameters are set for their behaviour. Evolutionary computation is used to make those parameters evolve in order to reach a set that represents one specific game variant. In the end, some prominent games were obtained. For instance, one of the evolved rule sets converged to a game where the user controls the agent with the goal of catching the blue mark, a simple materialization of a catch and run game. Despite these motivating examples, procedurally generating complex games, application interfaces and other features is yet to be proved possible.

As this dissertation focuses on a specific genre, the main efforts that will be presented in the remaining of this document should be seen as multi-level multi-content PCG. Some of our studies aimed to improve the bridging between different abstract layers of game content. That was done by relating the algorithms for game space generation with contextual information in which those spaces are intended to be used, as a background story or a set of missions and quests. We will detail that idea in section 3.3.

2.6. Concluding Remarks

We have observed PCG as an interesting research area in which every component of a videogame can be thought to be created automatically. While this is already possible for some of its features, there is still an uncountable set of future work to be done in other aspects. To close this chapter, we believe it is important to take a final look at some of the still arguable notions.

Even though there is some common ground in the previous definitions, there are still some aspects that are not clear if they should be considered PCG or not. Following the work of Togelius *et al.* (Togelius et al., 2011), one can point two main aspects of possible disagreement.

The first is described as offline player created content and refers to the usage of procedural based editors to generate that same content. The authors stated that such situation should not be considered PCG if the editing process provides immediate and controlled output. The presented example is *Spore: Creature Creator*, a character editor for the game *Spore* in which the user creates characters by stating their features, namely the number and the aspects of several body characteristics. Whereas the composition of those parts is procedurally implemented, the fact is that the user stated exactly the output he/she desired. Moreover, we add that the presented arguments are valid for any avatar editor as those that are popular in fantasy games. Still, the computational and algorithmic efforts are noteworthy and deserve their merits. We will point to these situations as procedural composition.

The second aspect is pointed as online player created content and refers to content that is generated as part of gameplay. As proposed by the authors, we will look at this situation using the mechanics of the game *Civilization*. While playing, the user places cities in the map, connects them with roads and evolves their content with certain buildings, meaning undoubtedly that, during gameplay, content is generated. The question is if this process of content generation can be considered PCG. To create even more entropy, this question can be extended to the content that is created by the AI opponents.

The authors point that this should not be considered PCG as "the human input to the content generator is part of a game, and the player directly intends to create content in the game". As it is understandable this point of view to make a distinction between PCG and strategy-games, we believe that this argument is not strong enough to exclude this type of content. A strategy game is a simulation with additional interaction rules, which does not compromise any of the concepts that are suggested to be PCG. In addition, we previously referred *SketchaWorld* as a PCG example. Despite the gameplay aspects regarding finance management, it is impossible to point differences to a game like *Sim City*. Naturally, if during gameplay the user can totally control the outcome of its actions regarding the generated content such as, for instance, the houses that are created in *The Sims*, this cannot be considered PCG.

Besides the previous arguable points of view, similar questions arise regarding the specific requirements of a possible PCG system. Specifically, one feature with further debate is randomness or, more specifically, determinism. A common question is if deterministic systems can be considered PCG. The authors state so and point the game *Elite* as an example where a single universe was generated with stochastic principles. We agree that such situations should be considered PCG. Indeed, for videogames where one intends to create infinite universes, procedural algorithms are required, even if they consist of completely deterministic implementations with a predefined set of parameters.

Finally, despite these dissimilarities, one irrefutable fact is that procedural principles can and should be used to make game content more dynamic and diverse in order to promote better gameplay. As long as the algorithms mimic or aid the work of the designers in some manner regarding the generation process, their research is important, either if classified or not under the designation of PCG.

3. Platform Videogames

"Platform games are fun to make and also pose some very interesting programming and design challenges. From a programming point of view, if you can program a platform game, you have reached a benchmark in your development as a game designer."

(Spuy, 2012)

3.1. Introduction

This work focuses on PCG for the automatic level generation in the specific genre of platform videogames (frequently referred as *platformers*, as already referred in the introduction). Even though this is a common designation used worldwide, the excessive informal use of the term and the lack of a formal definition make the boundaries of the concept unclear. We will start by focusing this issue in section 3.2, analysing the main features of this type of games and presenting some examples of videogames in the blurry classification zone, where we will suggest some distinctions. As this clarification is useful for the sake of objectivity in the remaining of this dissertation, it might also serve other studies where this same uncertainty occurs. Besides this domain identification, we will observe with more detail the design structure of *platformers*. Specifically, in section 3.3, we will look at the level design approaches that one can find inside this genre and analyse how they are used to incorporate story related aspects. In that matter, we will propose a set of quest design patterns, based on a top-down approach for the representation of stories within these games (section 3.3.4). These patterns can serve further automatic generation processes. Lastly, in section 3.4, we will finish this chapter with our concluding remarks.

3.2. Setting the Domain

The concept of *platformer* has not a formal definition and some specific titles fall into a blurry zone where it is not clear if that label is suitable. Naturally, it is impossible to think of a universal categorisation that clusters the design creativity of a certain domain. However, to assess the capabilities of procedural generation algorithms inside this genre, it is important to clarify the boundaries of that same genre to formalise the domain. It is also important to refer that this clarification does not intend to be the reference from the gamer's point of view but to provide a framework for analysis and representation from the scientific side, suitable for the study of game design patterns or automatic level generation. Some principles of the following descriptions are inspired by the work of Smith *et al.* (Smith, Cha, & Whitehead, 2008) and were integrated in the development of a framework for level representation that will be explored later in chapter 5.

3.2.1. Content Description

Avatar

In a platform videogame, the user controls a character in a graphical representation of a physical environment according to certain rules, which can be more or less realistic and even fictional, without compromising the concept. One key aspect of a platform game is movement, so the main task of the user is to control the character's motion. It is important to reinforce that physical realism is not a

requirement. For instance, several considered platform games allow changing the direction of the character in mid-air, presenting exaggerated jumping skills and unrealistic falling capabilities. This is a typical case where the requirement is functional realism (Ferwerda, 2003), where the main objective is to provide a good representation of concepts, using simplified geometry for the parallelism to the real world, in opposition to the more concrete notions of physical or photographic realism.

The physical representation of the character and its meaning does not pose a problem to the idea. Thus, the character can be anything as long as a challenging set of motions is plausible in a two-dimensional scenario. The game *Thomas was Alone* shows that, on the most abstract side, characters can be simple geometric shapes.

The specific movement controls are different from game to game but they normally contain walking and running as horizontal movements and jumping and crouching as vertical movements. Complementary movements can be found commonly in these games, such as wall jumps, climbing and double jumps. The avatar can also be equipped with items that allow extra actions, such as firing a weapon or throwing a rope.

The existence of multiple characters to choose from, or to switch over the game, does not compromise any of the previous principles and is acceptable inside the genre. The different characters may have diverse features and abilities as long as they do not compromise the core features of gameplay. For instance, in some titles featuring *Sonic*, it is possible to play with the character *Tails*, which has a limited flying capability. Even though this ability makes the game easier in some parts, in the majority of the scenario the game is played without using that ability. Therefore, this does not compromise the main concept of the game. In some manner, this type of modifier works as a power-up that slightly changes the capabilities of the avatar, either for a short or a long period that, in its limit, is maintained throughout the whole level.

Lastly, regarding the avatar, a multiplayer platform game is a *platformer* that allows multiple characters on the screen at the same time, sharing the same environment and controlled by different players.

Scenarios

Scenarios in *platformers* represented, initially, two-dimensional worlds on the screen plane. However, the appearance of three-dimensional environments in videogames, especially in the 1990's, expanded the initial ideas to an additional dimension. By this reason, it is common to split the original designation into *2D platformer* and *3D platformer* to reinforce the differences, according to the avatar's range of motion. The domain of this dissertation is strictly *2D platformers* because there are several differences in the game spaces of these two subgenres. The type of challenges involved in the automatic generation of levels is different for the two domains. For the sake of convenience, the designations *platformer* and platform videogame will be assumed hereafter to the specific case of two-dimensional gameplay and the three-dimensional platform concept should only be considered if explicitly said so.

Additionally, the designation *platformer* does not demand that the environment itself is restricted to two dimensions. The restriction applies only to the avatar's range of motion. To clarify this distinction, one can consider the original version of *Super Mario Bros.*, a pioneering title in this genre that is accepted universally over the designation. The recent remakes *New Super Mario Wii* and *New Super Mario U* are still considered *platformers* because, even though all the gaming entities are represented as 3D models, the avatar movements are limited inside one unique plane. In opposition, in *Super Mario 64*, the user can control the avatar in any of the existing dimensions, without any restrictions besides physical impositions, namely gravity, making this title a *3D platformer*.

Moreover, it is plausible to allow depth related information on the reference plane, given that it is more restricted and less recurrent in a level. For instance, it is considered valid under the definition of *platformer* to have multiple layers to represent the scenario. In fact, almost every platform game implicitly represents depth information since the primal approaches, even with strict two-dimensional graphics. For instance, loops in the videogame *Sonic* are a case of an implicit depth representation as they would not be physically possible with strict two-dimensional physics. Indeed, physics of conceivable two-dimensional worlds is a topic of its own, which can be explored apart from this context (Gardner, 1991). Another approach that is common in platform games to simulate depth is to include platforms that only detect descending collisions. These are normally referred as *one-way platforms*. They work as a background layer to where it is possible to jump to but from where it is not possible to fall. It is also admissible to have the player to alternate amongst layers, as it does not represent a crucial aspect of gameplay. An example of this situation can be found in the game *Little Big Planet*, where the avatar can move freely in the screen dimensions and more limited within screen depth, only among three discreet layers.

Gameplay

Scenarios should represent mostly platforms, hence the game designation of *platformer*. Platform is a broad term to describe parts of the scenario that can sustain the character, such as the main portion of the terrain and any other type objects where the user can stand on or use to reach another platform. Therefore, the previous notions about the character must ensure that the concept of platform is present. For instance, a game where the user controls an airplane or a spaceship, moving in a two-dimensional scenario is not a *platformer*, due to the inexistence of the platform entity.

Platforms can have all types of shapes and features as long as they can sustain the avatar. Over time, almost every possible object has been used somehow as a platform in a videogame, from giant mushrooms in *Super Mario Bros.* to clouds in *Little Big Planet*. Platforms can be flexible, such as a trampoline, bendable as, for instance, a tree branch or movable like a floating block. Furthermore, they can be permanent in a level or destructible, whichever established by a time limit, a set of events or after the usage of some item. To complement the existence of platforms, it is also common to have the inclusion of additional movement aids, such as ladders, elevators or any other type of feature that provides a simple moving mechanism that can be used in-between regular movement.

Triggers, which associate a certain feature in the scenario with possible changes in that same scenario, can complement gameplay, having influence in the game strategy and difficulty. For instance, in *Prince of Persia*, several floor tiles act as triggers to open or close gates, allowing or blocking passages, respectively.

In a *platformer*, the main goal in a level is to reach a certain place or entity that defines the end of that same level. Multiple checkpoints can be defined implicitly or explicitly without compromising this idea. For instance, it is plausible to have a level where the user has to visit two characters and only then head to the exiting spot. Optional objectives are also acceptable, such as achieving a certain score by defeating opponents, gathering collectibles and/or reaching the end faster.

The challenge in a platform videogame arises from the level geometry, which should represent a path that is not trivial for the user to complete, and from additional gaming entities, such as enemies and traps. Defeating enemies and escaping traps must be complementary to the main gameplay of moving the character to a certain place and should consist of direct actions. Games with an excessive number of opponents fall out of the category of *platformer*. Naturally, this notion has some subjectivity. One should consider excessive as being enough to change the core gameplay to confrontations instead of moving and jumping. These games are commonly referred with the informal term *brawler*, such as

Golden Axe and *Streets of Rage*, among others. To reinforce that these titles fall out of the *platformer* category, one should observe them without those confrontations. If the removal of the opponents implies that the challenge is lost and thus the game no longer makes sense, then its original form was not a *platformer*.

As a final point, different strategies can be used to promote the challenge and represent implicitly a certain notion of difficulty. For instance, in some games, there is a notion of *life* as being the number of possible fails during a level. In alternative, other games have a less discretised approach and represent *energy* as a buffer of possible penalties during the game. Even though those features have impact on the notion of difficulty, they do not have influence in the game mechanics and, therefore, they do not have implications on the categorisation. Furthermore, time can be used to promote the challenge as well, in short or long terms. A simple example can be found in the game *Prince of Persia*, where the user has a set of levels to complete within one hour. These aspects regarding challenges, with emphasis on the concept of difficulty, will be explored later in chapter 4.

3.2.2. Subgenres

With the diversity of ideas inspired by a common ground, the core concepts of platform videogames have been expanded in multiple directions. We will look at some of the main important differences and features that appear in some titles in order to understand if it is possible to establish a sub-genre or a classification in another genre should be considered. We believe that a way to understand if a certain title fits inside the *platformer* genre, despite the presence of some unique features outside the core idea, is to remove the unique aspects and see if the result is still a game that represents a challenge and can be fit in the original parameters.

One common variety of *platformers* is entitled puzzle platform games, which present enigmas in association to the elements of the scenario. However, one may wonder if these games should be considered *platformers*. Some examples that are commonly suggested for this type are *The Lost Vikings*, *The Cave*, *And Yet it Moves, Trine* and *Braid*, among several others. Considering what has been stated in this chapter, there is nothing to invalidate these games as *platformers*. We have referred the concept of trigger as part of the gameplay in platform games. A strong emphasis on triggers can result in a puzzle *platformer*, as the player has to find out complex combinations of triggers to progress. However, the geometric features of the scenario have the same requirements as a regular *platformer*. The difference lays in the placement of additional gaming entities. Automatically generating levels for these specific cases may pose additional challenges but it is still plausible to consider this as a sub-genre of classic *platformers*, as long as the game contains challenging jumps over platforms.

Another sub-genre that is commonly suggested by gaming press is *run and gun platformers*, where games as *Contra* and *Metal Slug* are pointed. We believe that this classification is comparable to the previously referred case of *brawlers*, but the confrontations with the enemies consist in the usage of guns instead of close combat. Once more, if those confrontations were removed, the results could hardly be considered games because of the lack of challenge. Therefore, we argue that these games should not be considered *platformers*.

An additional separation that is done refers to *single screen platformers*. Observing the features of these games, the distinction does not make particular sense since the way in which information is presented should not imply a different classification. For instance, a zoomed out version of any classic *platformer* would fit one single screen. Moreover, in *Prince of Persia,* the global scene is decomposed into single screen zones that are interconnected with each other. Therefore, the question in these cases is if the label *platformer* fits games with small levels, such as *Lode Runner*. We believe that these titles fit into the classification, as there is still a goal within the scenario to reach a certain place. Besides, if the goal

implies to reach certain items or entities the game still fits the category. For instance, in the game *Bubble Bobble*, all enemies have to be defeated and, in *KGoldrunner*, all coins have to be collected. In both cases, the goal implies moving to certain places in the scenario, so those games fit the category *platformer*.

To conclude, a recent concept in videogames that gained relevance after the expansion of mobile gaming is the (sub) genre referred as *endless running* games*,* such as *Canabalt* and *I Must Run*. These games share several concepts with *platformers* but most of the movement of the avatar is automatic. In *Canabalt*, the character runs automatically from left to right and the user can only interact to make that character jump. In this limited scope, one may wonder if these games can be considered *platformers*. Imagining a possible adaptation of *Super Mario Bros.* where some particular character would chase our hero, it would force the continuous running and the game would still be a *platformer*. In analogy to this case, a racing game is still a racing game even with an automatic acceleration feature. Therefore, the infiniteness factor by itself does not pose a problem in the classification. Even though the objective is not to reach a specific place, the task of moving as far as possible is equivalent regarding the gameplay aspects.

3.2.3. Categorisation

We have seen with the previous examples of *platformers* that the differences among them are significant and that there are several axes to consider when classifying this type of games. One work where this type of classification was considered was presented by Nelson and Mateas (Nelson & Mateas, 2007), integrated in the development of a prototype to design micro-games. The authors identified a set of classic game types and created a system to perceive the type based on gameplay. In our case, a similar classification is intended, even though the focus is on platform games only.

We defined three main aspects to analyse in a platform videogame, based on the most common mechanics:

- **Movement**, which expresses the range of motions that are included in the avatar and the respective control over those motions.
- **Confrontation**, which expresses the importance that is given to a confrontation environment represented with opponents.
- **Environment interaction**, representing additional gameplay features that are not directly related to the original idea of platform videogame, but present additional challenges and a more intricate set of actions.

According to those types of features, based on the identified aspects of a platform videogame presented in the previous subsection and following the main proposals that one can find in gaming press, it is possible to define the following subgenres:

- *Classic platformer*, in which the game contains the essential elements of this type of games, focusing the challenging parts on mastering the movement of the avatar within the scenario. The additional elements might exist but with a very limited influence over gameplay. *Prince of Persia* and *Sonic the Hedgehog* are two examples of this classic approach.
- *Minimalistic platformer*, which reduces the original idea to less controlled characters with automated movement and instant actions with little overall strategy. The game *Canabalt* is an example of this subgenre.

- *Puzzle platformer* takes the initial idea and extends the principles by allowing additional types of actions and environment interaction. Their combination result in puzzling situations. The videogame *The Lost Vikings* is an example of a *puzzle platformer*, in which a strong emphasis on environment interactions complements the different movements and actions performed by the characters. *Braid* is another example of this sub-genre that, however, presents a more restricted set of movements.
- *Action platformer* extends the concept in the confrontation axis, in which the number of opponents tends to be higher. As examples of this subgenre, we point to the videogames *Rick Dangerous* and *Super Mario Bros*. While both games present strong emphasis in confrontations, the later presents a shorter set of actions, only allows running and jumping.
- *Compound platformer*, which complements the gameplay with the referred features of confrontations and environment interactions without focusing any of these aspects in particular, creating a more complex gameplay mechanism but keeping the balance among those features. An example of this subgenre is the videogame *Little Big Planet*.

A summary of these classifications according to the emphasis on the considered features can be observed in Table 3.1.

Movement	Confrontations	Interactions	Classification
Weak	Weak	Weak	Minimalistic *platformer*
Weak	Weak	Strong	Puzzle *platformer*
Weak	Strong	Weak	Action *platformer*
Weak	Strong	Strong	Not a *platformer*
Strong	Weak	Weak	Classic *platformer*
Strong	Weak	Strong	Puzzle *platformer*
Strong	Strong	Weak	Action *platformer*
Strong	Strong	Strong	Compound *platformer*

Table 3.1 – Sub-genre classification for platform videogames according to the emphasis on movement, confrontations and interactions.

To conclude, it is worth mention that some games cannot be classified as a whole, because they have been designed as hybrids. If the game content is decomposed into segments where gameplay differs significantly, naturally they cannot fit one single category. For instance, in the games *The Adventures of Tintin* and *Another World,* the gameplay alternates between genres. Some situations through the narrative are presented as *platformers* but several others are represented differently, as races or puzzles, for instance. In this case, the games are not *platformers* but some specific parts of the content can be referred as platform levels, when those same levels are inside the described domain.

3.2.4. Study Set

In order to potentiate our further studies, a set of popular *platformers* was defined, comprising alternative systems from the early days of computer games to the contemporary times and featuring a wide range of sub-genres, containing the following titles (abbreviations in parenthesis serve further references where size restrictions may apply):

- *Blues Brothers* (BB);
- *Braid* (Br);
- *Little Big Planet* (LBP);
- *The Lost Vikings* (LV);
- *Prince of Persia* (PP);
- *Rick Dangerous* (RD);
- *Sonic* (So);
- *Super Mario* (SM);
- *The Cave* (TC); and
- *Trine* (Tr).

When possible, each of the titles should be interpreted merging the multiple versions of the corresponding game, character or franchise. For instance, *Sonic* refers to all game versions featuring that character, independently of the media, and *Little Big Planet* refers to the respective trilogy.

Moreover, we will consider the following platform games that use automatic generated principles (already referred previously in section 2.4):

- *Spelunky* (Sp);
- *Cloudberry Kingdom* (CK); and
- *Canabalt* (Ca).

The referred titles will be analysed in the next section, where we will observe the game design principles that have been used in this type of videogames.

3.3. Game Design

We have detailed the possible features of a game under the genre *platformer*. Now, we will observe the features inside platform videogames from the game design point of view. Specifically, more than understanding the syntactic validation rules of the content, which were analysed in the previous section, we intend to explore the semantics of that content and the strategies that game designers employ in the creation of those games. Two main perspectives will be given in this aspect. First, we will look at the stories behind platform videogames and their potential for story communication. Complementarily, we will observe the game and level design patterns for this genre.

3.3.1. Platform Videogames and Storytelling Capabilities

(Video) Games and Stories

The way how narratives and videogames are connected is a topic that is still open to debate. Nevertheless, it is undeniable that over the years several videogames have been used as a way to tell stories. In fact, this topic started to catch attention in former years with the appearance of textual adventure videogames, some kind of *interactive fiction* (Niesz & Holland, 1984). The term *ludology* has been proposed by Frasca (Frasca, 1999), as a movement based on *narratology* having different features with distinct merits. Therefore, the main question is not if games are effectively narratives but how they relate to narratives. An interesting approach to this question is a model presented by Aarseth (Aarseth, 2012). Briefly, the author confirms videogames as an interactive fiction medium, whose elements can

be characterised in a ludo-narrative dimension. The different elements can have a more meaningful (narrative) effect or a more challenging and amusing (ludic) importance. Naturally, some videogame genres are more suitable to describe narratives than others are. For instance, sports videogames are not likely to contain a specific story. They can provide emerging narratives (Jenkins, 2004), such as, the epic winning season of a certain team. However, that possible story was created by the user/player while playing the game and it surely does not make him/her a narratologist.

In some extreme cases, videogames are just games without any additional meaning, such as *Tetris*. In this (video) game, the experience is unattached from any additional interpretation or subjacent story. In the opposite side, the videogame can be mainly a story where the ludic factors simply reflect a mimic of a story metaphor. The limit case is something like the referred concept of *interactive fiction*.

Therefore, it is important to understand what one can state about this topic in the specific context of platform videogames. In particular, it is interesting to comprehend how these games have been used to tell stories and what are the main features and issues for this genre regarding storytelling capabilities. These aspects are unveiled in this specific subsection.

Stories and Quests

In order to bring higher abstraction aspects to level generation and provide meaning to the actions performed by the avatar, representing somehow a story within a certain context, it is important to understand the key concepts of plot representation. Initial requirements can be identified in Ryan's (Ryan, 2006) proposal for the conditions of narrativity. The author refers spatial and temporal dimensions as the essential requirements for a broad concept of story. A mental dimension is defined as the next step to include the human experience in stories, where intelligence and emotions are described with agents, goals, actions and events. A final formal and pragmatic dimension expands the former semantic concepts to significance, namely regarding the relations among events and their effective meaning.

Previously, Tosca (Tosca, 2003) identified the notion of quest as "a way of structuring events in games, and (…) incarnate causality at two levels: a semantic one (how/why actions are connected); and a structural one (plan of actions, interaction of objects and events)". The concept of quest is natural in videogames because they tend to represent user incarnations in a certain character, the avatar. This hero (possibly a villain or a team of heroes, as alternatives) centred approach that we find in videogames makes them suitable to be represented as quest-based games.

Sullivan *et al.* (Sullivan, Mateas, & Wardrip-Fruin, 2009, 2010) created a quest browser where they cite Howard's definition of quest as "a goal-oriented search for something of value" (Howard, 2008) and stated that quests "give meaning to the player's actions". The authors also gathered information from prior studies based on the videogame *World of Warcraft* to identify the following types of quests:

- Kill n enemies ($n \geq 1$);
- Kill enemies until n specific items are dropped ($n \geq 1$);
- Collect n specific items from the environment;
- Deliver an item to a certain NPC;
- Talk to someone specific;
- Escort someone; and
- Use a special ability.

Another perspective to look at quests in videogames was proposed by Aarseth (Aarseth, 2005). The author divided the concept of quest into three main types regarding its orientation, which can be:

- **Place**, representing events that have to occur in a specific location.
- **Time**, which represents challenges that are required to be accomplished at a specific time or, alternatively, that are represented by a set of conditions that have to be fulfilled for a certain time interval.
- **Objective**, in which a challenge should be achieved to satisfy a certain premise.

These three aspects can be combined and the following alternative types of quests can be derived:

- Place and time;
- Place and objective;
- Time and objective; and
- Place, time and objective.

A practical approach is presented in the playable quest-based story game *Mismanor* (Sullivan, Grow, Mateas, & Wardrip-fruin, 2012), where the authors applied their theories about playable stories. The main aspect to refer regards the proposed quest structure, which contains a certain intent and different sets of pre-conditions, starting and completion states.

Lastly, a motivating study about quests was presented by Smith *et al.* (Smith, Anderson, et al., 2011). Their work focuses on RPG, namely the patterns used in that genre, which were divided into level patterns and quest patterns. The second type is interesting in the scope of the present work because an adventure *platformer* can share several principles with an RPG. The authors proposed four different categories for quest patterns:

- Actions;
- Objectives;
- Structure; and
- Superstructure;

These categories represent a hierarchic decomposition of content, ranging from "patterns of player behaviour to common methods for storytelling through quests". An additional category, purpose, describes the gameplay reasons for the represented quests.

We will explore how the presented different approaches to describe quests can be merged within the specific genre of platform videogames and how we envision these principles as a way to improve automatic level generation. We share the previous idea of decomposing quest principles within different abstract layers. However, our approach narrows the division into concepts that are more suitable for PCG, as we will see next.

Quest-Based Stories

In *platformers*, the user controls a character or a set of characters in a certain environment with specific goals. If those goals have some meaning, it has to be possible to consider them quests, or parts of quests. For this purpose, it is important to perceive game design patterns for quests in platform videogames. There are different approaches to this challenge, as seen along this subsection. Looking at common situations in this type of videogames, one can think of a *boss pattern* to refer the creation

style where levels end with a confrontation against a stronger opponent, termed *boss*. Similarly, one can identify a *horde pattern* to refer the situations where jump and run gameplay is temporarily substituted by continuous confrontations with opponents in the same space until they are all defeated. It is perceptible that these two potential pattern examples have different levels of abstraction. The first case implies a structure for the whole level, while the second relates only to small parts of the levels. Consequently, game design patterns must be analysed according to an equivalent abstraction level and, as we are focusing mainly the game design patterns that are somehow story related, it is important to emphasise that stories have different abstraction levels as well. We propose the following abstraction levels to decompose a character centred story in the context of game design patterns and automatic level generation:

- **Universe**. The highest abstraction is the vague description of the spatiotemporal context, meaning the type of world and physics, among others. It states when and where the story occurs. *Nowadays*, *Tolkien's world* and *medieval times* are examples of rough descriptors for different possible story universes.
- **Main goal**. Defines the ultimate goal and its main motivation. It explains why further events will happen. *Find the Holy Grail* is a possible example of a main goal.
- **Chapters**. The main goal is naturally divided into sections, normally arranged in chapters in a book or scenes in a movie. In a long story, a grouping mechanism may be used, only for segmentation purposes, without any impact on the story itself. This is equivalent to dividing a same story into multiple books.
- **Quests and side-quests**. They represent a non-trivial task with a certain purpose that must be part of the multiple steps to reach the main goal. By some means, this describes how the main character reaches the ultimate goal. *Find the key and open the door* is a possible example of a quest. A side-quest has similar principles to a regular quest but it does not have direct implications in the story and, typically, it will have a shorter duration. It just endorses the character and his/her personality. Any random heroic task without any additional purpose serves as an example of side-quest.
- **Goals**. Represent the multiple steps to accomplish a quest/side-quest and is something that could be continuously expressed by the user meaning what he/she is doing at the moment. *Defeat the opponent* and *cross the gap* are examples of possible goals in a videogame.
- **Actions**. As a rule of thumb, one can consider an action as something that the avatar can do and that the player can assign to an input button. Some examples are walking, attacking, jumping or running. Actions do not mean anything in particular and can only be interpreted together with other actions, which represent the completion of a goal.

As an example, we may consider the structure on the videogame *Sonic*.

- Universe: Cartoonish-based world with animal characters having humanly-like features (a more complete description could be provided, naturally, using any narrative principles).
- Main goal: Sonic must rescue his friends, kidnaped by his archenemy, Dr. Robotnik.
- Chapters: Green Hill Zone, Lost Labyrinth, etc., which are decomposed into their respective levels, which can be seen as sub-chapters.
- Quests: Escape the exploding power plant, destroy Dr. Robotnik's machine, etc.
- Side-quests: Get 100 rings, gather a secret emerald, etc.

- Goals: Overcome the loop, cross the bridge, etc.
- Actions: Run, jump, spin dash, etc.

Regarding generation techniques, several approaches have already been developed for the bottom-most layers of the model (more details will be presented later in section 7.2). For the top layers, the most common efforts are mainly focused on the narrative and are not suitable to be used directly in an action game. Accordingly, one interesting goal for automatic level generation is to provide an appropriate bridge between the top and the bottom layers.

3.3.2. Ludo-Narrative Analysis of *Platformers*

The widespread idea is that *platformers* do not appear to contain an intense and intricate story. Probably, one of the main reasons of their success is that they are, as already referred, in section 1.1, typical cases of games that are *easy to play but hard to master*, presenting instant action. Normally, they have a smaller set of actions and commands than other games and challenges are mainly related to immediate skill. However, content on these games must play an important role as well, as *Sonic* and *Mario* are two of the most popular videogame characters which surely succeeded in the test of time. Therefore, we want to understand how stories have been included in *platformers* and comprehend how different elements are being used regarding ludic and narrative goals.

According to the *ludo-narrative* model proposed by Aarseth (Aarseth, 2012), world, objects, agents and events are the main storytelling elements in a videogame. Those elements can be more narrative if they are stricter to a certain description or more ludic, if their objective is mainly fun without particular relation to an initial narrative. We have characterised the videogames from our study set (presented in subsection 3.2.4) into that model, as described in Figure 3.1. It is noticeable that the elements of those games have features that are more related to the narrative pole rather than the ludic. More than meaning that these games have elaborated stories, this shows that *platformers* tend to be an expression of a predefined sequence of story related features. Story elements (kernels and satellites) are typically fixed, meaning that the character has an initial strict set of objectives that will lead the development of the whole story over the game. Hence, it is curious to wonder why *platformers* are not associated with stories and others genres are.

The first reason to explain the lack of story association to platform videogames can be found in the categorisation regarding agents. *Platformers* tend to use agents merely as ludic elements, normally in the form of NPC without a rich characterisation. *Little Big Planet*, *Trine* and *The Cave* are the exceptions in our sample. In *Little Big Planet*, especially in the second and third versions of the game, NPC are used as important characters in the story development. *The Cave* shows interesting dialogs and relations between the playable and the non-playable characters during gameplay. Lastly, in *Trine*, there is also a special situation regarding agents. The player controls three different heroes embodied in one single character that can take the physical form of one of them at any time. During the game, the hero's spirits inside the character talk to each other about the adventure that is taking place, exposing their personality. Even though they are not game agents in the common sense of the word, they work as so. Still, this leads us to another aspect that we believe it is important to consider as a relevant element and that should be included in the previous model, which is the hero himself/herself. In this type of stories based on quests, the hero is the centre of all the action and cannot be considered as a simple agent. Considering the hero as a fifth element on the presented model, we propose the following descriptions to define the features from the narrative to the ludic pole:

- A **character with personality** that is presented and explored throughout the game, as the presented case of the videogame *Trine*.

- An **expressive character** that shows his/her current feelings, opinions or suggestions, even though without any particular knowledge about the inner self, such as the *Sackboy* character introduced in the videogame *Little Big Planet*.
- A **living character** that suggests possible feelings but without manifesting them in any way through the game, such as the initial versions of the character *Mario*, which was just a static cartoonish person with a hat and a moustache.
- An **identifiable character** in a somehow adequate level of realism to be majorly accepted as a certain object, such as a car or a spaceship.
- A **simple entity** without any particular physical meaning by itself but that means something by contextualisation, such as the pads of the game *Pong*.
- A **meaningless entity** that plays solely a ludic role, such as the pieces of the game *Tetris*. While simple entities can be interpreted metaphorical as something else, meaningless entities are completely devoid of narrative meaning.

	World		**Objects**		**Agents**		**Events**	
Narrative pole	*Room*		*Static*		*Deep & rich*	Tr TC	*Fully Plotted*	BB Ca CK LBP LV PP RD So Sp SM
	Linear	CK Ca Br BB RD PP SM Tr LBP So	*Static usable*	BB CK Ca Br PP RD So LV		LBP PP	*Dynamic Satellites*	TC Tr Br
	Labyrinth	TC LV	*Modifiable*	SM LBP TC Sp	*Flat characters*	Br So		
	Hub-shaped	Sp	*Creatable*	Tr		Sp LV	*Dynamic Kernels*	
						SM BB RD Ca CK		
Ludic pole	*Open*		*Inventible*		*Bots*		*No kernels*	

Figure 3.1 – Ludo-narrative classification of an example set of platform videogames.

Excluding *Little Big Planet, Trine* and *The Cave*, which bring up the narrative side on the represented heroes, the remaining games of our sample contain a simple living character as the hero. This is another confirmation of the lack of efforts regarding storytelling in those games, as the hero does not present a particular narrative factor.

Another aspect that may justify the referred disassociation from stories to *platformers* is that they tend to present gaming elements that are disconnected from the main story. Typically, *platformers* contain a briefing mechanism when the level starts (a cut-scene, an animation, a small textual description, or others) and a similar debriefing mechanism on the end of the level. However, in-between, where the game action effectively occurs, world, objects, agents and events do not add any information to the initial briefing. Looking at our previous quest categorisation, these games tend to present a universe and a main goal that leads to a set of well-organised small goals and actions.

In conclusion, *platformers* can be used to tell stories and they may contain intricate storytelling aspects. Recent titles like *Trine, The Cave* and *Little Big Planet* are examples of that, as they centre the action in a charismatic character where the world, objects, agents and the sequence of events provide a narrative experience associated to the challenges. This is achieved by decomposing the main chapters into quests. For instance, *Super Mario Bros.* and other pioneering *platformers* do not provide a quest decomposition mechanism. In a certain perspective, this may be seen as a design feature and not as a flaw, because those games were intended to be that way, with a simple background story to contextualise a ludic experience. Over the years, it was possible to notice that those almost purely ludic experiences started to include some quest-based principles and contextualised tasks to create the new concepts of *compound platformers*, as we have previously described in subsection 3.2.3. In the same manner, it is important to follow a similar line of thought in the evolution of automatic level generation for platform games.

3.3.3. Level Design Patterns in Platform Videogames

We have seen in this section that *platformers* were initially created as action games without an intricate narrative. Meanwhile, the game design evolution in this genre has expanded the content richness in this direction and recent titles contain richer characters and agents. In order to create good automatic generation algorithms it is important to understand how humans generate content. In this subsection, we will observe the most relevant academic works regarding game design patterns to understand how levels are created, how the creation process has been identified and described, and how it can be used to make better generation algorithms for *platformers*.

Game design patterns have been used to understand the rules behind the minds of game creators. This concept was introduced as "commonly recurring parts of the design of a game that concern gameplay" (Björk, Lundgren, & Holopainen, 2003). Naturally, each different genre has particular design patterns, which have been studied lately as this topic is becoming popular. For instance, *first-person shooters* (FPS) have been examined by Hullet and Whitehead (Hullett & Whitehead, 2010; Hullett, 2012) and the phenomena of *'ville games* such as *Farmville* has been studied by Lewis *et al.* (Lewis, Wardrip-Fruin, & Whitehead, 2012).

The study of platform games was firstly brought to the academic research community by Compton and Mateas (Compton & Mateas, 2006). They proposed some principles to interpret and describe platform levels in a way that can be used in an automatic generation system. Two main principles were presented: movement patterns and level structure. On the matter of movement, the authors defined a model that contains four different patterns: basic, complex, compound and composite. These represent the possible ways to organise level components, which are platforms and other graphical entities that compose a level. The basic pattern consists of a single component or a simple

repetition. Complex patterns are not well identified, and are just stated as a tweaked sequence of components with configurable parameters. Compound patterns are sequences of basic patterns to be accomplished in a row. Lastly, composite patterns represent the usage of two or more components in a way that they have to be interpreted as a whole. The structure of a level is decomposed into cell structures that represent small parts of a level. These parts are essentially composed by a single pattern, which means that this representation is a flowchart of patterns within a level. Possible variations due to cell combinations are graphically represented in Figure 3.2 and described as follows:

- **Branch** describes situations in which the user can choose one from two (or more) alternative paths in order to proceed.
- **Parallel branch** represents cases in which there are two (or more) alternative paths to choose from, as in the previous case, but knowing that those alternatives will converge to a further common point.
- **Setback penalty** characterises transitions in the path that are a challenge that may cause the user to move to a preceding state in case of failure.
- **Valve** represents cases of strong transition between two states within a level, in which the transition itself is a challenge.
- **Portal** describes transitions in a level that can only be accomplished by using a certain entity, such as a ladder or an elevator.
- **Hub** denotes central positions to where the character has to get back while accomplishing challenges in distinct places, accessible from this central point.

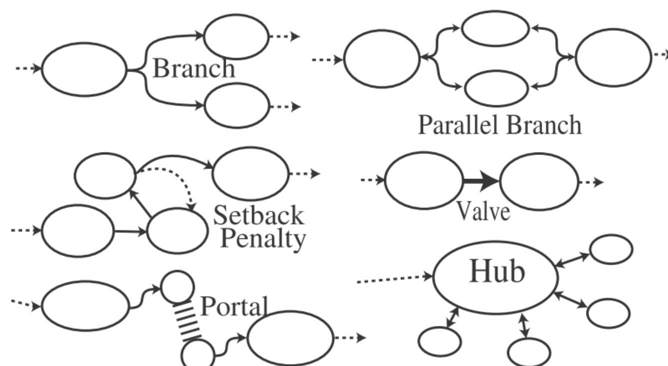


Figure 3.2 – Cell structures for non-linear platform levels (Compton & Mateas, 2006).

The structure of a level has impact over diverse aspects of gameplay. Later, in section 4.4, we will observe with more detail the possible structures of a level, where we will expand these situations. Specifically, we will observe how different structures have impact on the difficulty of a level.

Moreover, the question of game design patterns in *platformers* was also studied by Dahlskog and Togelius (Dahlskog & Togelius, 2012, 2013, 2014a, 2014b) in their work focusing the game *Infinite Mario Bros.* The authors analysed five gaming features, decomposed into multiple alternatives, described as follows:

- **Enemies**, representing the design strategies to create challenges based on the placement of the opponents, with the following variants:
    - Single enemy.
    - Hordes of two, three or four enemies.

    - Roofs, representing situations with a horde of enemies placed underneath hanging platforms, reducing the jump space and making the character bounce between the enemies and the referred hanging platforms.

- **Gaps**, describing the geometric structures of the level in which the avatar has to jump, which can be materialised in the following alternatives:
    - Single gap.
    - Multiple gaps, with common features.
    - Multiple gaps, with variable features, namely gap and platform width.
    - Gap enemy, representing situations where opponents are moving towards the gap, which at the same time provides an aid to the jump but also an additional difficulty to that same jump.
    - Pillar gap, representing situations where the jump over a gap is complemented with the usage of pipes or other types of platforms.
- **Valleys**, on behalf of the situations that force the character to move to lower positions. They can be represented in the following manners:
    - Regular valley, for any region that is delimited by a stack of blocks or pipes.
    - Pipe valley, to represent similar situations that are delimited specifically with pipes with piranha plants, a specific opponent that is common in various *Super Mario* games, which hides inside the referred pipes.
    - Empty valley, meaning than the delimited zone does not contain any opponents.
    - Enemy valley, for those areas that are filled with one or more opponents.
    - Roof valley, for such situations where the valley concept is mixed with the roof pattern that was referred regarding enemies.
- **Multiple paths**, representing situations where the user is presented with alternative routes to proceed, having the possible alternatives:
    - Two-path, for situations where hanging platforms provide two alternative paths, one below and one over those hanging platforms.
    - Three-path, for similar situations to the previous one but with two series of hanging platforms.
    - Risk and reward, to define situations with multiple path alternatives in which one of them includes a challenging element, such as a gap or an enemy, but also provides a bonus entity.
- **Stairs**, depicting the situations where blocks and platforms are used to create steps in a manner similar to stairs, which can be used in gameplay in the following different ways:
    - Stairs up, to represent the situations in which they should be used to move the avatar up.
    - Stairs down, when they are designed to make the character move down.
    - Empty stair valley, to represent specific valleys delimited by these stair structures.
    - Enemy stair valley, to represent similar situations to the previous one but in which enemies are included.

- Gap stair valley, for the situations in which gaps are placed between two stairs.

Despite the fact that the identified patterns provide an interesting overview of some of the aspects that can be found in some *platformers*, two important limitations of this study must be referred. First, the presented patterns are too specific and most of them are only suitable to be used in *Infinite Mario Bros.*, which is a strong limitation for further developments. Secondly, the decomposition of the situations in patterns is not systematic, lacks uniqueness and, in some cases, different abstraction level concepts are presented side-by-side without a proper hierarchic decomposition.

An interesting alternative study but yet related to the topic of game design patterns was presented by Sorenson *et al.* (Sorenson, Pasquier, & DiPaola, 2011; Sorenson & Pasquier, 2010b). The authors defined a system based on a genetic algorithm implementation that creates levels by encoding design patterns into the fitness function. The algorithm by itself is generic for any genre but some possible design patterns are given as an example for the videogame *Infinite Mario Bros.* to show how it could be applied. The authors identified the following six main construction patterns, depicted in Figure 3.3:

- **Block**, which is characterised by its coordinates *x* and *y*.
- **Pipe**, characterised by its horizontal coordinate *x*, its height and a boolean value stating if it contains or not a piranha plant.
- **Hole** (or gap), characterised by its horizontal coordinate *x* and a width value.
- **Staircase**, characterised by an horizontal coordinate *x*, a boolean value stating if the stair are ascendant or descendent and, finally, a width value that can be seen as number of steps.
- **Platform**, characterised by a certain horizontal coordinate *x* and a defined height *y*, having an additional value for its width.
- **Enemy**, characterised by a certain horizontal coordinate *x*, as the provided example only includes one specific opponent type for the sake of convenience.

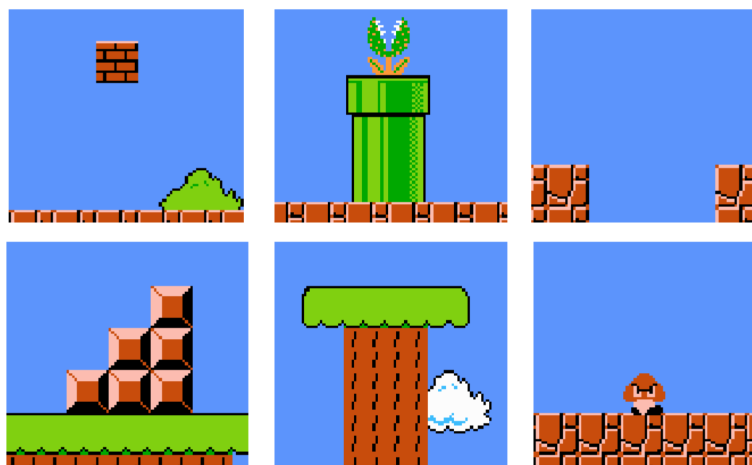


Figure 3.3 – Design elements for the game *Super Mario Bros.* (Sorenson & Pasquier, 2010b), described as block, pipe, hole, staircase, platform and enemy.

These construction patterns are analysed with the definition of constraints based on the following possible rules:

- A *require-exactly* function to state that some design elements must occur in a level an exact amount of times.

- A *require-at-least* function to define a minimum number of occurrences of certain design elements.
- A *require-at-most* function to establish a maximum number of occurrences of certain design elements.
- A *require-no-overlap* function to create overlapping restrictions among the different entities.
- A *require-overlap* to force some design elements to be used only in overlap with other design elements.
- A *traversable* function to ensure that the level can be successfully traversed by the avatar using the game's physical rules, meaning that there are no jumps that are too high or too far for the player to reach.

Once again, the described rules are tailored specifically to one particular game, therefore they cannot be used as general guidelines for platform videogames.

It is also important to refer that in both previous works, the efforts focused essentially on lower abstraction layers regarding the content of a game, depicting only the designers' strategies in a local scale. Whereas lower level abstractions are very precise in describing a particular game, they are not likely to be generalised to other titles of the same genre and they cannot present global design strategies within a level, such as the placement of an object in the beginning of a level to be used later in that level. Moreover, an adventure *platformer* that encourages exploration can get several ideas and principles from an RPG, where higher abstractions patterns have been defined, and still use action based mechanics. To generate game environments that allow this combination, it is important to understand how those higher abstraction patterns can connect to the lower abstraction set of actions.

3.3.4. Quest Design Patterns for Platform Videogames

We have seen that some features may be common to RPG or similar games and *platformers*, especially if we consider contemporary approaches for the latter, which try to include a quest-based mechanism to bridge gameplay with the story. Those quests can be described in different abstraction levels. Top layers represent the game core and it is not our objective to create them automatically. Bottom levels are meaningless actions by themselves and there are already effective approaches to combine them automatically. Our work focuses on the middle abstraction levels, namely the quests and goals. With those layers in mind, we have observed the level content of the games that compose our study set, listed in subsection 3.2.4. This was done by playing some of the levels, observing their structure using image files, viewing online videos of full runs on those levels and with informal discussions with game designers and game related developers. As a result, we extracted a set of quest design patterns for platform videogames, identified and categorised according to their mail goal as follows:

- **Checkpoint**. This is the most common type of quest in platform games and consists in reaching a certain part of the scenario. A reference goal is established for the quest from the following alternatives:
    - **Talk to someone** (e.g. in *Little Big Planet* the main character breaks one of *Zola*'s creation and is told that he should talk to him).
    - **Grab a certain item** (e.g. in *Prince of Persia* the character has to find a sword in the first level in order to proceed).
    - **Achieve a milestone** on the path (e.g. in *Trine*, the characters explicitly state land regions as milestones on the main path).

- **Deliver** an item to someone or somewhere (e.g. in *The Cave*, some NPC ask for certain objects to be exchanged for other items).
- **Escort** something or someone (e.g. in the *Bride Reception* level in *Little Big Planet*, the character finds the groom trapped and injured in a cave and is asked to help him getting back).
- **Escape** a certain enemy or object (e.g. in *Rick Dangerous*, the first level starts with the player escaping a rolling stone ball).
- **Activate** (or deactivate) a certain entity (e.g. in the end of the last level of *Rick Dangerous*, the character has to disable a missile using a computer console).

- **Hostile zone**. This pattern represents a stop to the main movement-based mechanism in a particular part of the level, switching to a confrontation-based mechanism. It may be represented by one of the following goals:
  - **Defeat *n* opponents** (e.g. in *Trine 2* the player occasionally encounter hordes of Goblins).
  - **Defeat a horde of opponents** until a certain condition is reached (e.g. in *Trine 2* the player encounters some situations where enemies are spawned infinitely while he/she tries to accomplish another goal, such as opening a door).
  - **Defeat one particular opponent**, in the typical form of final *boss* (e.g. in the end of each act, *Sonic* has an encounter with *Dr. Robotnik*).
- **Multiple checkpoints**. Refers to quests that represent multiple checkpoints that should be performed in a specific order, such as the following:
  - **Key/lock** situations in which the character has to take a detour to get a certain item that allows further passage at a certain place (e.g. in *Braid*, the character has to grab keys that open certain doors).
  - **Switch/door**, which is a similar situation to key/lock, but in this case the passage is allowed because of a specific trigger. This may lead to a timed challenge, as the trigger might open passage for a certain time window (e.g. in *Prince of Persia*, the character frequently presses buttons that open certain gates).
  - **Composed key/lock**, which are the situations where there is a need to gather multiple items to get passage (e.g. in some community levels of *Little Big Planet* we observed doors that open with a set of different keys).
  - **Composed switch/door** situations represent the need to activate several triggers to get passage (e.g. In *Trine*, some contraptions that are controlled with multiple levers that should be pushed in an appropriate order to open some passages).
  - **Character modifier** situations represent a first stop on a certain entity that somehow changes the characters' features, allowing him/her to reach another checkpoint that otherwise would not be accessible (e.g. in *Spelunky*, the player can get a rope to access additional areas).
- **Puzzle**. A non-trivial combination of game elements to overcome a small part of the level. Puzzle nature, content and difficulty are particularly subjective and we will consider those situations as a quest without any particular decomposition in different types of goals.

We have also identified a set of game patterns to promote additional gameplay, which represent side-quests, described as follows:

- **Grab collectible points** (in the path or in a detour). Represents the inclusion of scoring elements without any gameplay related reward or with a simple impact in the number of tries (e.g. in *Blues Brothers*, player earns an extra life for every 100 gathered records).
- **Reach a secret zone**. Represents the inclusion of areas that are not directly identifiable by the player, and that may contain the following type of reward:
    - **Collectibles data** (game art or videos) that do not have any effect on the game and are mere trophies (e.g. in *Trine*, there are secret arcs that unlock game art).
    - **Collectible powers**, which enhance the avatar features (e.g. in *Super Mario Bros.*, some blocks hide flowers that upgrade the character).

Regarding PCG, quests and side-quests can be interpreted as specific parts of a level, depicting one specific type of narrative motivation, as those that we have presented. They are decomposed into a set of challenging goals, some of which with a specific role to characterise the pattern. In this dissertation, we defend the possibility of generating levels based on this quest decomposition. Accordingly, the algorithms for different quest types should be different, as they represent different structures. For instance, the pattern *multiple checkpoints* requires the level to have a branched structure, a condition that does not have to be fulfilled in other patterns. Later, in chapter 7, we will explore this idea, categorising the different types of generating algorithm (section 7.6) and showing a prototype in which the user defines a set of quests that guides the generation process (section 7.7).

3.3.5. Multiplayer Design Patterns

In the previous two sections, we have seen different studies regarding game, level and quest design patterns for videogames, focusing essentially single player mechanics. However, multiplayer gameplay is one aspect that has been evidenced in recent platform videogames. Titles like *Little Big Planet* and the *New Super Mario Bros. Wii* are examples of that. In fact, this trend is applicable to almost every genre nowadays and the social role of videogames is increasingly being explored. Thus, it is also important to analyse the main references regarding game design for multiplayer gameplay.

Existing Approaches

In the subject of multiplayer design patterns, two main complementary articles should be mentioned. The first is a study presented by Rocha *et al.* (Rocha, Mascarenhas, & Prada, 2008) that consists in the identification of the main mechanics that can be promoted in multiplayer games, namely:

- Complementarity of roles;
- Synergies between character's abilities;
- Existence of abilities that can only be used on another player;
- Goal sharing;
- Synergies between goals; and
- Usage of special rules.

The second work was presented by El-Nasr *et al.* (Seif El-Nasr et al., 2010) and extends the previous set of patterns with additional reference situations, namely:

- Camera settings;

- Shared and multiple object interaction;
- Shared puzzles;
- Shared characters;
- Special characters targeting lone wolves;
- Vocalisation; and
- Resource constraints.

Multiplayer Design Patterns in Platform Videogames
To conclude our study regarding different game design features for platform videogames, we have also analysed their multiplayer aspects. It was possible to observe the majority of the multiplayer situations proposed in the existing literature that we have just presented. However, they concern gameplay features and do not relate directly to the representation of quests. For instance, we observed synergies among characters in *Little Big Planet* and *Trine*, but those are manifested all along the levels and quests were designed without that in mind. Regarding the design of quests, we have identified two particular types of situations that can be seen as quest patterns for multiplayer gameplay, which can be described as follows:

- **Players' co-dependency**, representing situations in which the characters are dependent of one another in order to complete the challenge (e.g. in *Little Big Planet*, some areas are only accessible if multiple buttons are pressed at the same time, mandatorily by multiple players). Regarding general multiplayer design patterns, this concept is inspired by the principle of synergies between character's abilities and complementarity of roles, which are presented as essential requirements. One can say that players' co-dependency is a quest generation pattern that promotes multiplayer gameplay mainly through synergies between character's abilities and complementarity of roles.
- **Expert and beginner integration**, representing situations that are designed specifically to be played by two players of different skill levels (e.g. several challenges in *Trine* are designed to allow one player to perform the most complex part of task easing the level to the other one). While this may seem an adaptation of the previous pattern, the difference is that this case represents a direct dependency instead of a mutual one. Here, the weaker element requires the skill of his/her leader to help him/her accomplish the task.

These patterns complement those presented in section 3.3.4, in order to create quests with situations that are specific for multiplayer gameplay.

3.4. Concluding Remarks

In this chapter, we have detailed the specific genre of platform videogames by characterising extensively the content of these games. Accordingly, we have analysed different gameplay features, namely movement, confrontation actions and environment interactions, in order to understand when one should consider the categorisation within the genre and when the game falls outside its boundary.

Moreover, we have studied the storytelling capabilities behind platform videogames to identify possible improvements on the existing automatic level generation techniques for this genre. Content in *platformers* tends to be structured in a way that should be considered more narrative than ludic, as they present specific quests with certain characters, in scenarios that are mostly composed by elements that are directly related to the game progress. Still, these games do not typically contain rich stories

because storytelling elements are not explored and, as stated in subsection 3.3.2, NPC have not been used properly for that purpose.

Regarding level design patterns, we have identified that the story represented in a videogame can be decomposed into different abstraction layers and level design patterns refer to a certain layer. Different *platformers* were analysed to identify the main containing structures where quest principles were used. A decomposition model was proposed, based on observed situations and recalling existing approaches to other genres. The final result is a set of quest design patterns, which include side-quests and multiplayer quests at the same level of abstraction. These patterns can lead automatic generation processes to a more contextualised output, as we will see later in section 7.7. We will observe that lower abstraction algorithms presented in some other works can be integrated to generate level sections where a certain quest is represented.

4. Human Factors

"Playing a game is the voluntary attempt to overcome unnecessary obstacles."

(Suits, 2005)

4.1. Introduction

Human factors have been studied in multiple contexts from distant times. In this chapter, we will focus on the human factors that are related to games and gameplay in order to perceive important aspects that can enhance automatic level generation algorithms. The usage of computers for entertainment changes the initial computational paradigm directed to calculations and processes represented as black boxes that transform one or more inputs into a set of results. The act of playing a game is a different kind of interaction between humans and machines.

Likewise, the context of using a common application is different from playing a videogame, as "players seek entertainment and the challenge of mastery, while application-system users focus on the task and may resent forced learning of system constraints" (Shneiderman, 1987). For these reasons, it is common in game design literature to have a distinction between interface and gameplay (Juul & Norton, 2009). Initial approaches focused mostly on usability rather than player enjoyment. They tended to observe mainly the interface, the mechanics and the gameplay (Federoff, 2002). However, contemporary user-testing is willing to adapt the mechanisms to include additional features related to human factors, expanding the concepts of usability to the perception of user-experience. Desurvire *et al.* (Desurvire, Caplan, & Toth, 2004) adapted studies about video, computer and board games, and grouped them in order to define a set of heuristics for evaluating playability. These guidelines present several design principles grouped in gameplay, story, mechanics and general usability. Still, the borders between interface and gameplay are not always clear, which poses some challenges in this specific aspect.

The study of gameplay metrics is becoming increasingly important and performed by relevant companies such as *Eidos* (Drachen & Canossa, 2009) and *Valve* (Booth, 2009). More than identifying the general player profile, they intend to have a better perception of what challenges the players and how certain game events may have influence over their emotions.

The remaining of this chapter is divided as follows. First, in section 4.2, we will explore the concept of playing a game and observe some of the main features already identified in this topic, namely the referred importance of users' emotions. Afterwards, in section 4.3, we will direct our attention to the specific notion of difficulty. Mainly, the contributions in this topic focus on estimating or adapting difficulty. Therefore, in subsection 4.3.1, we will observe multiple approaches to measure difficulty in different videogame genres, other than platformers, and, in subsection 4.3.2, we will observe the main concepts explored regarding difficulty adaptation. We will address difficulty in the specific genre platformer in section 4.4, where we will also include our proposals for its estimation, in different types of challenges. The concluding remarks of this chapter will be presented in section 4.5.

4.2. Playing Experience and Emotions

As a science, Psychology was born in the end of the 19th century, though the philosophical studies in that area can be found back in ancient civilisations. One popular connection between Psychology and Computer Science can be found in Csikszentmihalyi's concept of *flow* (Csikszentmihalyi, 1991), which describes a state of mind where a person is completely focused on a task with a sense of immersion and full control. The following aspects characterise the state:

- A challenging activity requiring skill;
- A merging of action and awareness;
- Clear goals;
- Direct and immediate feedback;
- Concentration on the task at hand;
- A sense of control;
- A loss of self-consciousness; and
- An altered sense of time.

This concept is valid for almost every domain of actions, being commonly used in HCI and, specifically, in videogames, as supported by Chen's work (Chen, 2007). Nonetheless, the specific case of fun in videogames has been studied from distant times, as it is possible to observe in Malone's prior works (Malone, 1980), where some key aspects about gameplay have been identified. Three key aspects have been established for good computer games: challenge, fantasy and curiosity. One common term is found both in these aspects and in *flow*'s characteristics, which is the existence of a challenge. Naturally, this core feature of videogames is very important to consider when generating game spaces, so it is relevant to match game challenges to players' skills. This relationship can be referred as *difficulty*. An easy task is one where skills are meaningfully higher than the challenge and, on the contrary, a hard task is one where the skills are significantly lower than the challenge. The act of playing a game involves several emotions and some other interesting human factors play an important role, as we will see next, before digging into the notion of difficulty.

Sweetser and Wyeth proposed the concept of *game-flow* (Sweetser & Wyeth, 2005), an extension of the initial concept of *flow*, with emphasis on the domain of videogames. Specifically, the authors presented *game-flow* as a model to design, understand and evaluate player enjoyment in games, merging into one single framework the heuristics and concepts of earlier approaches.

Komulainen and Takatalo (Komulainen & Takatalo, 2008) presented a psychologically structured approach to user experience in games. Their work is a questionnaire-based study in which they tried to identify some of the most relevant emotions behind the act of playing a videogame. In the top experience descriptors, we can find terms such as enjoyment, excitement, relaxation, entertainment, success and challenge, among others. Their work also provides a categorisation of those experience descriptors into cognition, emotions and motivations, the three key elements in mental activities proposed by Hilgard (Hilgard, 1980). Regarding cognition, the authors identified that the challenge can be represented in three different types, namely a challenging task by itself, a competition or a problem solving assignment. These can be overcome with success, winning or discovery, respectively. Additionally, the process of overcoming a challenge is described to be rewarded with knowledge as a result of a learning process or with progression enclosed by a larger task. Moreover, regarding emotions

and motivation, the authors analysed expressions that can relate directly to *flow*. For instance, they identified terms as frustration and boredom in opposition to enjoyment, excitement and inspiration.

The concept of entertainment is by itself an emotion, so it is natural that, in the development of software in this field, emotions play an important role with direct impact on user engagement. We will now observe the different types of emotions that have been studied in this area and we will observe how those have been assessed.

First, regarding emotions, it is important to start with the basis of game design and the efforts that are directed to this specific feature. Accordingly, some applications, tools and plugins have been developed aiming the design of emotions, in which game content is structured taking that into account. For instance, Zagalo *et al.* (Zagalo, Prada, Alexandre, & Torres, 2008) presented a plugin for *Inscape* and *Teatrix*, two authoring tools, in which stories are created with explicit support for emotional responses. As another example, *InStory* (Barrenho, Romão, Martins, & Correia, 2006; Correia et al., 2005) is a mixed-reality system that explores the social side of emotions.

With the importance of emotions in game design, one may wonder which emotions are more relevant to understand in gaming context and how those can be analysed or measured. For this purpose, Mandryk and Atkins (Mandryk & Atkins, 2007) presented a method to model emotion recurring to fuzzy logic and using physiological data as input. Specifically, the authors analysed galvanic skin response, cardiovascular measurements and electromyography in order to detect states of boredom, challenge, excitement, frustration and fun.

Regarding the specific genre of platform videogames, Pedersen *et al.* (Pedersen, Togelius, & Yannakakis, 2009a, 2009b, 2010) studied users' emotions in the game *Infinite Mario Bros*. They investigated how content parametrisation have influence over different components of player experience. To start with, the authors tuned an existing level generator in order to allow control over the following parameters:

- The number of gaps in the existing level.
- The average width of the existing gaps.
- The width variance of the gaps, normalised into values from 0 to 1.
- The number of direction switches, representing imposed situations in which the level representation is mirrored from left to right or vice-versa.

Besides those controllable features, the authors collected gameplay information, such as the number of jumps and the gameplay duration, and performed a simple questionnaire to detect the existence of fun, challenge and frustration, a subset of the aforementioned emotions presented by Mandryk and Atkins. They claim to have good predictors for the final emotion based on the measured data, especially frustration. However, it must be referred that these predictions are mainly obtained through gameplay data and are not reliable when considering only the controllable features of a level. This means that it is only possible to estimate the emotions after observing how users behave in the level.

In continuation of the previous work, Shaker *et al.* (Shaker, Yannakakis, & Togelius, 2010) explored in more depth how the modification of certain aspects of game content can affect players' in-game experience. For that purpose, the authors expanded the collected data set in which they have also included gameplay data from sessions using agent-controlled characters. An online game adaptation mechanism was implemented in order to adjust level content to the obtained results of prior sessions, which, in this case, can be performed by agents. As this experiment presents an initial expansion to

the prior concepts and shows that content can be adapted to certain profiles, it is still sustained by acceptable accuracy values and weak design configurations, which opens new ideas for further developments. Additionally to this work, the authors (Shaker, Yannakakis, & Togelius, 2011) also explored alternatives for data extraction, namely by applying overall statistical values in conjunction with common feature sequences. More recently, those same authors (Shaker, Yannakakis, & Togelius, 2012) presented another extension to their work in which they detected that different emotional states require different session sizes.

Following the previous studies about identifying emotions in gameplay, a line of work that is becoming more popular in PCG is entitled *experience-driven PCG*. A survey about this topic can be found in the work of Yannakakis and Togelius (Yannakakis & Togelius, 2011). *Experience-driven PCG* is presented as the "personalisation of user experience via affective and cognitive modelling, coupled with real-time adjustment of the content according to user needs and preferences".

To conclude the background of experience and emotions, it is important to refer the work of Sorenson and Pasquier (Sorenson & Pasquier, 2010a), in which the authors guide an automatic level generation algorithm with the concept of fun, associated to the represented challenge within a level. For that purpose, an estimator for anxiety during the level is established. This estimator is obtained with the integration of the challenge over time and applying a decay factor that lowers the values in calm locations.

4.3. The Concept of Difficulty

The notion of difficulty is an actual interesting human factor to analyse in automatic level generation. In order to understand if a level is challenging, one has to perceive the represented difficulty and quantify it. However, describing this kind of abstract notion is not a direct task. Furthermore, creating levels automatically allows generating content that is specific for a certain user profile. The automated generation can provide levels for a certain difficulty and try to fit other user characteristics, such as genre, age or any others. This directs the content generation to focus on the user experience and, in particular, to the previously referred notion of *flow*. Still, adapting the challenges might not be just a matter of casual versus expert gamers and it can also be about accessibility. For instance, one interesting work to refer inside this question was developed by Oren (Oren, 2007), consisting in the adaptation of platform videogames in order to make them suitable for visually impaired people, where the main features are represented with audio events. In fact, the reduction of the boundary between videogames and users with disabilities is a goal of contemporary research. A common reference of this topic is the idea of universally accessible games (Grammenos, Savidis, & Stephanidis, 2009), "proactively designed to optimally fit and adapt to individual gamer characteristics and to be concurrently played among people with diverse abilities, without requiring particular adjustments or modification".

An informal overview over the design patterns regarding difficulty has been presented by Nicollet (Nicollet, 2004), where some interesting insights are exposed. Even though some of the principles are arguable, this is an interesting starting point to look at this subject. Particularly, we would like to reinforce the following rules about representing difficulty:

- Surprise is not the same as difficulty, meaning that the existence of random events without a defined pattern does not contribute in making a challenge harder. One may argue that surprise makes the task harder but if the corresponding surprise factor cannot be mastered then the challenge does not have a background motivation.

- Difficulty implies possible and probable failure, which has to be avoided by applying specific skills.
- Reducing the time windows of an action or a set of actions increases difficulty.
- A continuous sequence of actions has greater difficulty than the same actions performed independently, with structural separations in-between each of those actions.
- Reducing the player control over the character generally increases difficulty.
- Increasing the precision to overcome a challenge increases difficulty regarding that specific challenge.

Moreover, Juul's work in this topic (Juul, 2009) expands in a more formal way the aforementioned principles. The author explored the popular design patterns to represent difficulty in gameplay. Besides, the implementation of failure is described as being represented in the following manners:

- Setback punishment, forcing the user to replay a certain part of the level.
- Energy punishment, which serves as a warning of a stronger penalty, even though it does not imply direct penalty at the moment. Possibly, it may produce a setback penalty but this feature is not required.
- Life punishment, forcing a setback in the continuous gameplay sequence to a certain check-point and making the gaming session closer to its end.
- Game termination punishment, meaning that the player completely loses the challenge and has to restart the game from the beginning.

Nowadays, depending of the game genre, the usage of game termination punishment is less recurrent. This approach was more popular in early games, especially because gaming sessions were independent, without any possibility of saving a certain state. It is important to understand that this was mainly a technical limitation and not an intentional design strategy. The possibility of saving states in a gaming session and restoring them later changed the perspectives about punishment, resulting in new design strategies. A simple approach is to decompose levels or challenges into chapters, and the game termination punishment forces the player to restart only that chapter. As an example, this is possible to observe in recent *Sonic* games. Once the player achieves a certain level, that same level is unlocked for any further gaming sessions. A broader alternative is the possibility of saving the game state anytime and restoring it later. This is a more flexible solution for the user but it allows unwanted usages that might break the challenge. The players can save the game constantly to avoid any type of punishments.

Another approach regarding difficulty was presented by Aponte *et al.* (Aponte, Levieux, & Natkin, 2011a, 2011b). The authors defined a generic mechanism for assessing difficulty starting by splitting gameplay elements into measurable items. Their approach is directed to the usage of probabilities of success and failure to analyse individual challenges. In order to use probabilities it is important that the challenges meet the following criteria:

- The difficulty must be measurable;
- The different difficulties of similar situations must be comparable; and
- The difficulty must be related to a certain progress in story and/or skills.

This idea of using probabilities is also part of our work in this topic and will be presented below in section 4.4. Furthermore, the authors identified that difficulty can be controlled with two different methods. A first alternative to make a challenge harder is by making its features somehow harder for the user. For instance, an opponent might move or shoot faster. Alternatively, challenges can be made harder with the creation of composed challenges, in which the user has to combine techniques that were mastered independently. For example, distinct challenges of jumping over gaps and avoiding certain entities might be combined, creating a much harder challenge.

4.3.1. Difficulty Measurements and Approaches in Other Videogame Genres

We have seen difficulty as a human factor regarding gameplay that maps the users' skills to the represented challenges, commonly related to a certain probability of failure. Different game genres have distinct types of challenges, having notions of failure that are not alike. Therefore, difficulty has to be assessed also differently. In this subsection, we will overview some of the existing approaches in different genres.

Togelius *et al.* (Togelius, De Nardi, & Lucas, 2006, 2007) presented a system to automatically generate racing tracks for a driving simulator. To evaluate the quality of each track, for usage as fitness function in a genetic algorithm, the authors used artificial drivers, mapped to a certain profile, to extract some attributes such as timings and speeds. Those attributes are mapped to the referred fitness function in order to represent a simplified notion of fun, which combines an adequate amount of challenge and respective variance. A final heuristic is included to require the existence of multiple segments where it is possible to achieve high speeds.

Another interesting work to refer was developed by Pereira *et al.* (Pereira, Santos, & Prada, 2009). In this case, the considered genre was *strategic multiplayer browser game*, which consists in a slow-paced evolution system, accessed by players a few times on one day to establish some strategies about virtual resource management. The system tries to involve the player in an ambient that fits the user preferences, with a balanced distribution of resources to avoid repetition. The authors refer the importance of efficiently handling the constant appearance of new players in the earlier stages of the game.

Another important idea that spans different genres is the possibility of constantly adapting the challenge during gameplay. This concept is commonly referred as Dynamic Difficulty Adjustment (DDA) and it will be explored in the next subsection.

4.3.2. Dynamic Difficulty Adjustments

When the users start playing a game, they have different skill levels. In a possible general learning curve for a certain game, they can be placed in different positions, as their experience in other games serve as knowledge for the experience they are currently engaging. Besides, different players have different learning rhythms, as learning itself is one of the skills to put into practice. These are some of the motivations behind the principles of DDA. Moreover, multiplayer games present further challenges regarding difficulty and possible adaptations. Naturally, in top competition, the ultimate goal is to be the best at a certain task. However, in lower competition levels such as training, learning purposes or simply to promote social experiments, the way difficulty is handled can be different. For instance, in sports, golf uses a handicap system to even matches with players of different skills. Furthermore, when parents play a game with their children they intentionally make mistakes to even the game. Likewise, in cooperative tasks, parents tend to play as guardians, implementing their inner protective role (Dalsgaard, Skov, Stougaard, & Thomassen, 2006). Next, we will explore the challenges that we face in this topic and observe how these ideas have been transposed and used in videogames.

Single Player DDA

Hunicke (Hunicke & Chapman, 2004; Hunicke, 2005) presented a system entitled *Hamlet* where DDA is applied to FPS. They use a set of probabilistic studies to identify the appropriate time to intervene, meaning essentially the ideal time to help players in struggle by providing additional energy or ammunition.

In a different genre, *real time strategy* games, Olesen *et al.* (Olesen, Yannakakis, & Hallam, 2008) created an AI opponent using an evolutionary algorithm that adapts to the users' strategies. The adaptation capabilities are guided by a set of challenge related parameters, meaning the agent is evolved following those guides, resulting in a balanced opponent to the user.

Finally, regarding platform videogames, it is important to refer *Polymorph* (Jennings-Teats, Smith, & Wardrip-Fruin, 2010a, 2010b), a DDA system that employs machine learning to understand difficulty and players' skills in the process of level generation. As stated, when the users start playing a game, they begin with different skill levels and thus are not ready for the same kind of challenge. Moreover, in the beginning, users tend to prefer lower difficulty challenges while they get used to the playing mechanisms (Klimmt, Blake, Hefner, Vorderer, & Roth, 2009). Polymorph implements a data gathering mechanism based on small sessions of user interaction, using level segments. Segments are tagged manually by the users, in a Lickert scale from 1 to 6 describing their perception of difficulty in that same segment, from very easy to very hard. Furthermore, the system gathers direct gameplay data such as the "amount of time the player spends standing still or moving backwards, the total completion time of the level segment, the number of coins collected, and whether the player died or completed the segment". In the end, the existence of certain elements and their combination are correlated with the identified difficulty in the level segments. Those correlations allow the creation of continuous levels with increasing difficulty perceptions.

Multiplayer DDA

The previous examples show how to adapt the challenges that are represented in a videogame, in order to make them suitable to a certain player or group of players. Still, the main principles can be applied to the game mechanics when the challenge arises from a direct competition among multiple human players. One practical application of this concept that can depict the main idea was used in a simple *Pong*-like game in order to keep the match levelled between two players of different skills, proposed by Ibáñez-Martínez and Delgado-Mata (Ibáñez-Martínez & Delgado-Mata, 2009). A simple algorithm is defined to react to three main unwanted cases:

- Excessive easiness for both players that results in mutual boredom;
- Excessive difficulty for both players that results in mutual frustration; and
- Uneven matches that produces boredom in one side and frustration in the other.

As a general rule, if a player is performing much better than the other, then the game should adapt itself. This adaptation should lower difficulty to the weaker player if the game is being intense or increase difficulty in the better player if the game is being slow paced. Besides, if the system detects that the game is being excessively easy or hard for both players, consisting of high and low ball movements, respectively, even modifications are applied. The proposed solution is simple but yet effective.

Although it may seem directly interesting for players, some drawbacks of this approach need to be addressed. As stated by Hunicke (Hunicke, 2005) players might "feel cheated if games are adjusted

during or across play sessions". This type of adjustment was commonly performed in racing videogames by, for instance, increasing acceleration capabilities to cars in former positions. As this is an obvious implementation of the idea of DDA, players can notice it and use it as a strategy.

In addition, multiplayer gameplay does not imply a direct confrontation among players. Cooperative games are not only a way to promote socialisation but also an important form of multigenerational bonding. In family context, this is good way to "engage parents with their children's thinking, character development, and learning" (Siyahhan, Barab, & Downton, 2010). Still, the design of suitable games for intergenerational gameplay is not an easy task, as "different combination of game design patterns can lead to different cooperative behaviour" (Barendregt, 2012).

4.4. Difficulty Measurement in Platform Videogames

Regarding difficulty representation, we share the idea of other authors (Aponte et al., 2011a, 2011b) that it has to do with probabilities of success and failure while performing tasks, either when they are analysed independently or within a whole level. Our approach is based on two different types of level analysis:

- In a first step, a level is decomposed into segments that represent distinct independent parts and possible transitions among segments are identified. This means that the challenge is decomposed in multiple steps to achieve the goal, in which different alternatives might be available.
- The second step consists in the analysis of each small component individually, based on the probability of success of those components, which is directly related to their features.

The two presented types of level analysis are intended to be consecutive, as we will explain next.

4.4.1. Overall Level Structure Difficulty

This first step involves understanding the main level structure and how individual challenges and groups of tasks are positioned in that structure. A level is composed by several individual small challenges, like specific jumps, opponents and other elements. However, their organisation within the level has impact in the overall notion of difficulty. For instance, a sequence with all the existing challenges is naturally harder than splitting them in two alternative paths. Besides, failing some of the challenges may force the user to repeat some level parts, which may result in additional difficulty. In order to understand the impact of such arrangements and the important parameters to analyse, we present a set of example situations that one can often find in *platformers*, depicted in Figure 4.1 and described as follows:

a) A straight path to the end of the level, where it is impossible to fail. As the user knows his/her task, the technical efforts regarding gameplay are meaningless. This situation allows multiple variants with the definition of a variable for the distance between the starting and the ending point, represented as d.

b) A single try challenge with a certain difficulty, represented as a jump over a gap with a configurable size s. The user automatically fails the level if he/she falls into the represented gap.

c) A challenge attempt consisting of a jump over a gap, comprising a setback penalty in case of failure, materialised with the following example alternatives:

    1. A jump over a gap in which the user may retry as long as he/she wants. The size of the gap, s, is again configurable. This setback is equivalent to the situation of example *a)*.

2. A jump over a gap in which the user may retry after failing, but each retry has an additional challenging task equivalent to the example situation *b)*. The parameter s_1 represents the size of the main jump and s_2 represents the size of the jump that the user has to perform as setback penalty.

As stated, case *a)* represents a trivial situation in which the user has to accomplish a task where it is impossible to fail and, thus, his/her skill do not have impact in the challenge. A simple state chart can be used to represent this situation, considering a mandatory transition between the initial and the final state, which occurs with a probability value of one, as presented in Figure 4.2 (a_1). However, if we increase significantly the value d that defines the distance from the beginning to the end of the level, the user might not succeed because he/she does not have the motivation to fulfil the task. Recalling the notion of *flow*, the user is facing a situation of excessive boredom. At the moment, we can relate that as an emotional state of will to continue throughout the level, defined as $w(t)$. It can be interpreted as the probability of continuing to play at a certain instant of time t. The adapted state chart for this situation is represented in Figure 4.2 (a_2).

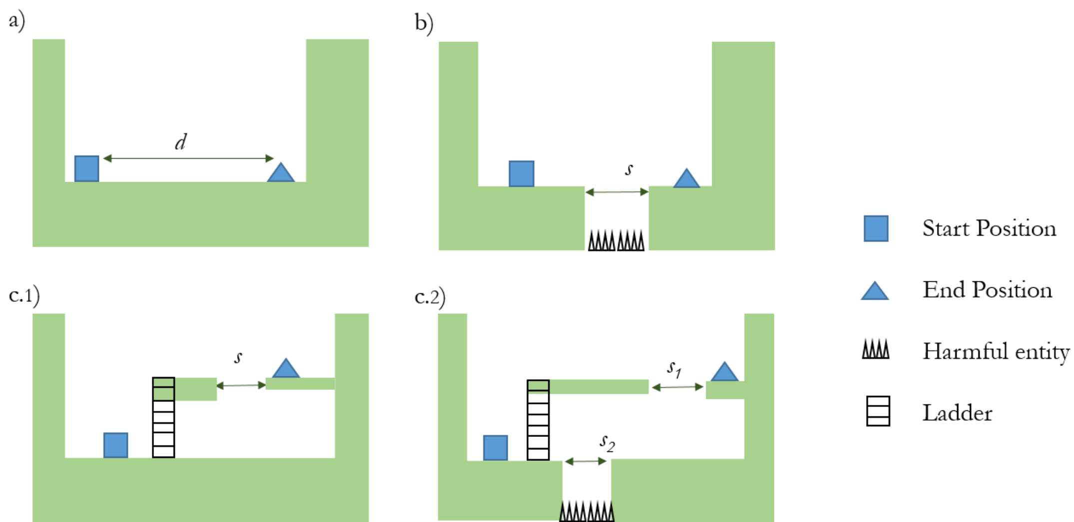


Figure 4.1 – Example of common challenge representations in platform videogames.

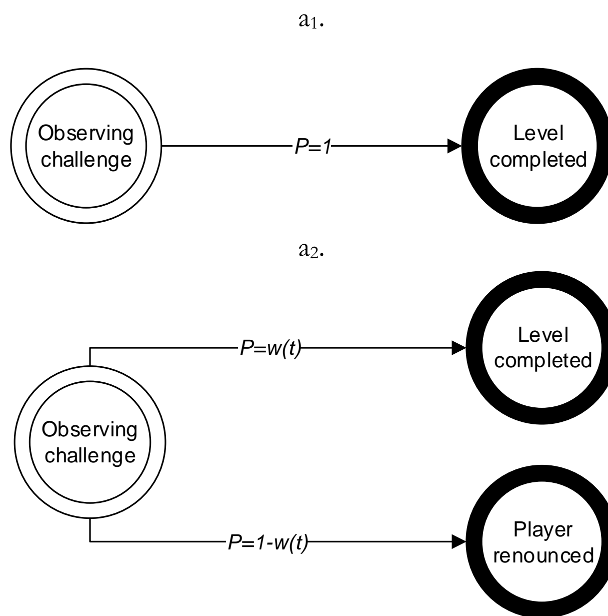


Figure 4.2 – Example platform videogame situations. State chart for situation *a)*.

Situation *b)* represents a case with a direct challenge, represented as a gap. It is intuitive that if the gap is larger, then the probability of success in that gap is lower. Mathematically, this means that the referred probability is a predictor of success, defined as a function of the gap size, so we will represent it as *pj(s)*. In limit cases, we have the easiest situation with a gap size of zero, where the probability of success is one, and the most difficult situation occurs with a success probability of zero, for every value of *s* over the jumping capability of the player's character. For the values in-between the limits, an estimator for the probability based on the player's behaviour is required. That question will be addressed below, in subsection 4.4.2, when focusing on the specific challenges individually. Graphically, we can represent this situation again with a state chart, as presented in Figure 4.3 (b_1). With the previous notion of will to continue, one can assume an initial state where the user decides if he/she really intends to try the represented challenge, making the situation representable as shown in Figure 4.3 (b_2).

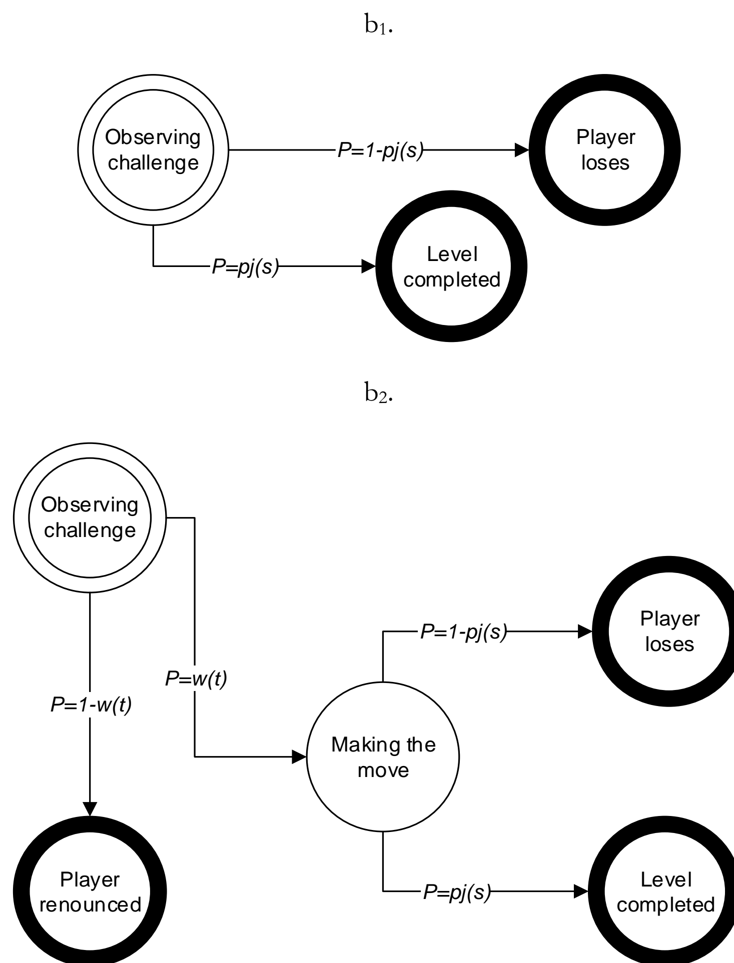


Figure 4.3 – Example platform videogame situations. State chart for situation *b).*

In the last two sub-situations, *c.1)* and *c.2)*, we present again the challenge of jumping from one platform to another one but failing the jump does not harm the character, and the player is able to retry that same jump. In case *c.1)*, regarding the jump, the situation is identical to the example *b)*, with a probability of success that is represented as *pj(s)*. However, in this case, the player can retry as long as he/she wants. Whereas this means that the user will eventually succeed (as long as the jump is accomplishable), the practical case is that, after failing an attempt, the user might choose to stop trying and finish the gaming session. Therefore, continuing depends on his/her will to continue. The overall situation regarding this example can be observed in the state chart that is presented in Figure

4.4. The resign conditions are more complex to identify in comparison to the initial cases. The user might quit because he/she feels that the challenge is too hard, because he/she is tired of the process of going back and performing the jump all over again, or any other reasons. In addition, those reasons might not be constant in a level. For instance, in the beginning of the level, the player is less mentally tired, possibly having a higher will to retry the challenges. This reinforces that the concept of will to continue is time dependent and should be related to a predictor of the different emotional states throughout the level. In case *c.2)*, the retry process is also related to skill and it may cause the player to lose. The situation is depicted with the state chart presented in Figure 4.5.

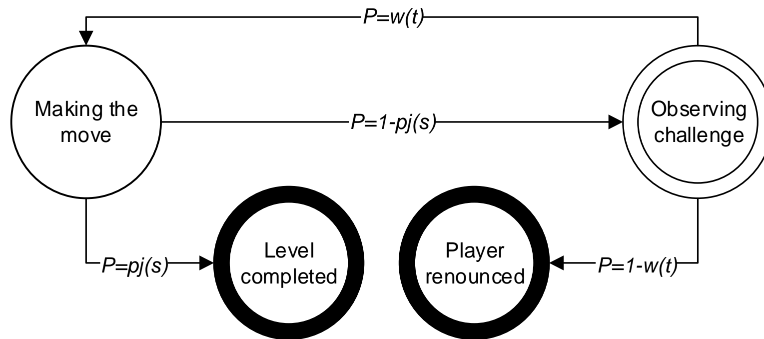


Figure 4.4 – Example platform videogame situations. State chart for situation *c.1)*.

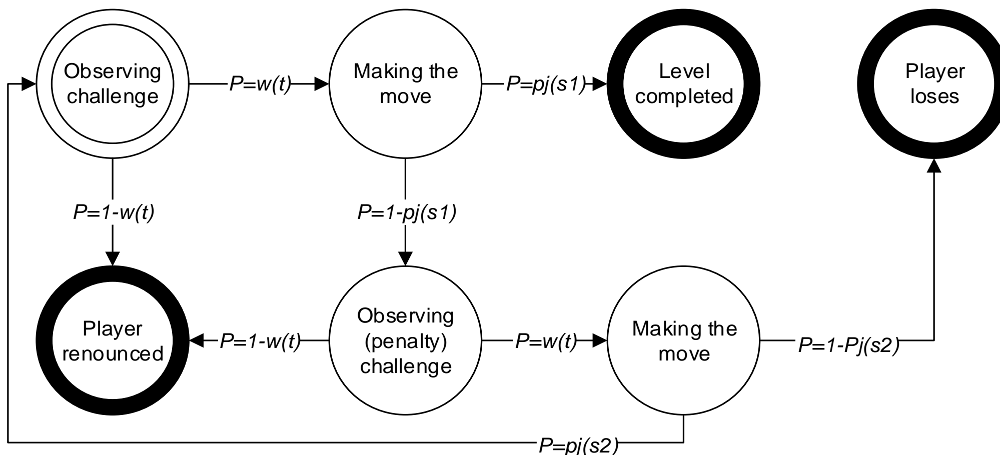


Figure 4.5 – Example platform videogame situations. State chart for situation *c.2)*.

Despite that, in the last two sub-situations, we have represented the setback penalties as simple situations inspired by the first two cases, it is plausible to have any sequence of challenges or any type of sublevel as a setback penalty, with its own specific state chart. Thus, a general setback penalty can be described with the state chart presented in Figure 4.6.

After the previous examples, we can identify two main variables that contribute to difficulty in one level:

- *w(t)*: The probability of keep trying the upcoming challenges. This value can be obtained with a predictor of the players' emotional states during the level and represent their resilience.
- *pj(s):* The probability of achieving success in the jump defined by its size *s*. This concept can be expanded to a more universal perspective considering that it represents the probability of success in a general challenge, where *s* denotes the set of features of that challenge with

influence over the probability of success. This type of analysis is the content of the next subsection.

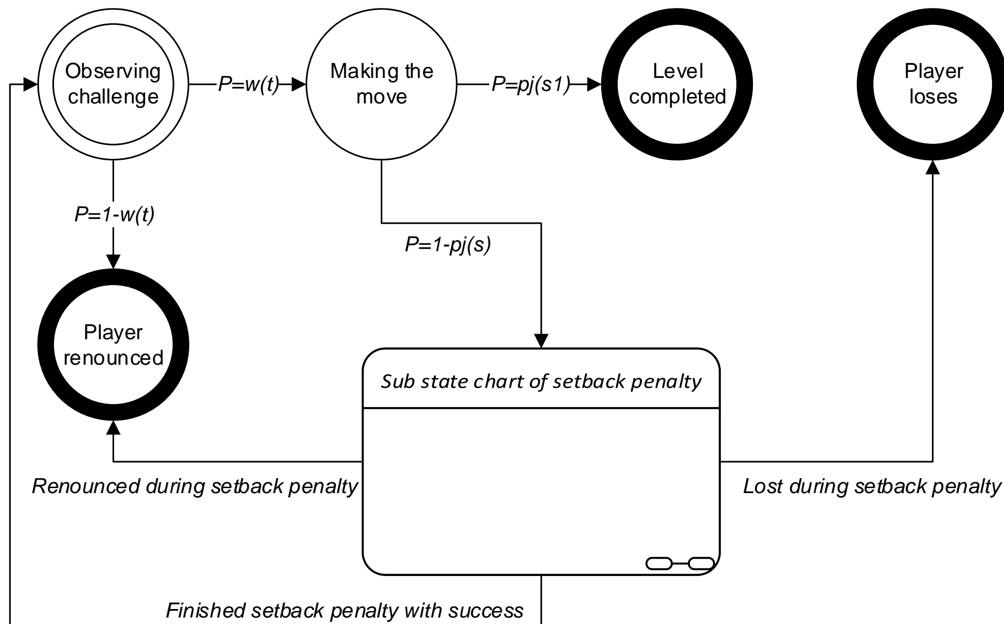


Figure 4.6 – Example platform videogame situations. State chart for situation c, generalised.

In order to estimate the difficulty in a whole level, it has to be decomposed into parts, which are then mapped into the previous situations. That results in a global flow chart representing possible state transitions. As those transition are associated with probabilities, the whole level can be represented as a Markov Chain (Markov, 1971; Norris, 1998). The definition of a Markov Chain implies a process that assumes a finite or countable set of states and in which only the current state has influence in the decision of what will be the next state.

4.4.2. Independent Challenges

Identifying probabilities for segment transitions in a platform level based only on the level content and the avatar's characteristics is a non-trivial task. Characters' motions are different for each game and distinct games provide their own enemies and traps with their own possible models. Transposing physical constraints to a difficulty measurer is also a complex task. It requires changing the original source code, if it is available, or reverse engineer the game mechanics to establish an equivalent model. For these reasons, we propose to observe general characteristics in those models and analyse them regarding space and time.

Spatial Challenges

The most natural case of a spatial challenge is jumping from one platform to another. In a jump, looking from a high level of abstraction, the player is trying to jump at least from one limit point to another one. These points represent the edges of the origin and destiny platforms. In other words, these two points define the minimal required jump that the character has to perform.

Based on this idea, Sorenson and Pasquier (Sorenson & Pasquier, 2010a) proposed that a challenge *c* can be estimated over time *t* with the following equation:

$$c(t) = d(p_1, p_2) - \left(fp(p_1) + fp(p_2)\right) + 2.fp_{max}$$

The value d represents the Manhattan distance between the points that define the gap and the values fp denote the notion of footprint, represented as the length of the platform bounded to the maximum possible jump that the avatar can perform. The metric is depicted in Figure 4.7.

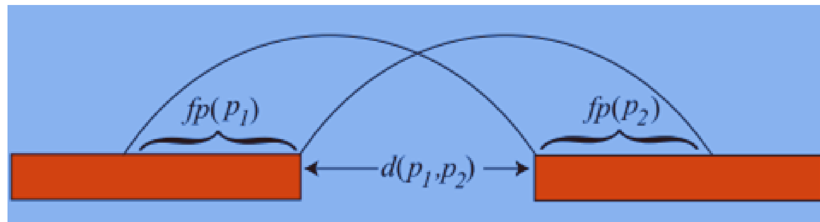


Figure 4.7 – Principle of margin of error based on gap footprint (Sorenson & Pasquier, 2010a).

Even though this metric provides a function that is related to the players' success and failure, some limitations can be pointed. First, the difficulty of a jump is calculated symmetrically, which does not include the fact that, for platforms at an equal horizontal distance, it is easier to jump to a lower platform than to a higher one. Furthermore, the values do not represent any specific unit, making them hard to analyse and compare. Difficulty is normally seen as a probability of failure, which is complex to infer from the established function.

With similar principles, we have proposed an alternative method (Mourato & Próspero dos Santos, 2010) to supress the main flaws that were referred. It consists in launching a projectile from the first point, P_0, and measuring the possible margin of error for its trajectory relatively to the second point, P_1, calculating an additional point, P_2, at the intersection of the projectile with the destination platform. This trajectory represents the maximum accomplishable jump that the avatar can perform. Recalling basic physics, the following set of equations define the trajectory of a projectile, in order of time:

- $x(t) = x_0 + v_{0x}.t$
- $y(t) = y_0 + v_{0y}.t - \frac{1}{2}a.t^2$

The projectile starts its trajectory at the end of the origin platform and ends it at the intersection with the destiny platform. A margin is then established, representing how shorter that jump could be without compromising its success. Once every jump is relative to the platform from which the user is jumping, we can consider the origin point (P_0) as the reference, to simplify the equations, meaning that both x_0 and y_0 have a value of zero. Besides, we will not consider a throwing angle, so the initial speed can be defined as a constant (K_i) based on the character's velocity, configurable for different games according to that same character. Finally, these calculations are space oriented and the jump duration is not used, so the calculation can be reduced to one expression, a quadratic equation intersecting the origin, defined by:

$$f(x) = K_i.x - \frac{1}{2}a.x^2$$

By intersecting the projectile with two lines that are parallel to the cartesian axis and that also intersect the destiny point, we can estimate the possible deviation to the trajectory that is still a valid jump. Henceforth, this deviation will be referred as the *margin of error* or *error margin*. In fact, we are identifying two values: the height of the player when he/she horizontally reaches the platform in relation to the origin (Δ_y) and the horizontal amplitude of the jump (Δ_x). Therefore, our error margin has values in x and y axis (m_x and m_y) defined by $\Delta_x - x_1$ and $\Delta_y - y_1$, respectively, with $P_i = (x_i, y_i)$. This concept is represented graphically in Figure 4.8.

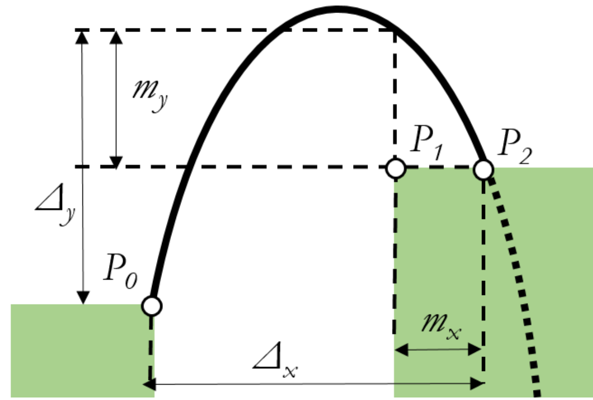


Figure 4.8 – Graphical representation of the concept of error margin.

Obtaining m_y requires knowing Δ_y, which consists in calculating $f(x_1)$. A negative value means that the platform in unreachable with the reference jump. In this case, the system has to identify whether the platform is in fact unreachable and thus the error margin is immediately zero without any other calculations, or the jump has to be attempted differently, moving the origin point to the left (we will get back to this specific situation). For positive values, we measure the horizontal tolerance by calculating the value of x for which $f(x)$ equals y_1, which consists in solving a second-degree equation and selecting the appropriate root. To normalise the results, we consider the error margins as relative values to the full distances measured along the axis. In the end, we multiply both the normalised error margins to define the final error margin (M) for the obstacle, a value between zero and one.

The calculation of the error margin is possible for every placement of P_1 in relation to P_0. In fact, for a certain game, this method divides the domain of the possible gap configurations into those that are accomplishable and those that are impossible to overcome. Additionally, as we have referred in the previous paragraph, a third situation may occur, requiring trajectory corrections. Such case occurs for jumps wherein the character does not have enough space to reach the maximum height. Figure 4.9 summarises the possible situations regarding the location of P_1.

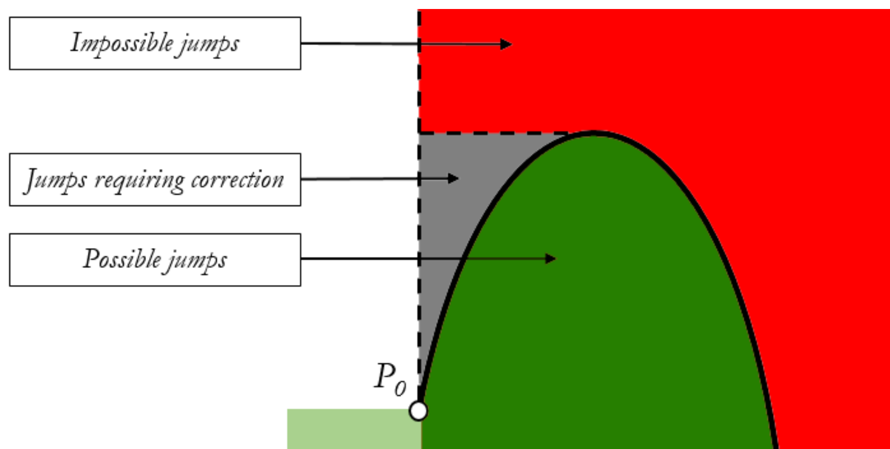


Figure 4.9 – Jump classification regarding the placement of the destination point, P_1, in relation to the origin point, P_0.

The trajectory correction consists in performing the jump earlier, providing enough space for the character to achieve the maximum possible height. For such process, one has to calculate the minimum and maximum shift in x that can be applied to the trajectory to make it possible. We refer to these corrections as c_{min} and c_{max}, respectively, as depicted in Figure 4.10.

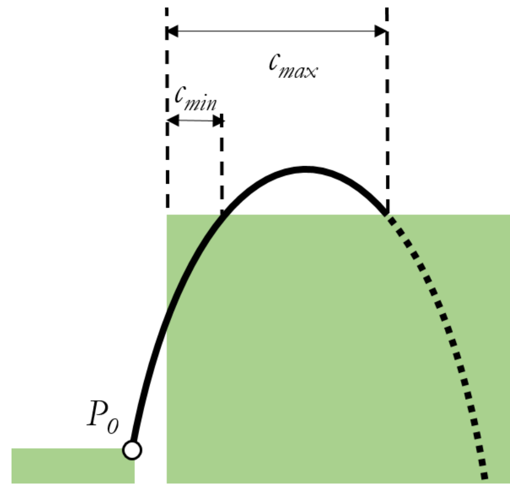


Figure 4.10 – Maximum and minimum correction values to the trajectory to make a problematic trajectory possible.

Even though these values are useful to define the boundaries of the correction, their direct usage is not possible. If we apply a translation to P_0 to the left, equivalent to c_{min}, the trajectory is theoretically possible but its respective value of m_y is 0, making it practically impossible. Similarly, if we translate P_0 to the left with a value of c_{max}, the trajectory will have a value of 0 for m_x. Our proposal is that, in this type of situation, one should consider a translation with the average of the two values. With this correction, every placement of P_1 results in a final value of error margin, established between zero and one.

Empirically, one can perceive that difficulty relates to the error margin in a non-linear fashion. For higher error margins, it is natural to obtain values of success around 100% and the real effect of difficulty is noticeable especially for lower error margins, where failure rises fast. The effect is observable by experiment, as depicted in Figure 4.11, where we relate the error margin of different jumps with the measured percentage of failure in those jumps. This data was obtained as part of a larger validation study that will be presented with more detail in chapter 8.

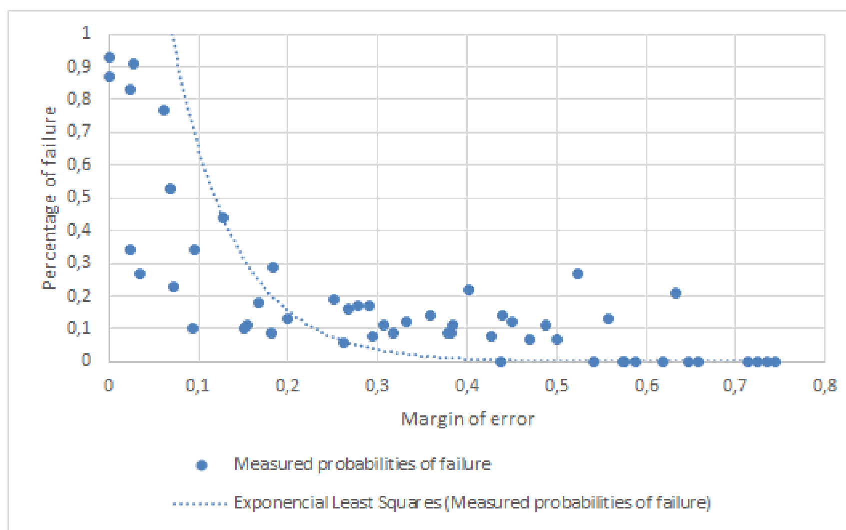


Figure 4.11 – Experimental measurement of percentage of failure in relation to the identified error margin of a challenge.

To reflect this principle, we represent the final difficulty value for each obstacle based on an exponential function with the following equation for difficulty in the obstacle of index i:

$$D(i) = 1 - M_i^{K_d}$$

The constant K_d represents the mapping of the linear error margin and the expected exponential distribution, which can be configured to the profile of a certain player, with a value that ranges from 0 to 1. Typically, lower values represent the profile of an expert player and higher values represent less skilled players. In Figure 4.12, it is possible to observe the differences between different values of K_d.

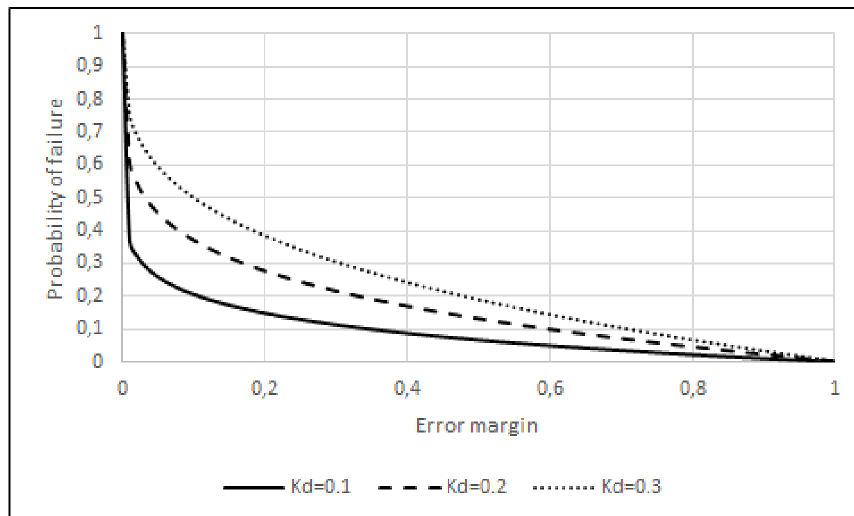


Figure 4.12 – Probabilities of failure in relation to the error margin considering different exponential values (*K_d*).

A first experiment to validate the proposed approach was implemented with the game *Super Mario Bros.* We selected a set of levels that share similar objects and obstacles to avoid having particular aspects biasing the conclusions, such as the existence of *bosses*, portals or different physical conditions. For instance, levels that are played under water were not considered. The levels were manually segmented, based on the visual interpretation of bitmap representations and world parameters were obtained by test experimentation. The concept of will to continue was mapped into a maximum number of retries for each challenge (*T*). In Table 4.1, we present the obtained difficulty predictions for different values considering alternative values of *T*.

Level	T=1	T=2	T=3	T=∞
World 1, Level 1	10.8%	32.4%	40.8%	44.6%
World 1, Level 2	3.5%	10.7%	13.7%	15.0 %
World 1, Level 3	0.5%	0.8%	0.9%	1%
World 2, Level 1	0.22%	3.15%	5.8%	7.4%
World 3, Level 1	2.4%	15.9%	21.6%	24.0%
World 3, Level 2	3.6%	13.2%	15.3%	15.7%
World 3, Level 3	0.40%	0.51%	0.52%	0.52%

Table 4.1 – Estimated probabilities of success for different levels of the game *Super Mario Bros.,* considering alternative numbers of retries.

It is possible to observe that, in general, difficulty values are smaller for early levels and higher for later levels, which confirm one basic notion of game design: levels are gradually more difficult along the game.

Another experiment was implemented with the game *Little Big Planet*, to perform a primal test with the difficulty predictions based on projectile physics against the effective probabilities of success. This game allows the users to create their levels, thus this feature was used as a research tool. One particular level was created with multiple independent jumps with different gap characteristics, which were measured to extract an estimated probability of success. A set of users played the level, allowing the extraction of the effective probability of success. It was possible to observe similarities between measured and predicted difficulty values, as presented in Table 4.2. In this case, our best results were achieved using K_d with a value of 0.05. A more complete evaluation of this estimator will be presented in chapter 8.

Margin of error	**Measured *P(s)***	**Pred. *P(s)*, *Kd = 0.2***	**Pred. *P(s)*, *Kd = 0.1***	**Pred. *P(s)*, *Kd = 0.05***	**Pred. *P(s)*, *Kd = 0.01***
4,7%	40,40%	54,3%	73,7%	85,8%	97,0%
8,7%	89,30%	61,3%	78,3%	88,5%	97,6%
15,4%	96,20%	68,7%	82,9%	91,1%	98,1%
23,8%	91,70%	75,0%	86,6%	93,1%	98,6%
36,7%	98,00%	81,8%	90,5%	95,1%	99,0%
43,7%	97,10%	84,7%	92,1%	95,9%	99,2%
49,5%	98,00%	86,9%	93,2%	96,5%	99,3%
62,9%	98,00%	91,1%	95,5%	97,7%	99,5%
67,8%	98,00%	92,5%	96,2%	98,1%	99,6%
81,4%	96,20%	96,0%	98,0%	99,0%	99,8%
81,4%	98,00%	96,0%	98,0%	99,0%	99,8%
84,3%	100,00%	96,6%	98,3%	99,1%	99,8%

Table 4.2 – Measured and predicted percentages of success in a test level of the game *Little Big Planet.*

While this approach adds anisotropy to the concept, depicting the fact that it is easier to jump to a lower platform than to a higher one, it is still based on a displacement of theoretically perfect jumps. Therefore, we have researched possible improvements for such measurements (Mourato et al., 2014). In the specific case of jumping through a gap, we raised the hypothesis that the players have a tendency to perform their jumps slightly before the end of the origin platform. Still, some jumps occur before this point and some of them after it. Considering d as the distance to the end of the platform, it is expected to find a distribution centred in $\overline{X_d}$, as represented in Figure 4.13. The probabilities of success and failure can be obtained based on this distribution. Therefore, one needs to define the value $\overline{X_d}$ and measure, for each gap, the maximum value of d in which the gap is accomplishable. This value (d_{max}) can be extracted from the trajectory curve. If the jump is performed at distances from the edge that are greater than d_{max}, then the player will not reach the second platform, and if the

jump is performed at negative values, the player will fall without jumping. Figure 4.14 represents the areas within the distribution in which the jump is successful. In our previous approach using error margins, we have referred a specific situation in which the jumps that are too close to the edge of the platform are unsuccessful. They happen when the gap is small and the character needs additional space to reach the required jump height. We have explained how to calculate c_{min}, the minimum distance that the player has to leave in relation to the edge of the platform in order to succeed in the jump. With this distribution-based approach, when such type of situation occurs, the area within the distribution in which the jump is successful should be obtained from c_{min} to d_{max}.

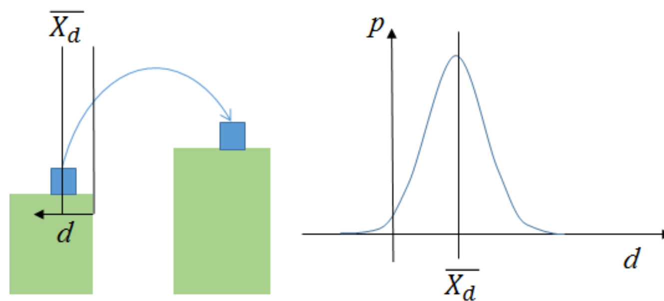


Figure 4.13 – Distribution on the jump origin point in a gap challenge.

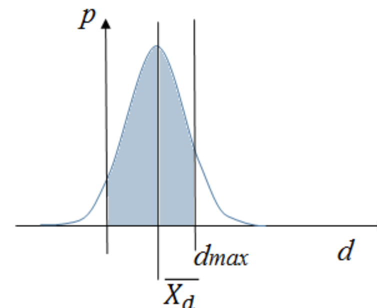


Figure 4.14 – Boundaries to obtain the probability of the successful jumps.

In order to verify the viability of this approach, we have presented a study with a prototype game that was played by 40 users. The majority of the distributions for each individual player, regarding the distance *d*, was shown normal after a Kolmogorov-Smirnoff test (K-S test), presenting *p*-values over 0.05. In addition, the overall distribution for that variable was also analysed but, in that case, normality could not be assumed (*p*-value under 0.05). This might be due to the fact that different groups of players have indeed different distributions that cannot be analysed globally. An extended version of this study will be presented in chapter 8, where we will also compare this approach for estimating difficulty with the previous metrics based on error margins.

The same approach has also been used in this study to analyse difficulty in other types of challenges besides jumps over gaps, as it will be explained next.

Time-Based Challenges

Some of the challenges that one can find in a videogame are related to time instead of space. In these challenges, the user has to perform a certain action or a set of predefined movements within a certain time window. For instance, choppers in the game *Prince of Persia* are an example of such situation. They are represented by two blades closing within a certain period, killing the character if he is in contact with those blades while they are closing.

To analyse this situation, we will consider T as the period of the challenging entity, t_{f1} the harmful time window and t_{f2} the time required to overcome that entity. In the referred time span, failure occurs if the player crosses the chopper during its harmful time window (t_{f1}) or if the movement is performed too late within the harmless interval (during the interval with a duration of t_{f2} that occurs before t_{f1}). Once again, our hypothesis is that the players try to perform the transition during the harmless interval, defined as t_s, following a similar distribution model as presented in Figure 4.15.

That distribution can be analysed within the period that defines the challenging entity in order to estimate the amount of actions that are successful and failed. Considering an approximation to a known distribution, it is possible to calculate the definite integrals for each region and determine an estimation for the probability of success.

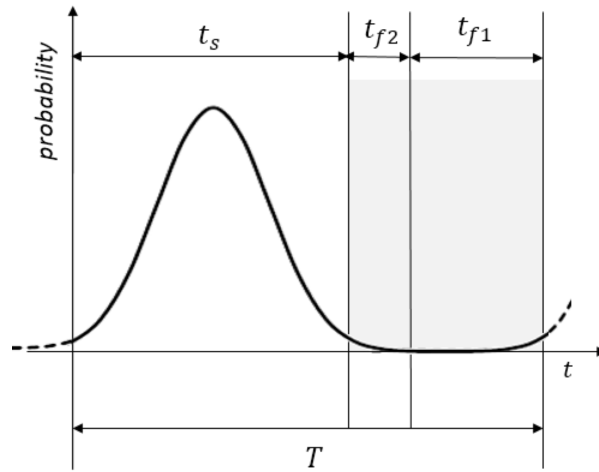


Figure 4.15 – Distribution over time for trials in a time-based challenge.

The performed tests resulted in the same conclusions as in the spatial challenges. Most of the individual distributions can be considered normal, as a result of a K-S test (individual distributions have a *p*-value over 0.05), but that conclusion is not valid for the overall distribution.

Dynamic Entities (Moving Platforms)

While in the previous examples the challenge was directly related to space or time, sometimes both features are interconnected and mutual dependent. An example of such situation is given by a dynamic/moving platform. In this case, the player has to perform a jump from one static platform to a dynamic one.

We will analyse this situation within a time interval T corresponding to one cycle of the moving platform, centred in the instant at which the distance $d(t)$ is minimum in relation to the origin platform. For each distance value, it is possible to establish a probability of success $P_s(d)$ using the presented method for regular gaps. Considering that the player tries to time his/her jump at the instant that corresponds to the minimum distance, we set the hypothesis of having a jump distribution $Pj(t)$ also centred in the interval T. The probability of success P can be obtained by multiplying $P_s(d)$ by $Pj(t)$, which represents the distribution of the successful cases. Its definite integral represents the overall probability of success. The previous functions are graphically represented in Figure 4.16.

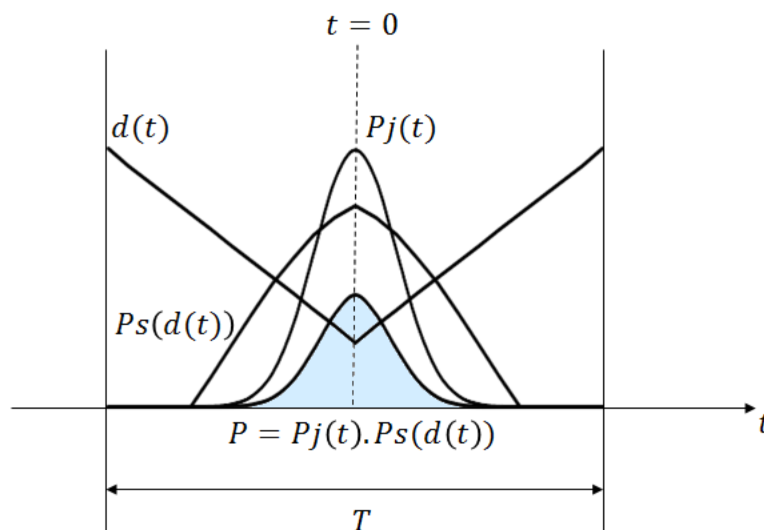


Figure 4.16 – Distance and probabilities as a function of time for a moving platform challenge.

In the same manner as in regular jumps, we have started by measuring the distance from the avatar to the origin platform edge for each of the jump attempts. Once again, it was possible to observe, with a K-S test, that the overall distribution cannot be considered normal (*p*-value under 0.05), though the majority of the individual distributions for each player are considered normal (*p*-values over 0.05). These results suggest again that different groups of players have different distributions that cannot be analysed globally.

Regarding the time related component of the challenge, we have also registered at which distance the moving platform was when the player attempted the jump. In this case, it was possible to verify the intuitive notion that the players aim to perform the jump with the minimum gap distance. The K-S test for normality was applied resulting in a *p*-value of 0.2, confirming our hypothesis. In addition, it is important to refer that the obtained mean value for the distribution was -0.4, which represents an additional motion compensation. This means that, in general, the player performs the jump slightly before the platform reaches the nearest position, aiming to land at the minimum distance.

Spatiotemporal Challenges

Lastly, the direct combination of challenges was also analysed. Our reference situation consisted in combining into one single challenge the principles of the spatial and the time-based challenges, as a jump through a gap that has to be performed in time. The main hypothesis we wanted to test was if in this combined situations it was plausible to use the two independent probabilities of the represented situations, thus making possible the extraction of a combined probability of success. The performed tests did not reveal any dependencies or correlations between the variables, which goes towards our hypothesis. We have also compared difficulty for different parameters in such challenges for the combined situation and the individual cases. The differences of success regarding gap size were analysed with and without the presence of a time-based challenge placed at the gap and, in the same manner, the differences of success regarding the time window were analysed with and without the presence of a gap. Once again, we could not identify any trend, and the success probabilities in the referred challenges appear to be dependent only of its relative features.

4.5. Concluding Remarks

We have explored human factors as a way to improve some of the capabilities in automatic level generation. As stated, the representation of a scenario that is playable as a game level has an implicit difficulty perception by the users that reflects directly in their performance. As part of this work, we have presented the main following principles about difficulty:

- The level structure has impact on the difficulty perception as it defines the composition of the presented challenges.
- Individual challenges can be analysed within the global level structure as long as it is possible to establish a function to define general probabilities of success of overcoming those challenges.
- Individual challenges can be estimated either regarding spatial or temporal features.
- Predicted difficulty values for a level can be adapted to a certain profile by establishing a coefficient factor based on the users' skills, which has repercussions over individual challenge metrics.

A common way to analyse the impact of difficulty in performance consists in identifying probabilities of success and failure for the existing challenges within a level. For that matter, we have proposed a set of metrics to estimate such probabilities. Those estimations lay on broad abstract representations of the challenges in order to make them generic and thus applicable in different games. These metrics

can be used in automatic level generation processes, as we will observe in the remaining of this document. In chapter 5, we will present the architecture of a PCG system in which we include difficulty analysis, as well as other gameplay data, as part of that system. Moreover, in chapter 7, we will dig into the detail of level generation algorithms, in which we will see how difficulty can be used directly in those algorithms in order to generate personalised levels. Finally, in some of our initial tests, we have verified that the proposed metrics provide a valid estimation for difficulty in a level, behaving similarly to the effective values that are observable in players' statistics. Later, in chapter 8, we will look again to these metrics and present with more detail the obtained results of such estimators using data retrieved from more extensive gameplay sessions.

To conclude, it is important to refer some interesting points for further research, which have not been considered in the proposed metrics. Occasionally, the composition of individual challenges creates overlapping zones, shared by more than one challenge. For instance, in a level that has two consecutive gaps close to each other, the player can jump across the first gap and fall directly in the second one. Additional research regarding these cases is needed, in order to understand how such situations are different from the independent composition and how to it should be tackled. Typically, in this type of situation, the player will not perform the maximum possible jump and will stop the trajectory after reaching the destination platform (for games in which it is possible). Therefore, the analysis of different types of trajectories and deviations from the perfect theoretical jumps should be considered in further research.

5. Architecture for a PCG System

"Often, the ontological elements are derived from common game terminology and are then refined by both abstracting more general concepts and by identifying more precise or specific concepts."

(J. Zagal & Bruckman, 2008)

5.1. Introduction

In the previous chapters, we have established the background to understand the most relevant aspects to consider in automatic level generation for platform videogames. We have perceived the main challenges to face in this topic and the most important lines of work to follow in the future. With that in mind, we have defined a global architecture that embraces the presented features, which we will describe in section 5.2. Additionally, in section 5.3, we will detail how level content can be defined and structured, and how to make it suitable to be used within the proposed architecture. In section 5.4, we will present an application that was developed to work within such system, either as a level editor or as a testing environment for generation algorithms. Moreover, in section 5.5, we will also present the first concretisation of the proposed architecture into a game prototype. Lastly, section 5.6 closes this chapter, where we will present our concluding remarks.

5.2. Architecture

We have designed and developed an integrated system that comprises the most relevant aspects that an automatic generation system should contain. It was envisioned to comprise the modules presented next:

- The **Level Generation** module contains different algorithms and techniques to produce levels as its output. A common language to define the game content and represent the levels is essential to allow communication between modules. This module also contains several tools to support level analysis, which can serve different algorithms, such as difficulty predictors and path finders, among others.
- The **Game Engine** module contains a videogame implementation that allows, in the first place, to view the output with details in a graphical environment and, furthermore, to effectively play those levels. This module reinforces the need of a common language to bridge it with the level generation part.
- The **Statistics and Profiling** module plays two different roles. First, data should be gathered from the game implementation to extract relevant information about gameplay. Secondly, this information can be used beyond statistical studies. It is also suitable to feed the generation module with global or specific data to tune the output. For instance, the generation process may take into account a set of parameters regarding the user's preferences, provided by this module, leading it to a more personalised content.
- **System Manager and User Interface** works as an additional module that provides a front-end to the users (players), functioning as a game start menu.

The process of playing a level involves communications between different modules, as all of them have a part in the process. This procedure involves the steps depicted in Figure 5.1 and presented as follows:

- Depending on the choices of the creators/designers, the user, via the main interface, asks for a level with more or less control over the process. For instance, a stricter version of the system can have simply a *play random level* button but a more unrestricted version can allow the player to define a set of preference parameters, such as the desired difficulty.
- The UI forwards the request referred in the previous step to the Level Generation module, possibly having an additional set of parameters regarding that request and depending on the referred control that the user might have.
- The Level Generation module receives the request that was forwarded by the UI and can query the Statistics and Profiling module to gather information about the user such as his/her skills and preferences. A set of algorithms is selected and the level is generated. As the level is obtained, it is sent back to the UI module.
- As the UI receives the level (description), it is forwarded to the Game Engine module to render it and to provide a gaming interface to the player.
- The Game Engine module receives the level structure, builds the level in the rendering engine and returns the virtual environment to the user, via UI. As the user plays the game, data is possibly gathered and sent back as feedback to the Statistics and Profiling module.

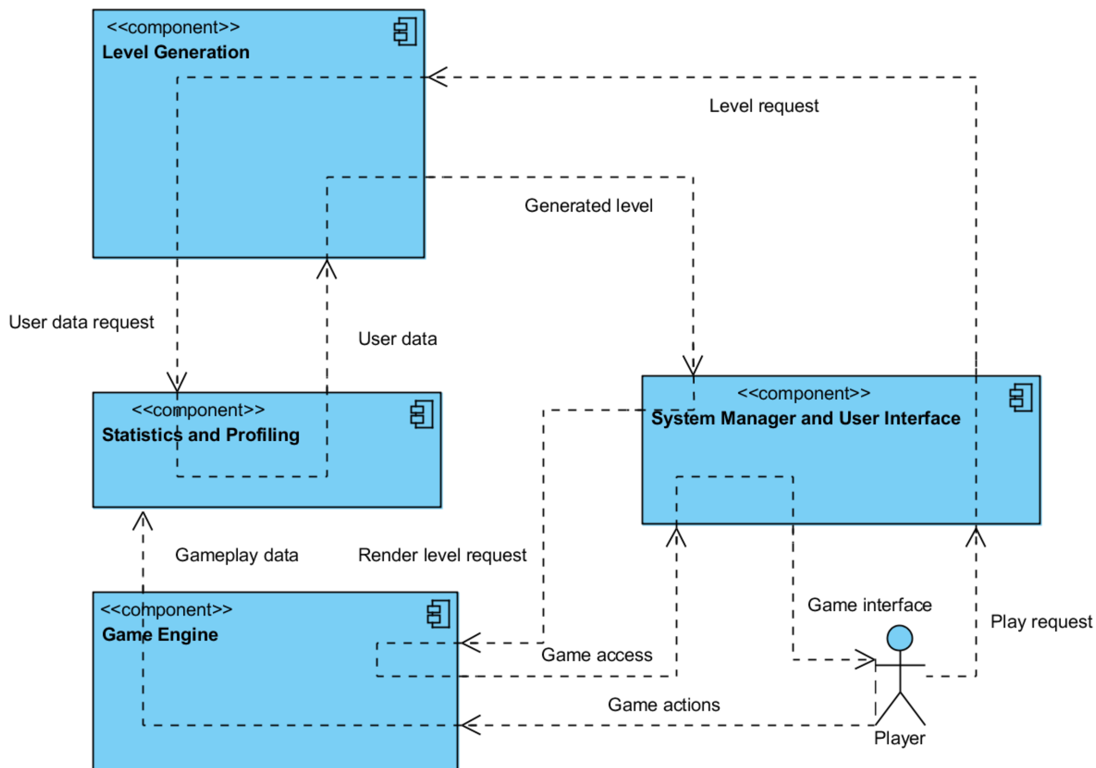


Figure 5.1 – Overview of the system architecture, composed by the following modules: Level Generation, Statistics and Profiling, Game Engine and System Manager and User Interface.

5.3. Framework for Platform Games

The presented architecture is generic and independent of the game. In order to promote that independency, it is also important that the game content of different games can be described within one

single representation. Additionally, different modules need game levels, so it is important to have a common level representation mechanism for their integration. For this purpose, we have defined a scheme for content description and level representation, flexible for future game implementations and that allows mapping levels from different existing games. The existence of this framework to generically represent game content and levels encourages the standardisation of the techniques that we will analyse and propose in the remaining of this document, providing a common ground for comparisons among distinct titles. Besides, the representation scheme should be suitable to be used by PCG algorithms. For instance, a pixel-by-pixel description of a level scenario would be valid but would not contain any useful semantic information that could be processed in the automation of level creation. In addition, it is interesting to have a representation that is also user-editable, meaning that is suitable to apply in a level editor for users to create their own levels.

5.3.1. Existing Approaches

As previously referred in subsection 3.3.3, the study of platform videogames was brought to academic research by Compton and Mateas (Compton & Mateas, 2006). We observed that the authors analysed the overall level structures and the main strategies to organise those structures.

Later, Smith *et al.* (Smith et al., 2008) presented a more extensive analysis about the existing components of a level for a platform game. The authors defined a conceptual model with an associative and hierarchical description among different entities, as presented in Figure 5.2. The main principles follow some of the thoughts proposed in the *Game Ontology Project* (GOP) (J. Zagal & Bruckman, 2008; J. P. Zagal, Mateas, Fernández-Vara, Hochhalter, & Lichti, 2005), where a more generic model is presented. The GOP is an effort to create a single framework to describe, analyse and study games, based on a hierarchy of concepts. In a global view, levels are composed by cells that are possibly connected between them. Naturally, each cell has at least one connection, designated as portal. The authors defined cells as sections of linear gameplay. Cells are composed by one or more rhythm groups, combining the following entities: platforms, obstacles, movement aids, collectible items and triggers. One particular note goes to the fact that gaps between platforms are also represented as obstacles.

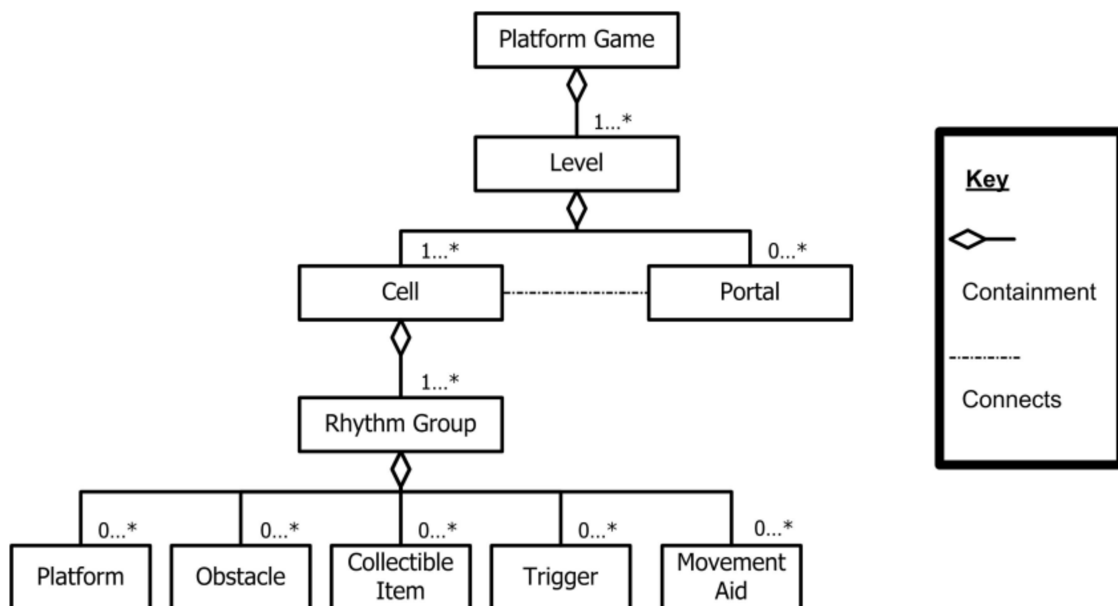


Figure 5.2 – Model of level framework proposed by Smith *et al.* (Smith et al., 2008).

Besides the presented structural aspects, the authors also used this framework as a basis for automatic generation processes, which we will cover in detail later in chapter 7, where possible techniques to automatically generate content for platform videogames will be studied. The defined hierarchy represents an interesting approach to the problem and largely covers the concepts inside a platform videogame. However, it is based on a non-systematic decomposition of levels with broad concepts. For instance, considering common level representations in existing levels, the mapping to the previous framework requires manual editing of that content, as games do not explicitly refer rhythm groups and connections between them. Moreover, level editors do not commonly describe gaps as obstacles. The framework that we will present in the next subsection intends to be more direct on the effective representation of entities in the scenario and their respective analysis.

5.3.2. Basic Organisation

Spatial Approach

We propose to displace the content spatially within a grid, with a representation that is based on a discreet spatial enumeration as a two-dimensional matrix. Different independent overlapping layers are allowed, though the majority of the further examples only use a single layer for the sake of clarity. Even though this representation restricts the domain into a specific resolution, the fact is that a large number of videogames contain structures based on a grid representation. This is still a popular approach to represent content in *platformers*, as well as in other genres with emphasis on two-dimensional structures. Likewise, some utilities that already exist with generic approaches for level construction in two-dimensional environments also lay on this type of representation, such as the *Scrolling Game Development Kit*[1] and *Tiled*[2]. We envisioned our framework to be used within a level editor in a similar manner as these applications, allowing a generic definition of content for multiple purposes. Furthermore, the inclusion of semantic notions that are only present in *platformers*, in a generic way yet including implicitly some semantic principles about the content, promotes the usage of PCG algorithms.

Content Ontology

Each cell in the spatial displacement represents a value from a set of possible cell values that we will refer henceforth as blocks. We have defined that blocks are arranged on a hierarchy. A block can be represented as a child of another block, meaning that it is, somehow, a subtype of that block. This is not a mandatory feature, as a flat hierarchy is valid. However, it can be useful for user level creation and as a base for some procedural generation techniques that can be used on a grid representation, as we will see along this subsection. Additionally, cells have a set of properties with their respective values. These properties are defined initially for each block.

As an example for the definition of an ontology for a game, we will consider the screenshot of the videogame *Super Mario Bros.* that is shown in Figure 5.3. A possible block hierarchy to map the elements in that scene is shown in Figure 5.4. To simplify the representation, the clouds and the bushes were not considered and should be interpreted as an independent second layer that only has aesthetic purposes. The remaining ten different types of blocks were divided into four main groups. Even though this grouping might be arbitrary, it shall make some sense in a certain perspective. In this case, we considered the division into solid blocks, enemies, pipe parts and empty (spaces) as a typical grouping that one could find in a level editor or a similar application.

[1] http://sgdk2.sourceforge.net/
[2] http://www.mapeditor.org/

Figure 5.3 – Screenshot of the videogame *Super Mario Bros.*

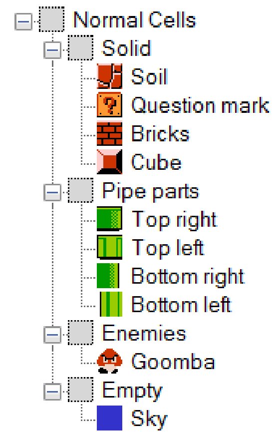


Figure 5.4 – Example hierarchy for a simplified version of the videogame *Super Mario Bros.*

5.3.3. Spatial Groups

Despite the representation based on cells, it is possible to define spatial groupings to explicitly state larger objects composed by smaller parts (blocks). For instance, in the game *Prince of Persia*, level exit doors are represented with two cells. As another example, in the game *Infinite Mario Bros.* regular pipes occupy four cells. We refer these larger specific entities as *spatial groups*. This concept is common in *platformers*, as these examples show and, in fact, it is also used in other genres with grid representation schemes. For instance, in *Sim City,* a strategy city building game, this type of situation also occurs. Levels are top-viewed maps where each cell represents the terrain type, such as grass, water, sand, trees, roads and several other varieties. However, larger entities, such as schools and hospitals, occupy more than one cell.

Moreover, the pipe example has also another characteristic. Pipes are always two cells wide but they can have two or any greater value as height. This type of object is entitled as *resizable group* and refers to game entities that are larger than the basic cell space, as a spatial group, and that can be stretched at least along one dimension. Resizable groups are defined with maximum and minimum height and width values. Graphically, they are represented with 9 independent blocks obtained from the initial hierarchy, representing a graphical block for the possible positions within a 3 by 3 array. This concept is similar to the 9-slice scaling feature of common vector graphics editors such as *Adobe Illustrator*. In Figure 5.5, we present an example of a resizable group and some of its possible concretisations with certain sizes.

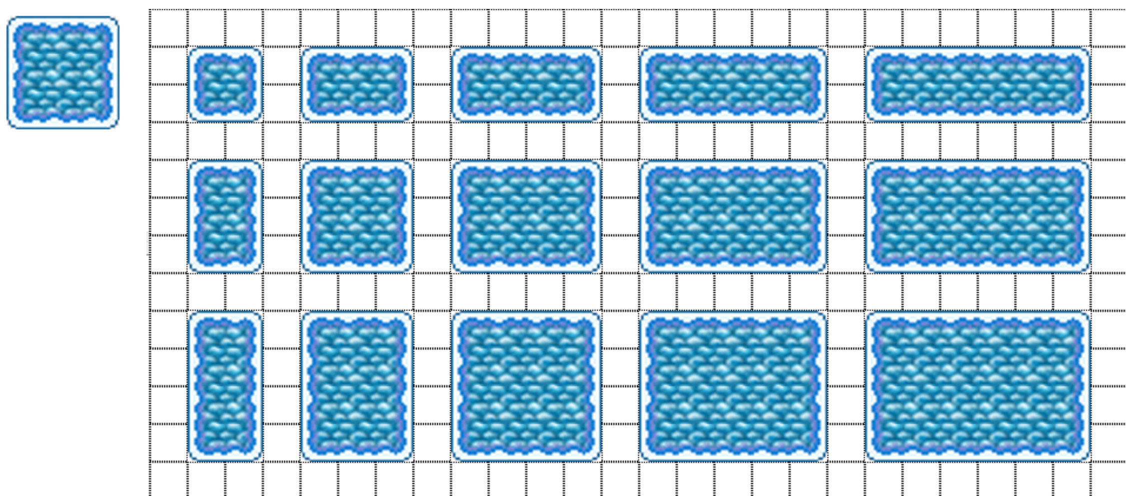


Figure 5.5 – Example of a resizable group and different possible concretisations.

Groups, either static or resizable, are typically optional for the purposes of gameplay, but, as a creation tool, they can be used to enhance design interfaces providing a direct editing mechanism for that type of entities and can also serve procedural algorithms once they provide additional semantic information about the game content and the existing structures. For instance, a generator that knows the block composition of a pipe will not create invalid representations with such blocks.

5.3.4. Categories

The main block hierarchy, explained in subsection 5.3.2, is already a grouping mechanism, which defines the main organisation structure of the existing content. This consists of an exclusive categorisation that does not allow multiple interpretations for one single block. However, depending on the context, a block can indeed have multiple interpretations. For this reason, an additional categorisation mechanism was desired for the framework to allow the definition of arbitrary logical concepts regardless the main grouping approach.

In order to understand the presented issue, an example is provided. We can consider a simple *platformer* based on the game *Rick Dangerous*, including only three types of blocks:

- Solid blocks, representing walls and solid floor for the avatar.
- Empty blocks, representing tunnels where the avatar can move.
- One-way platforms that, as we previously explained in subsection 3.2.1, allow the avatar not only to step on but also to jump through them. In practice, they are a platform if the avatar is already on top of them, but are otherwise traversable.

Each of the existing blocks is different from the others, so the initial grouping mechanism does not contribute in particular for their organisation. Therefore, one can consider a flat representation in the hierarchy. However, in order to improve the knowledge of artificial agents in the game or automatic level generation algorithms, some additional block categorisation might be useful. For instance, in order to calculate possible paths within the levels, it might be useful to enumerate which of the blocks are traversable, in which blocks the avatar can stand on, and which blocks prevent passage. Such description is not possible only with a block hierarchy because some blocks play different roles.

To supress this issue, we have defined a mechanism where it is possible to establish *categories*, based on the definition of logic rules with boolean operators to define which blocks fit a category. Those allow the enumerations of traversable and sustainer blocks, which are obtained with the following expressions:

- *Traversable* = *Empty* ∨ *Platform* or, alternatively, *Traversable* = ~*Solid*.
- *Sustainer* = *Solid* ∨ *Platform* or, alternatively, *Sustainer* = ~*Empty*.

Naturally, with a larger set of existing blocks with different features, the usage of such grouping mechanism has higher relevance regarding the automation of level analysis. In the next subsection, we will show a situation where this additional categorisation mechanism can be put into practice, using those enumerations to identify character movements along the game levels.

5.3.5. Movements Aids

To conclude the description of the proposed framework, the model can be detailed by setting instructions to relate cell content to possible avatar actions. This can be useful in further steps to extract a sketch map of the main available movements within a level to understand its basic structure. Every type of movement can be described by rules that are composed by a cell pattern and a scheme that represents the avatar movement within the cells denoted in that pattern. Cell patterns are defined as

a two-dimensional matrix of categories and the movements are represented as possible transitions between cells, structured as a graph. Recalling our example case presented previously in subsection 5.3.2, regarding *Infinite Mario Bros.*, we can define a category named *Platform* to represent blocks with physical mass, and a category *Space* to represent blocks without physical mass, using the following expressions:

- *Platform* = *Solid* ∨ *Pipe Parts*
- *Space* = *Enemies* ∨ *Empty*

Accordingly, one can define the basic horizontal avatar movement with the rule presented in Figure 5.6. Briefly, this rule expresses that the avatar can move from one cell, containing an empty space or an enemy, to another one, horizontally adjacent that also contains an empty space or an enemy. It also requires the movement to be performed with solid blocks or pipe parts below.

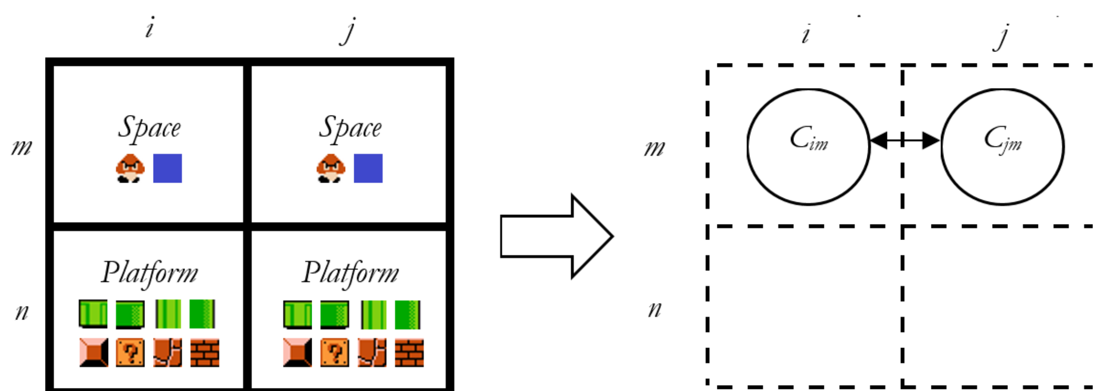


Figure 5.6 – Example of a rule composed by the pattern (on the left) and the corresponding graph (on the right).

In chapter 6, we will explore the potential of this type of rules, as we will observe with more detail how they can be used to extract a global movement representation for a certain level, describing possible avatar movements. Furthermore, in chapter 7, we will show how that information about the level structures improves automatic level generation algorithms.

5.4. Level Editor

In order to test the described framework, we have developed a level editor using the proposed approach. In Figure 5.7, we provide a screenshot of the referred application while editing a level for the game *Prince of Persia*. On the left part of the screen, it is possible to observe a practical usage of the aforementioned concepts that are included in the framework, namely the main representation of the blocks within their hierarchy, the existence of spatial groups using blocks and a block browsing mechanism using the set of categories.

This application works as a generic level editor and it is possible, in its configurations, to define the content of the game within the previous framework. The editing process typically goes through the following steps:

- Setting the images that compose the bitmap representations within the editor, by using the screen presented in Figure 5.8, where it is also possible to define the visual size of each cell.

- Defining the block hierarchy that describes the core elements of the game. Such process is done with the screen presented in Figure 5.9 and follows the principles that we have explained in subsection 5.3.2. Besides the organisation within the hierarchy, each block allows the definition of a set of properties with their respective type.
- Establishing a set of spatial groups, following the principles presented in subsection 5.5.3 and using the screen presented in Figure 5.10.
- Defining the categories, as explained in subsection 5.3.4 and using the interface presented in Figure 5.11. In that screen, we are using this feature to define a category that encloses all empty cells regarding the avatar's range of motion. We have used the hierarchy to separate the different types of floor cells from the empty cells. However, a specific type of cell, *breakable*, plays both roles. These breakable floor tiles are destroyed after being stepped by the character, meaning that, primarily, they act as a normal floor plank but, afterwards, they are an empty space. Therefore, the category *empty* also includes this type of cell, using an expression with the logical disjunction operator. In the bottom of the screen, it is possible to test the defined category and list the blocks that fit that same category.
- Creating the movement aids, referring to the notions described in subsection 5.3.5 and using the screen presented in Figure 5.12. In that screen, we are editing a rule to define the basic horizontal movement of the character within two adjacent cells. The pattern is composed by two side-by-side cells of the category *floor,* with their coordinates (0, 0) and (1, 0). It corresponds to possible movements of the avatar from (0, 0) to (1, 0) and from (1, 0) to (0, 0). Both these movements are presented with unitary cost.

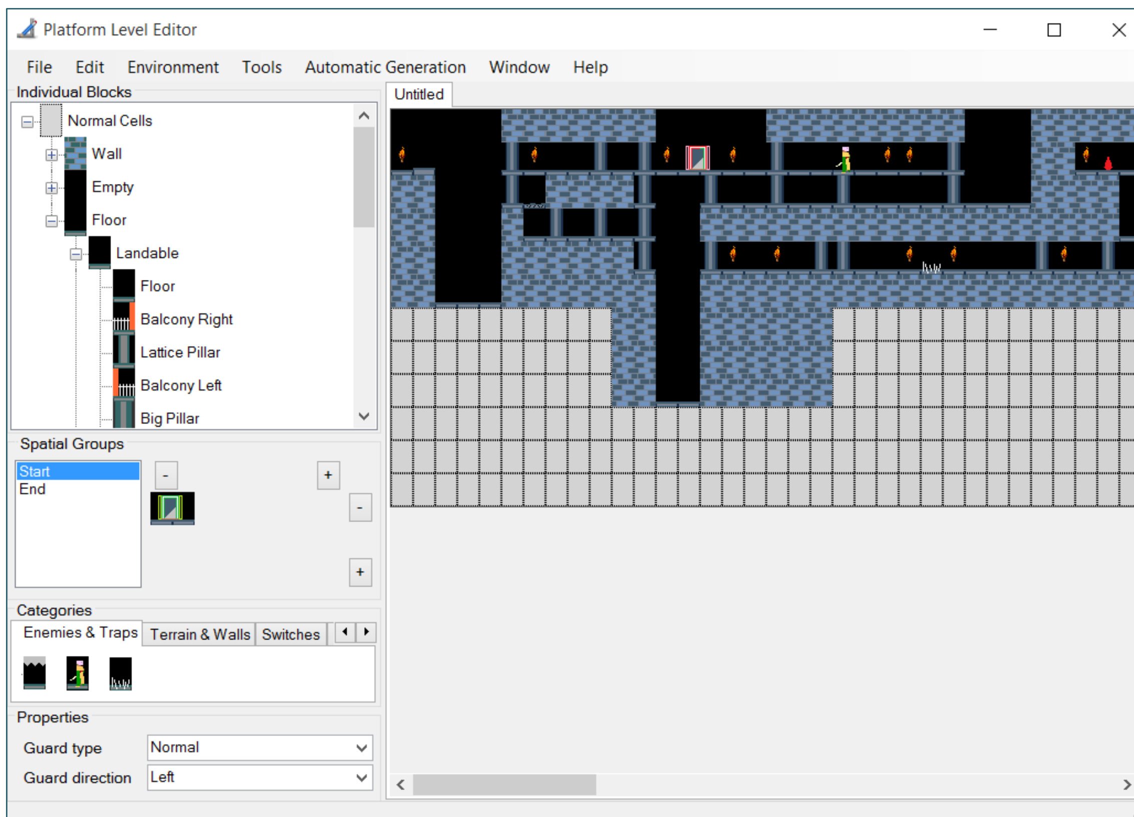


Figure 5.7 – Screenshot of the implemented level editor using the proposed level representation framework, while editing a level for the videogame *Prince of Persia.*

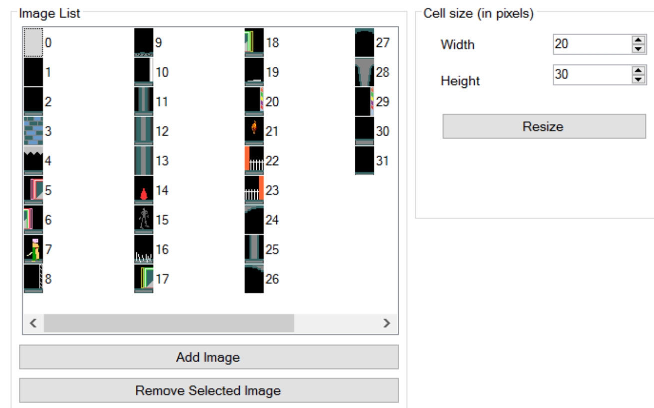


Figure 5.8 – Setting the images that compose the bitmap representations within the *Platform Level Editor.*

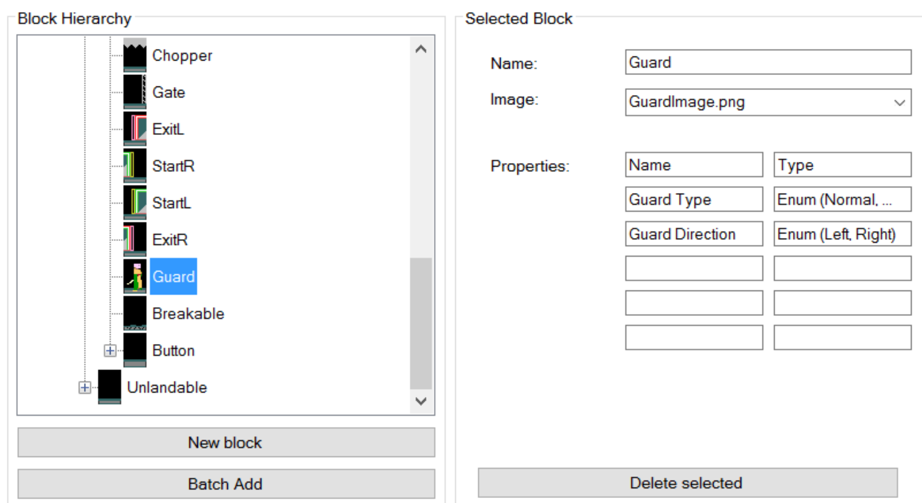


Figure 5.9 – Setting the block hierarchy for the game description within the *Platform Level Editor.*

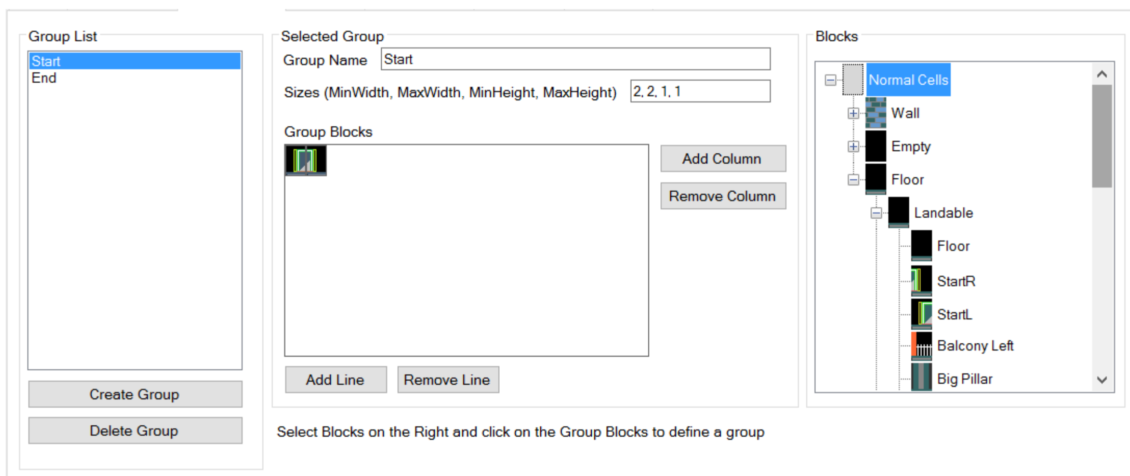


Figure 5.10 – Setting the spatial groups for the game description within the *Platform Level Editor.*

The content description is stored as an *XML* file (an example for the game *Prince of Persia* is presented in appendix A.4), making it easy to integrate with other modules or applications. In addition, the levels created within level editor are also stored as *XML* files, where the values for each block refer to elements of the framework (Appendix A.5 contains an example of a level for the game *Prince of*

Persia, using the description file of appendix A.4). Naturally, a level file is only valid when interpreted in association with the referred content description file.

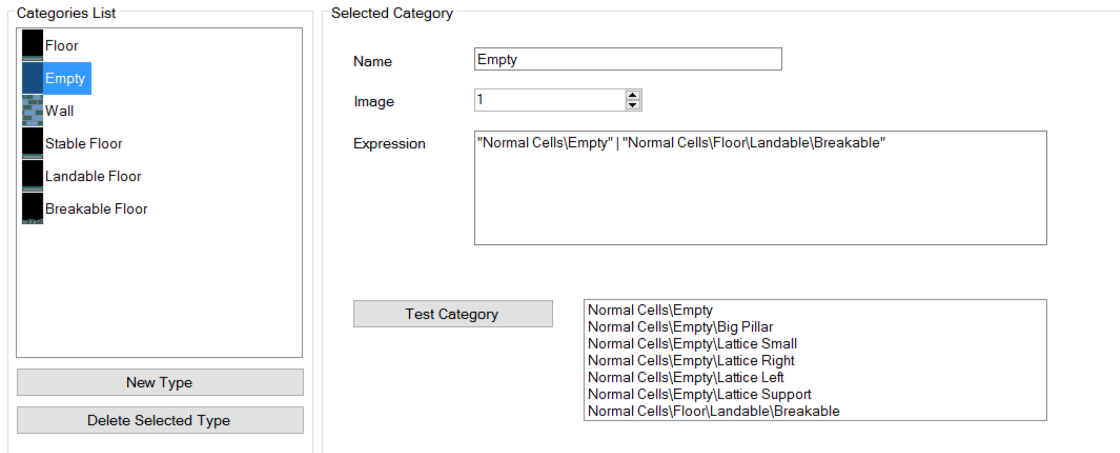


Figure 5.11 – Setting the categories for the game description within the *Platform Level Editor*.

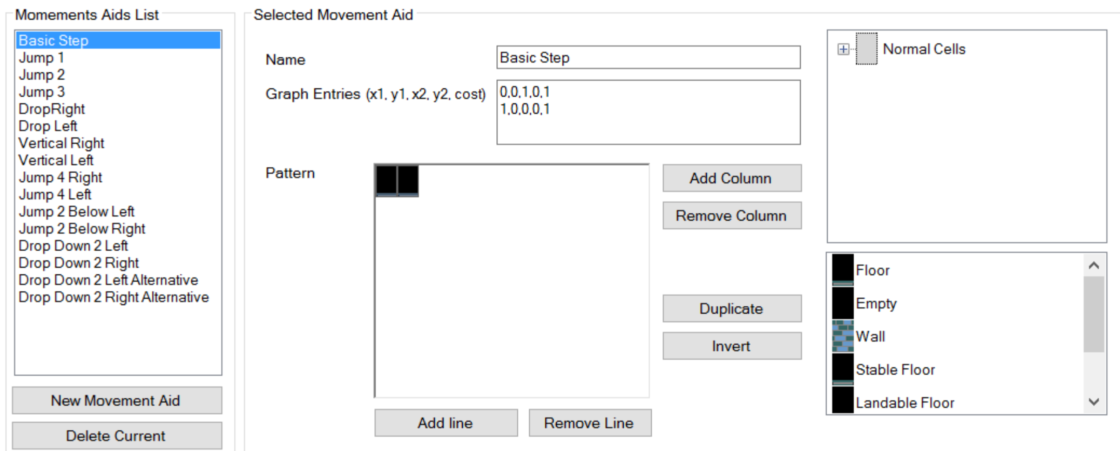


Figure 5.12 – Setting the movement aids for the game description within the *Platform Level Editor*.

The framework was mapped into a set of classes using the *C#* language, compatible with the platforms *.NET* and *Mono*, in order to make the concepts independent of the operating system. Therefore, it is possible to use the framework in different game engines supporting *C#*, such as *XNA* in the *.NET* framework or *Unity* with *Mono*.

Regarding external file formats for the integration with existing game content, we have implemented importers and/or exporters for the following videogames:

- *Rick Dangerous*, using the open source remake entitled *XRick*;
- *Infinite Mario Bros.* and *Infinite Tux*, based on the original source code; and
- *Prince of Persia* using the original technical documentation produced by Mechner (Mechner, 1989) and based on the reverse engineering document written by Calot (Calot, 2008).

To ease further integration with other videogames, the editor was implemented with a plugin-based architecture. This allows the independent development of new compatibility features with other videogames, which can be added to the application at any time. The implementation of a plugin for a certain game must follow the interface presented in appendix A.3 and include the following features:

- The model that describes the level content, represented within the previously presented framework, which consists of the *XML* file with that description.
- A level importer, which consists in the implementation of a method that reads and parses the original game level, using that information to create an equivalent version for the editor format. Additionally, the importer may include an interface window to allow the user to configure some import features. For instance, one common option to include in that window is a control to select which level should be imported from the original game.
- A level exporter, which consists in the implementation of a method that transforms the information of a level within the editor into the format of the respective game. In the same manner of the import feature and depending on the game, developers may include an interface window to allow the configuration of some export features, such as which level should be replaced in the original game.
- A test/play interface with the original game, allowing the user to create a level within the editor and access a button to test that level using the game engine. It consists in the implementation of a method where the approach is highly dependent on the architecture of the original game, as we can see with the two following different example cases. For the game *Prince of Persia*, our play/test feature uses the exporter feature to save the level inside the game engine and runs the game. For the game *Infinite Mario Bros.*, we use the save feature of the editor to create the *XML* file for the level and, after that, we run an adapted version of the game where we have replaced the original level generator with a level importer, which reads an *XML* file of a level represented within our framework.

The level editor was developed also to support our research regarding automatic level generation. Therefore, we have implemented features to allow the inclusion of different generation algorithms. As we will detail later in chapter 7, there are several distinct algorithms with different advantages and weaknesses. Besides, they may be designed for specific purposes in the generation process. For instance, some algorithms are meant for the creation of simple draft level structures while others serve the purpose of perfecting those types of draft structures. Following these principles, the editor allows the definition of a pipeline for automatic level generation with multiple consecutive steps, where the user may test different algorithms by defining their parameterisation and the order in which they are applied. We implemented such feature also with a plug-in based architecture, meaning that it is possible to continuously develop new automatic generation algorithms and easily include them in the editor. The definition of a generation algorithm has the following requisites:

- A set of parameters to be used within the generation process;
- The number of levels that the algorithm receives as input, for a certain configuration of parameters;
- The number of levels that the algorithm produces as output, for a certain configuration of parameters; and
- One method for the process of level generation.

The respective interface is described in appendix A.2. To serve as an example, we can consider a basic platform generator for the game *Infinite Mario Bros.* that simply creates sequences of terrain patforms at random heights, separated by a random distance and with random length values. The minimum, maximum and average values for those features can be considered as configuration parameters for the algorithm. Some basic restrictions might be applicable in the algorithm to make sure that all the generated gaps are crossable. In this specific case, regardless the configuration features,

the input of the algorithm is empty, as it does not receive any level, and the output is one level, even though it would be possible to adapt the generator to produce more than one level simply by iterating the core algorithm. In this last case, the number of levels to generate would be also a configuration parameter. It is possible to link any two-generation algorithms having that the output dimension of the first is equivalent to the input dimension of the second. To complete our example as a generation pipeline, one can think of an enhancement algorithm that is able to add aesthetical elements to a level. This specific algorithm takes one level as input and produces one level as output, the adapted version of the initial one. The previous platform generator can work seamlessly with this decoration algorithm, as the output dimension of the first is equivalent to the input dimension of the second. Figure 5.13 shows how the pipeline is configured within the editor for such situation and, in Figure 5.14, we present the output after the application of each of the algorithms.

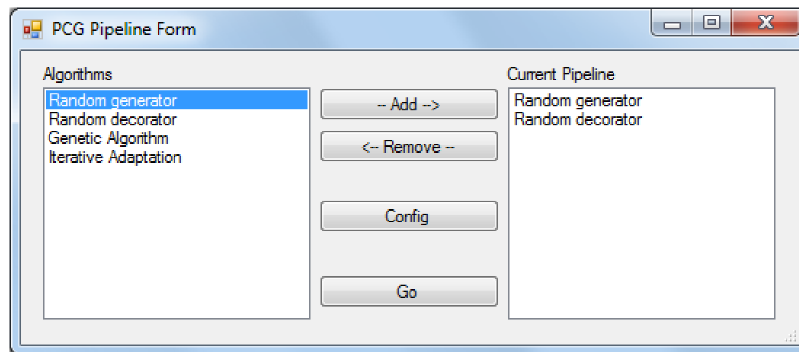


Figure 5.13 – Configuration window for the definition of the automatic level generation pipeline.

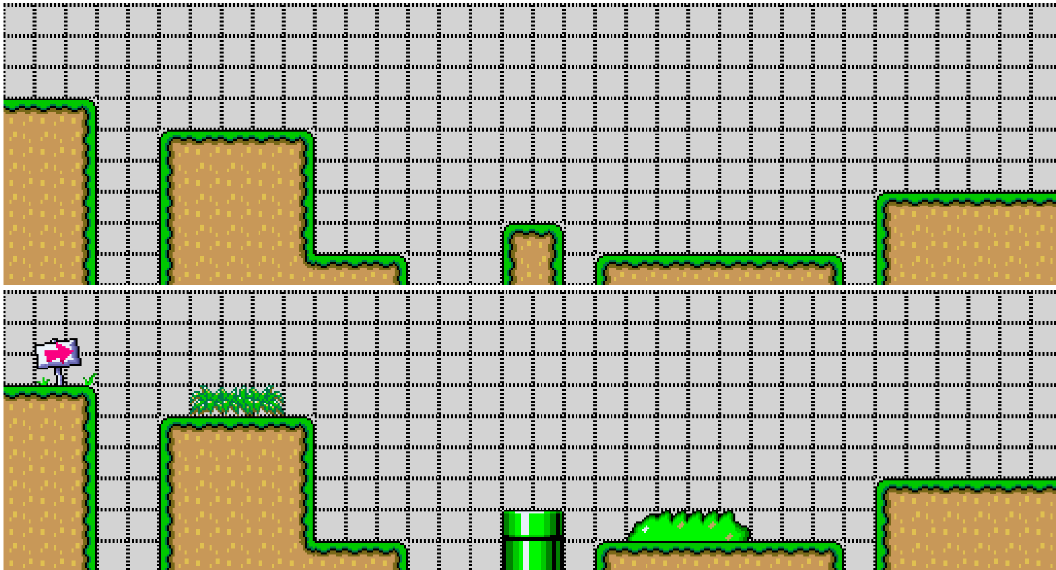


Figure 5.14 – Example of automatic level generation with a sequential set of algorithms. At the top, the result of the first algorithm generates the base level structure. At the bottom, algorithm produces a decorated version of the previous level.

The inclusion of this feature in our editor presented the potential of considering the generation process as a combination of independent steps. However, we have identified two main restrictions that unveil possible improvements for further research. First, we have referred that for two consecutive steps, the output dimension of the first must match the input dimension of the second. It is plausible to overcome this limitation by iterating a prior algorithm or truncating a set of previously generated levels. However, a simple pipeline does not allow the user to control how those processes will occur. For instance, if more levels are required at a certain step of the pipeline, they can be obtained by

repeating only the last algorithm or by repeating a subset of the previous algorithms in the chain. Secondly, the proposed sequential approach does not allow an algorithm to have two different input sources. For instance, it is not possible to have two different algorithms generating two different types of structures and a third one integrating them. We believe that an interesting point for further research is to define a framework to represent the generation process as a flowchart of different techniques, including a mechanism to explicitly represent iterations and trimmings. Such approach is a route to expand the proposed pipeline and overcome the identified issues.

To conclude, it is important to reinforce that this generation pipeline should be used with human supervision. Two different practical scenarios can be identified. First, it serves as a tool to test automatic generation algorithms and the possible combination between them. Secondly, it can aid the authoring capabilities of the editor, as it is possible for a designer to use the algorithms before a final stage of human perfecting. In both cases, the user should be aware of the features of the different techniques and the types of combinations that make sense.

5.5. Prototype Implementation

A first testing phase of the envisioned system architecture was implemented, using *Tux Likes You*, our own adapted version of *Infinite Tux*, allowing the users to play automatic generated levels in this game. The system was implemented with the following principles:

- The Game Manager and UI module provides the user the choice of playing a random level, an appropriate level to his/her skills or a level with a specific difficulty value. A simple difficulty classification was defined to tag levels as easy, medium or hard, mainly according to their overall failure probability.
- The Generation module implements a two-phase generator, with a configurable random platform creator followed by an adaptation algorithm. This adaptation algorithm adds or removes enemies to control the difficulty of the level and can also include aesthetic elements to make a more visually appealing level.
- The Game module was implemented as an adaptation of *Infinite Tux* providing some changes, namely the possibility of loading external *XML* levels, represented with our framework, and a gameplay recorder that stores the position of the avatar at every frame.
- The Statistics and Profiling module receives the gameplay data of every run. This is used to determine the percentages of success for each type of jump, which allows estimating a failure probability for the new generated levels. This difficulty prediction is based on the principles presented in chapter 4. Similarly, the failure probability is observed for each player for each type of jump and compared to the average in order to classify the player.

The Game Manager and UI module works as the main menu for the player and, as stated before, it is an intermediary for module communications. The game engine is also stored within this module. The combination of the two is, in practice, the game package as the user sees it. The remaining modules are accessible remotely via *HTTP*. After the difficulty selection, the UI requests a level to the server, which is generated and then returned by *HTTP* as an *XML* file. Once this file is received, the adapted *Infinite Tux* engine opens it and runs the gaming session. For each gaming session, a file is created to store the gameplay data, consisting of multiple lines with timestamps and the corresponding avatar positions for the respective timestamp. As the UI detects that the gaming session is finished, the gameplay file is sent to the server, once again via *HTTP*, where it is forwarded to the Statistics and Profiling module, which processes the data. After this, the initial menu is shown again and the process restarts.

5.6. Concluding Remarks

In this chapter, we have proposed a general architecture that may serve the implementation of a platform game using PCG, including several interesting enhancement features such as statistical studies using gameplay data and its further usage as input for profiled automatic generation algorithms. As seen in the last section, where we have presented a prototype game, this architecture is viable and it allows a good integration between the different modules. In chapter 8, we will present a second prototype using the same principles, a larger experiment where we validate different aspects of this dissertation. At that point, we will also provide additional implementation details about the generation algorithms and the statistical studies that the system potentiates.

The system behind the implemented prototype uses the framework for content description, which was particularly useful to ease the referred communication between modules, making them share a common representation. In general and regardless the purpose, the main advantage of this framework is its flexibility, allowing a simple and direct representation of the most common principles in this genre of games, suitable for the majority of grid-based platform videogames. We also believe that it is possible to use this framework with slight adaptations in other types of grid-based environments, such as strategy game maps. The main disadvantage is that the framework may be considered too generic for certain purposes. For instance, level model representations are so generic that they are not unique, meaning that is possible to define different models for one single game.

Additionally, the framework does not represent directly every type of component that one can find in a platform videogame, as it would be an impossible task in practice, due to the vast number of titles in this genre. However, with some abstractions it is possible to overcome some omissions. One typical case of this is the existence of dynamic platforms in a level. Even though the dynamic movement has not been considered, blocks and spatial groups can be used to represent it. For instance, an elevator can be represented by a group covering its shaft to state the respective range of movement. The graph representations are still possible as long as movement rules are established to state transitions from normal platforms to elevator shafts.

To complement our studies regarding automatic level generation algorithms, we have implemented a level editor. Besides supporting our tests during the development of new generation algorithms, which will be presented in chapter 7, this application also served to assess the application of the defined framework in the representation of level content within platform videogames. As seen, it is possible to edit levels for different games using this application, meaning that it serves as a valid level editor and that the proposed framework is compatible with the concepts of those games. The plugin-based architecture potentiates the integration with any formats, as it provides a mechanism to translate content from other games into our representation scheme. Finally, in chapter 8, we will also use the editor in the analysis of different approaches to create levels, namely to compare this traditional editing approach with a semi-automatic generation tool.

6. Using Graphs for Gameplay Representation

"Games provide rich interaction possibilities allowing for emergent gameplay that is sometimes hard to anticipate for the designer beforehand. Visual methods that allow for a more explorative data analysis are therefore a promising and increasingly important tool for game analytics."

(Wallner, 2013)

6.1. Introduction

A platform videogame has strong emphasis on movement and actions. In order to represent possible character movements in a level, our research comprised studies regarding graph representations and calculations. Graphs can be described as models that define mathematically the relations between pairs of objects. They consist of nodes (or vertices) and edges that connect them. This type of graphical representation is suitable to depict several distinct situations that one can find in the quotidian life. For instance, nodes may represent people whereas the edges denote friendship relations between two people. As a different example, nodes can be seen as train stations in a map and the edges represent railroads connecting those stations. Graph theory refers to the study of these structures and their usage to represent existing concepts as the two prior examples and solve problems within their domain.

Graph theory is a base knowledge within computer science. For additional references within this subject, we point to Bondy and Murty's book (Bondy & Murty, 1976), where classic graph theory is presented, to West's introductory book to graph theory (West, 2000), focusing the mathematical background of graph theory, and to Jungnickel's book (Jungnickel, 2008), which contains a more practical approach focusing on the contemporary problems and algorithms. It is also worth mention the software tool *Graphviz*[1], a popular tool to represent and draw graphs. In the scope of this work, it was used for documentation as well as within the implemented software.

This chapter presents our studies regarding graphs, focusing their possible applications in the automatic level generation domain. To start with, in section 6.2, we will refer the existing work regarding videogames where graphs have been used, in particular regarding path representation within a level. Afterwards, in section 6.3, our graph representation will be described and the main conditions for its usage will be referred. Two different methods will be presented for mapping level content into graph representations that describe gameplay within those levels. Furthermore, in section 6.4, we will reveal how those graphs can be analysed regarding gameplay. This analysis allows the identification of specific situations in gameplay, such as the existence of multiple path alternatives or dead-ends, among others. These structural features represent additional knowledge about the level structure, an innovative approach to enhance automatic generation algorithms. Lastly, in section 6.5, we will finish this chapter with our concluding remarks.

[1] http://www.graphviz.org/

6.2. Using Graphs to Represent Level Structures

As stated in the last section, graphs are possible representations for different situations. They can be used in physical problems, for instance, to represent road structures, allowing the calculation of routes between two points, or may refer to more figurative concepts, for instance to describe associations among webpages, allowing the establishment of ranking for search results. In this section, we present some researches in which graphs were used in videogames, focusing their usage to represent level structures.

In chapter 4, we referred gameplay metrics as an important feature to analyse the player's emotional state and his/her feelings within the gaming experience. In this chapter, gameplay metrics are directed to the importance of certain structures and how they have influence over the strategy, such as existing routes, central passage zones, among others. Naturally, these two different aspects of gameplay analysis can be correlated.

To start with, one of the most popular techniques to analyse gameplay in a level is entitled heat map, "a map that uses colour or some other feature to show an additional dimension" (Fry, 2004). In other words, it consists of a two-dimensional map representing a level, where colours or shades are used to reflect the density of a certain variable depending on the position in the referred map. Heat maps can provide good spatial information of certain gameplay events and variables but they lack dynamism and interactivity. Furthermore, the basic data lacks semantics, as the coloured areas do not represent anything in particular before an appropriate human interpretation of that same data. To understand the meaning of such data, complementary analysis or additional methods are required.

It is possible to point some work in which the principles of heat maps are expanded, namely by detecting correlations regarding events within a map. The work of Hoobler *et al.* (Hoobler, Humphreys, & Agrawala, 2004) shows a method to visualise competitive behaviours in multi-user virtual environments, focusing FPS games. This is done in real-time during the game by adding visual elements such as trails, allowing a spectator to observe with more detail the multiple events that are occurring at the same time. Andersen *et al.* (Andersen, Liu, Apter, Boucher-Genesse, & Popović, 2010) proposed gameplay analysis based on the projection of states within a graph representation. The authors mapped states into the nodes of a graph, having edges to represent possible state transitions. For visual interpretation, nodes are represented with different sizes, depending on the number of players that reached their respective state. In addition, those same nodes are coloured according to the probability of winning for the players who reached each one of the possible states. Also, cycles can be detected, which represent situations of setback penalties. As another example, *Play-Graph* (Wallner & Kriglstein, 2012; Wallner, 2013) is a methodology and a visualisation approach for the analysis of gameplay data, based on node-link diagrams where it is possible to visualise player's progression. The authors recur to difference graphs to detect specificities of a certain subset, which may be related to the user profile, such as the gender or age, or to gameplay choices, such as the selected faction or avatar features.

The examples presented in the previous paragraph aim to extract data after gameplay sessions, which is particularly useful to understand mass behaviours and detect specific gameplay details that would be impossible to find otherwise. However, it is also useful to consider different approaches in which similar methods can be used in earlier design stages, providing a predictive analysis. A common method that is used in large environments is entitled navigation mesh (Snook, 2000; Tozour, 2002) and consists in the construction of an accessibility graph, starting with the original terrain mesh, which is then processed and transformed in a less granular structure of possible way points that might serve, for instance, path calculations for artificial players. Considering its usage in shooter games or similar, these waypoints can be annotated regarding visibility and coverage properties, which might

serve the definition of strategies for such artificial players. An example of this line of thought can be found in the work of Darken (Darken, 2007), where the author presented a method using virtual agents that simulate the behaviour of the player within a level, in order to explore the navigation mesh and tag waypoints using an empirical model of human target detection. Regarding the level structures of platform videogames, Bauer and Popović (Bauer & Popović, 2012) used a probabilistic search, namely the Rapidly-exploring Random Tree (RRT) algorithm (LaValle & Kuffner Jr, 2000, 2001), to sample a level's state space. This produces a tree with numerous game states as the nodes, which are condensed into a graph representation by applying the Markov cluster algorithm proposed by van Dongen (van Dongen, 2000). The resulting information is a graph suitable to be used within a level editor during the creation process. By overlapping the graph with the level structure, the designer can identify the areas of a level that are reachable and analyse how the players can reach them. As the graphs are updated interactively, the designer can also analyse the impact of certain changes during the editing process.

6.3. Mapping Levels to Graph Representations

When a designer creates a level, it is possible to ask him/her to describe the possible main movements within that level in a graph structure. However, to use this type of approach in a system that generates levels from scratch and, therefore, without that possibility, it is important to think that manual graph creation is not an option. Consequently, it is important to have a method to create those graphs automatically, based on an initial level representation. A plausible solution would be the usage of physical simulations based on the game rules, or the game engine itself, to calculate all possible movements for the avatar. However, this would be a complex and time-consuming task, which could compromise a process of online level generation. For instance, an algorithm based on a *generate-and-test* approach, as explained in section 2.3, computes several level variants during the process, in which applying a physical model for each alternative makes the process computationally expensive. Even the alternatives presented in the previous section, such as the RRT, denote computational times that are prohibitive for a vast analysis within an online generating process. The alternative presented by Bauer and Popović referred calculation times of about one second to recalculate the graph of a certain level. In addition, this type of simulation would require different implementations for every distinct game, as each game has its own rules. In this section, we will present two alternatives for the automatic graph extraction, with applicability in the generation algorithms that will be presented later in sections 7.4 and 7.5, and that can serve other complementary studies regarding gameplay. One of the approaches is directed to analyse the level structures in a design phase, prior to its usage by the player, while the second one is directed to the construction of similar information based on data retrieval after some gaming sessions.

Beforehand, it is important to clarify the requirements of the graphs that we will consider henceforth. As seen, a physical simulation featuring an extensive combination of possible moves within a level would give the knowledge about accessible areas, existing routes and other features, yet with a prohibitive computational effort and with an unnecessary level of detail. For instance, when considering a single platform, the generation algorithm does not need to know every possible combination of jumps back and forth that allows reaching the end of that same platform. A direct information stating that the avatar can cross the platform from one edge to the other is enough for an algorithm to define possible paths avoiding the aforementioned computational bottleneck.

In chapter 5, we have presented our framework for level representation in which we have considered a spatial discreet representation as a grid. In order to promote coherence within our system, the gameplay graphs representing possible avatar movements and actions within a level follow that same granularity. In practice, a graph for a level has one node for each cell in which a steady state regarding gameplay can be identified in the corresponding position, and the edges connect nodes when a certain

action allows the transition between two nodes. It is also important to refer the usage of directed graphs, as some transitions might be unidirectional, such as the character falling into a hole, which has no way back. To provide a clearer notation, we will avoid mixing representations. Therefore, all edges are directed and the cases of undirected edges connecting vertices v_1 and v_2 are represented as a directed edge from v_1 to v_2 and another directed edge from v_2 to v_1. Besides the benefit of the notation, in some cases this distinction is mandatory because costs from v_1 to v_2 and from v_2 to v_1 might be different, as they may represent, for instance, a difficulty measure. Figure 6.1 presents a sample of the first level of the game *Prince of Persia* and a graph representing the possible avatar movements within that level, following the presented principles.

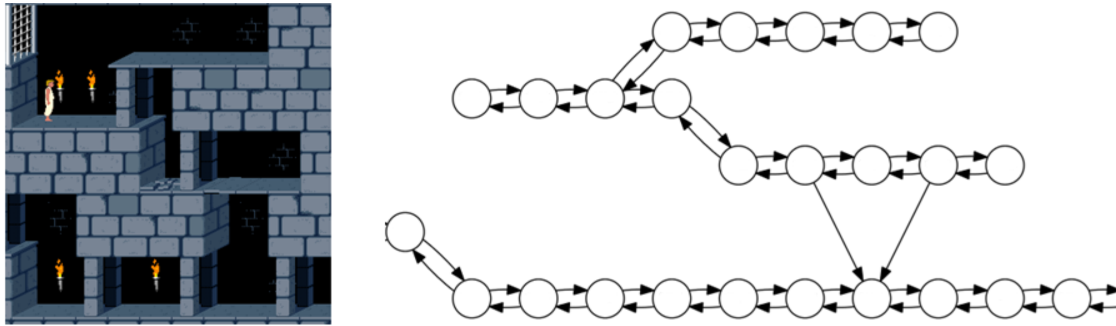


Figure 6.1 – A sample of the first level of the game *Prince of Persia* (on the left) and the corresponding graph (on the right).

6.3.1. Graph Extraction Based on Pre-defined Rules

In section 5.3, we have presented a framework for a generic level representation suitable for common *platformers*. Recalling the proposed framework, the content of a level can be described within a grid, in which each cell contains a value from a predefined set of blocks. We have also defined the existence of specific combinations of blocks, designated as *groups*, and an additional tagging mechanism based on the application of logical operators, designated as *categories*. This representation allows mapping the structural information of a level and, eventually, its implicit visual expression. However, it lacks semantic description that can allow the inference of gameplay data. For instance, it is not possible to perform a simple validity test to check if there is a valid way from the beginning to end of the level. In a common design situation, this is not an issue, as one can assume that the designer has knowledge and perception to create correct scenarios and that a further testing phase can also confirm that validity. Nevertheless, in an automatic level generation scenario, the human perception is not available and levels are typically used without a testing phase.

We have presented, in subsection 5.3.5, a mechanism based on rules to associate certain cell combinations to a corresponding sub-graph. This is done by associating certain cell patterns to sub graphs that represent the avatar movement for that specific situation. In Figure 6.2, it is possible to observe two example rules used to define the movement for the videogame *Rick Dangerous*. On the left part, it is possible to observe the rule to define the horizontal movement of the character and, on the right part, the movement among ladder cells is represented.

For a certain game, a set of pattern matching rules can be defined. The complete set of rules is applied by going through the entire grid of the level, searching for matches. This process has shown to be particularly efficient in situations with a small range of movements, such as those with *rotoscoping* animations, as *Prince of Persia*, for example. The graph formerly presented in Figure 6.1 was generated using this pattern matching approach.

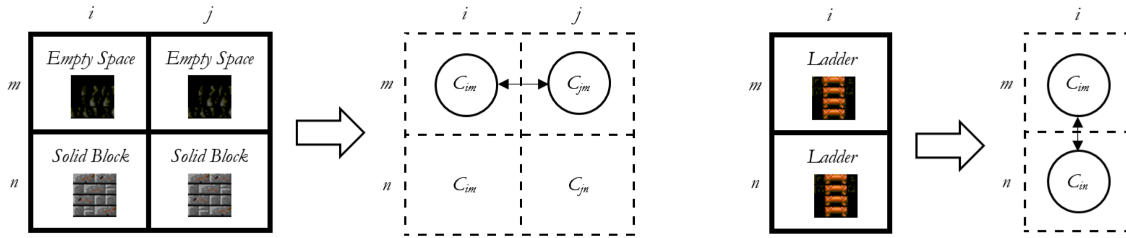


Figure 6.2 – Example of two rules for graph construction in the game *XRick*, consisting of a pattern and the corresponding graph entry.

A natural concern and important aspect to take into account is that pattern matching in two dimensions is still a computationally expensive task. Considering a full text of n by n characters and a pattern of m by m characters, in an alphabet of q characters, a naïve implementation presents a worst case scenario in the search for a pattern of $O(m^2n^2)$ and a mean case of $O((q/(q-1)).n^2)$. Even though we have applied this implementation in our prototypes, we are aware of the possible alternatives to lower the demand. Baker (Baker, 1978) proposed a method that reduces the complexity of the problem to $O(m^2 + n^2)$, both for worst and average case, using a multistring searching algorithm. Karp and Rabin (Karp & Rabin, 1987) tackled the problem by mapping each line in the pattern as keys for an hash table, which are then searched in the scope of the whole document, also achieving results of $O(m^2 + n^2)$ if possible collisions regarding the hash-table are ignored. Zhu and Takaoka (Zhu & Takaoka, 1989) adapted the previous approach with pre-computation steps for partial optimisations. Lastly, the most effective technique that was found in literature has been proposed by Baeza-Yates and Régnier (Baeza-Yates & Régnier, 1993), which reduces the computation complexity of the task to $O(mn^2)$. The proposed algorithm searches the content in alternating lines, spaced m lines apart to detect possible cases of matching, while the remaining lines are only analysed for the detected potential matches. The authors claim that their approach is up to two times faster than the alternatives. Although in an automatic generation process, such improvement is not enough to change the viability and the scope of an algorithm, it is still an interesting enhancement to consider in further developments.

6.3.2. Graph Extraction from Example Levels

As an alternative to the method for graph extraction presented in the previous subsection, we have also tested the possibility of having the system automatically learning the most relevant rules to be used to sketch the movements of the avatar, based on gameplay testing sessions. This is suitable for situations in which the character has a wide range of motion that is difficult to express within a reasonable sized set of rules. We believe that a possible scenario of usage for this alternative is the case where the game has already been created with a set of humanly designed levels and one wants to add new game content automatically to expand the current set of levels without an additional designing effort. We have tested this alternative with our prototype *Tux Likes You*, described in section 5.5, with some interesting results. The game was configured to record the avatar positions each time the user played the game. Representing the recorded positions into the map allows identifying the main areas where the player was positioned. Figure 6.3 shows a level sample on the left part and, on the right part, it is possible to observe the most common regions for the avatar positions. Areas were faded out inversely proportional to the respective frequency of the avatar presence.

Additionally to the simple area analysis inspired by the concepts of heat maps, we have detected which movements represented cell transitions and used that information to create level graphs. In Figure 6.4, we present, for the same level situation, the obtained graph for the avatar's movements. The generated graph and the original level structure allow the automatic extraction of some move-

ment rules. This can be achieved by analysing the cell neighbourhoods in pairs of vertices that represent transitions between equally spaced cells. For instance, considering only this small level sample, our system detected that the transitions between side-by-side cells occur frequently in the situations presented in Figure 6.5. This means that those situations represent potential rules for that horizontal movement. To confirm those rules, the potential situations are searched in the whole set of levels. If whenever those situations occur, the corresponding graph entry can be found, then the rule is confirmed. In the example case of Figure 6.5, all rules are confirmed except the third one, since there are several occurrences of that pattern without a corresponding graph entry.

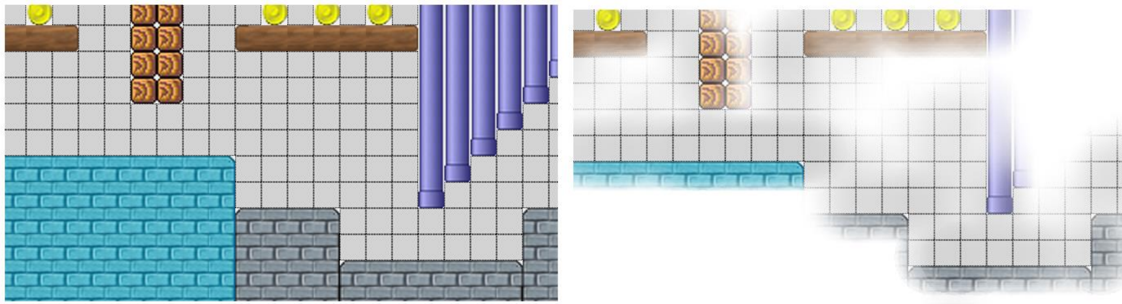


Figure 6.3 – Gameplay mapping of avatar positions.

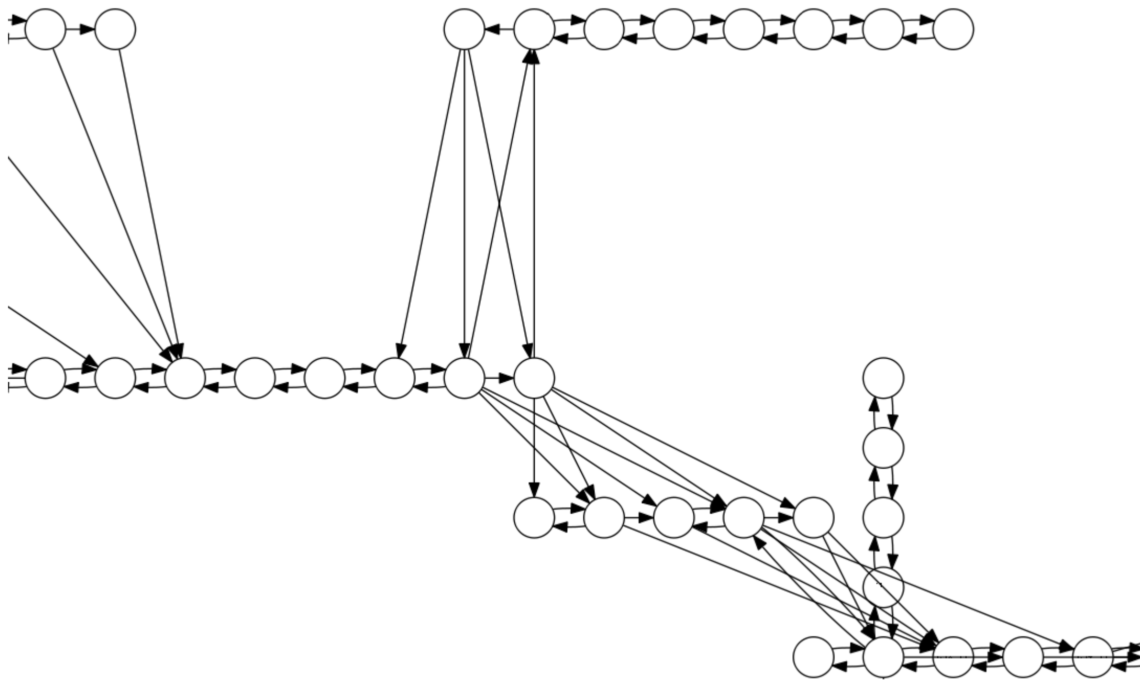


Figure 6.4 – Obtained graph based on gameplay data

In this process, we have detected two major issues that could compromise the usage of this technique. First, even though it is possible to map the majority of the level with just a small number of gameplay sessions, it is particularly hard to achieve global coverage. Small level parts tend to be unused even though they are reachable. In second place, a simple bug in a gameplay session can compromise the whole reasoning. For instance, if the avatar crosses a solid block inadvertently, that move is likely to be assumed as plausible, which may lead to inaccurate conclusions. To solve both the previous situations, we transpose to this domain the concept of a low pass filter. The erratic detection of transitions due to inadvertent moves, caused by possible game implementation bugs, can be corrected by ignoring transitions that only occurred sporadically (formally, with a percentage of existence under a

certain threshold). In the same manner, the confirmation of potential rules should not search for 100% matches but a value over a certain threshold, which should be tuned according to the existing data.

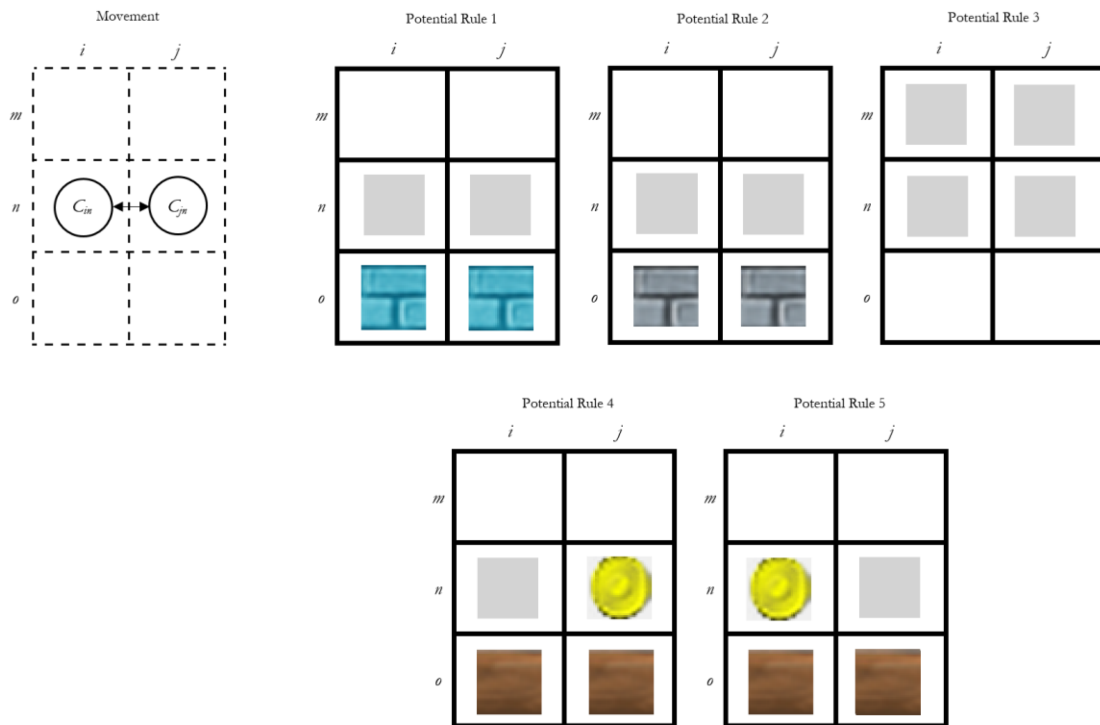


Figure 6.5 – Example of five frequent situations for horizontal transitions.

6.4. Graph Analysis

Subsequent to the previous methods to automatically define a graph that sketches the avatar transitions within a level, a wide variety of analysis based on the level structure is possible. We have referred the importance of testing if a level is winnable. This is possible directly with the level graph, using for instance a breath-first search to find a possible walk between the nodes that correspond to the starting and ending position of the level. To optimise this search, a more efficient algorithm can be used, such as A* (Hart, Nilsson, & Raphael, 1968). In addition, we have stated that the level representation framework considers the possibility of establishing costs for the edges that are generated. Considering that those costs represent estimators for difficulty, it is also possible to find the easiest route from the beginning to the end of the level, for instance recurring to Dijkstra's algorithm (Dijkstra, 1959). In a more ambitious perspective, the generation of a good level takes into account more than optimal routes. It can also include the creation of path detours, force additional level exploration for items, and several other types of creative content richness. In the remaining of this section, we will explore additional graph analysis that may infer certain level features and thus feed generation algorithms with extra useful information.

6.4.1. Initial Graph Processing

In order to reduce the computational effort of further processing, we propose an initial step of graph compression, which transforms the original generated graph into a reduced version with a smaller amount of vertices and edges. This compression involves removing vertices that are not significant to path computation processes and that can be seen as obvious intermediate steps in major transitions. In particular, the two following rules are applied:

- If a certain vertex v has exactly one incoming edge e_0 (from v_0) and one outgoing edge e_1 (to v_1), this means that, regarding path calculations, v is just a transitional step from v_0 to v_1. In this case, vertex v and edges e_0 and e_1 are removed from the graph structure. A new edge is created directly from v_0 to v_1 with a cost corresponding to the sum of the defined costs for e_0 and e_1. This compression is depicted in Figure 6.6.
- If a certain vertex v has exactly two outgoing edges e_{O1} and e_{O2} (to v_1 and v_2, respectively) and two incoming edges e_{I1} and e_{I2}, from the same vertices, this means, in a similar way to the previous rule, that the vertex v is an intermediate step in the connection between v_1 and v_2 in both directions. Again, this vertex is removed and the original edges are replaced by one edge from v_1 to v_2 with a cost value summing the costs associated with e_{I1} and e_{O1} and another edge from v_2 to v_1 with a cost that sums those from e_{I2} and e_{O2}. Figure 6.7 depicts this compression.

Figure 6.6 – Vertex compression illustration for unidirectional passages. On the left, the original graph is presented and, on the right, the resulting graph is shown.

Figure 6.7 – Vertex compression illustration for bidirectional passages. On the left, the original graph is presented and, on the right, the resulting graph is shown.

For representation purposes, the original nodes are represented as circles and the removed nodes are marked with a dot. Bidirectional passages with removed intermediate points are represented with overlapping edges to distinguish from original bidirectional transitions, represented with non-overlapping edges.

An example of the compression mechanism is provided in Figure 6.8, which contains a compressed version of the original graph represented previously in Figure 6.1.

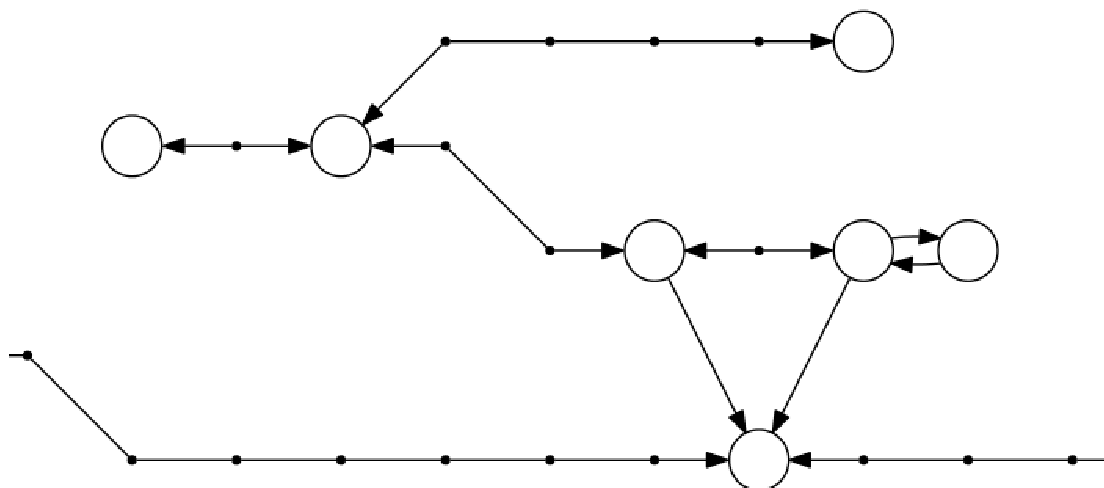


Figure 6.8 – Example of a compressed graph. The compressed nodes are represented with dots.

Some restriction might apply in vertex removal. For instance, vertices corresponding to the start and end position should not be eliminated, as they have an active role on the level. This was implemented with the definition of a set of irremovable vertices, created from a set of rules.

Furthermore, some particular compression schemes might be deliberated depending on the game that is being considered. For instance, in the videogame *Prince of Persia*, it is common to have the situation represented on the left part of Figure 6.9, which will be represented by the corresponding graph that can be seen on the right part of that same figure. The triangular shape represented on the graph image by the three interconnected nodes can be merged into a single node without compromising path calculations. As another example, with a simple rule set, hill platforms in the game *Infinite Tux* tend to create excessive vertical movement alternatives and prevent vertex compression, such as the case presented in Figure 6.10. Vertices corresponding to the middle cells of the hill platform can be removed without compromising path analysis, again using a set of rules of the process.

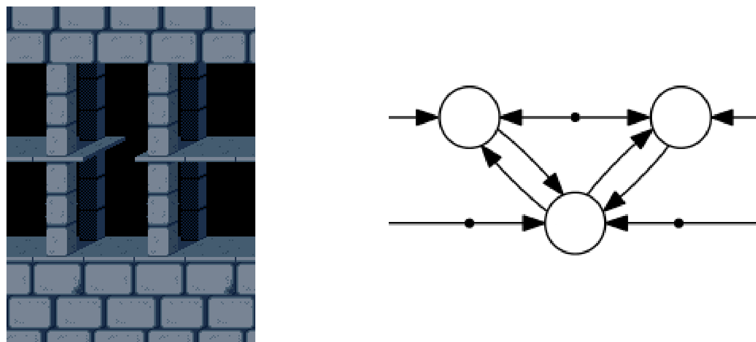


Figure 6.9 – An example of a potential additional graph compression in the game *Prince of Persia*.

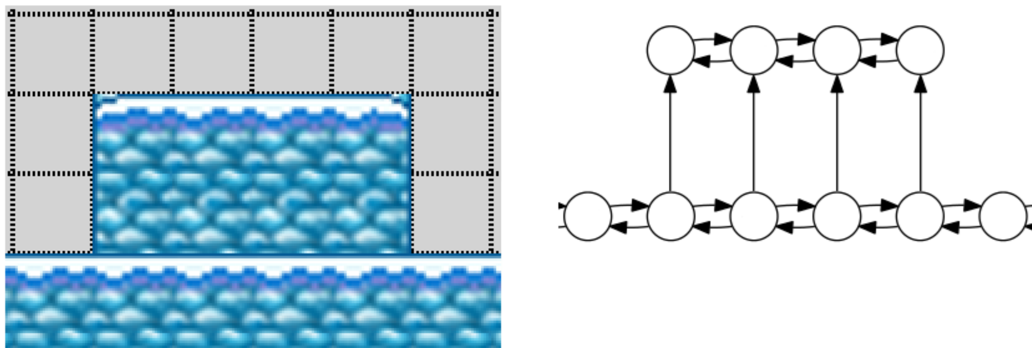


Figure 6.10 – An example of a potential graph compression case in the videogame *Infinite Tux*.

6.4.2. Path Extraction and Vertex Classification

Having a compressed graph representing the level, the next step consists in categorising the role of each vertex, considering all routes between the start and the end of the level. The algorithm searches every possible path from the level entry to the level exit, using a breadth first search and ignoring all alternatives that visit the same vertex multiple times.

Following this step, every vertex is labelled with one of the following values:

- **Mandatory**, if the vertex exists on all calculated paths.
- **Optional**, if the vertex is only on some of the calculated paths.

- **Dead-end**, if the vertex is not part of any possible path and has only one outgoing edge to a certain vertex and one incoming edge from that same vertex, meaning that if the character reaches that position he/she is forced to go back.
- **Unreachable**, if the game character cannot reach the position that corresponds to the vertex.
- **Vain**, if there is no way back from that vertex to the main path.
- **Path to dead-end**, to all remaining vertices that, by exclusion, have the only purpose of providing passage from mandatory or optional vertices to a dead-end.

To complement this information, a tree is created representing path segments and alternatives for each segment, recursively, in which leaf nodes represent trivial graph transitions.

Example

To serve as an example, we have manually created the level structure represented in the background of Figure 6.11. It does not contain enemies, gates, switches or any traps. This type of structure benefits from the previous classification mechanism, which can serve adaptation algorithms, as will further see in section 7.4. The starting point is the door on the left side and the level ends when the character reaches the door on the right. The system extracted and compressed the level graph presented as an overlay on that same image. Nodes have been named according to their coordinates on the grid, also marked in the image for convenience.

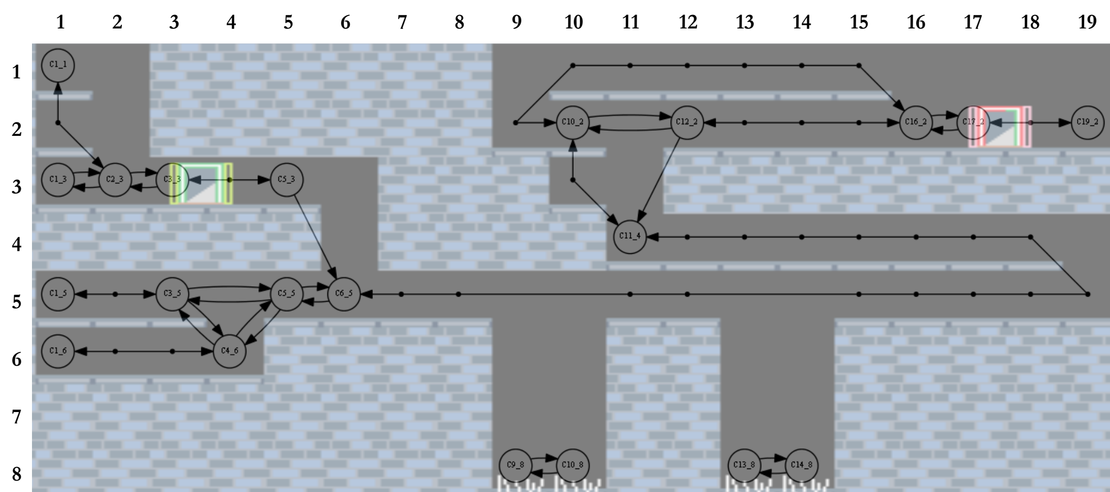


Figure 6.11 – Graph generated from the example level.

The system classified the vertices as follows:

- **Mandatory**: C3_3, C5_3, C6_5, C11_4, C10_2, C16_2, C17_2.
- **Optional**: C12_2.
- **Dead-end**: C1_1, C1_3, C1_5, C1_6, C19_2.
- **Unreachable**: C9_8, C10_8, C13_8, C14_8.
- **Path to dead-end**: C2_3, C3_5, C4_6, C5_5.

In addition, the tree presented in Figure 6.12 was calculated, which presents the different route alternatives.

Furthermore, routes to the existing dead ends were also calculated. In this case, the system identified the following possible routes:

- C2_3 → C1_3.
- C2_3 → C1_1.
- C5_5 → C3_5 → C1_5.
- C5_5 → C4_6 → C3_5 → C1_5.
- C5_5 → C4_6 →C1_6.
- C5_5 → C3_5 → C4_6 →C1_6.
- C17_2 → C19_2.

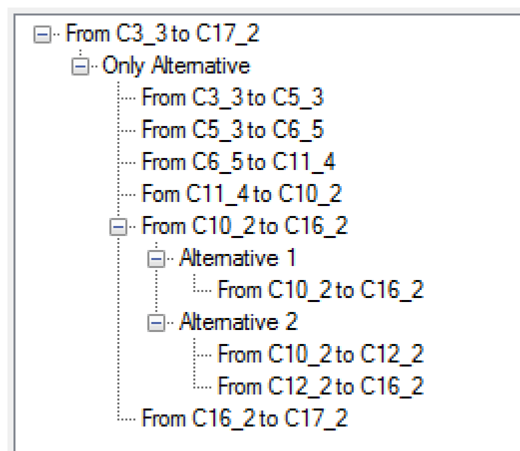


Figure 6.12 – Calculated tree with all possible paths from the beginning to the end of the level.

These represent small segments that are optional to accomplish the level goal but in which it is possible to include additional gaming entities to make the level more complete, as we will further see in section 7.4.

6.5. Concluding Remarks

In this chapter, we have explored graph-based mechanisms that potentiate their usage in automatic level generation. Our contributions address the process of creating a graph representation that depicts the structure of a level and a tagging mechanism that enriches those representations, providing semantic information to the existing nodes.

Regarding the creation of graph representations, we have presented, in section 6.3, two different alternatives for distinct scenarios. In subsection 6.3.1, we have applied the principles of our framework, described in section 5.3, to create the graphs by applying a set of rules to a level, based on patterns. This is an interesting approach to use in a design stage, where it is possible to analyse the reachability of the various zones and detect if it is possible to finish the level. It has the advantage of providing this type of assessment only with a simplified model of the game mechanics, without requiring the game's physical model. Alternatively, we have presented, in subsection 6.3.2, a graph creation mechanism that uses gameplay information for the process. One possible application is the confirmation of the established rules of the prior process.

In relation to our tagging mechanism, it was designed to analyse level structures, considering the specificities of platform videogames and the type of challenges that they represent. As seen in the

related work, presented in section 6.2, graphs were already used for similar purposes. They have strong emphasis on the context of AI algorithms for path finding problems, normally used in the development of artificial players. In this context, level parts are normally analysed regarding their meaning for the player's strategy, for instance, to detect hiding spots. A similar type of categorisation was desired in our work but with two main differences. First, the desired categorisation was intended to be directed to the specific genre of platform videogames, though without one particular title in mind. Secondly, the categorisation was intended for design purposes instead of strategy related usages. Therefore, the proposed categorisation mechanism for graph vertices takes into account their role in the global level structure. Nodes are tagged as *mandatory*, *optional*, *dead-end*, *unreachable*, *vain* or *segment before dead-end*. Whereas the number of categories is considerably small, they open different paths for further research regarding generation algorithms, particularly in refining phases. For instance, levels are not normal mazes, hence a dead-end can represent more than a meaningless alternative. A *dead-end* can be materialised as a hidden zone containing secret items or even be transformed into a mandatory region regarding the player's course with the inclusion of a required item for the player's progress. Such type of changes can only be performed by automatic generation algorithms with the referred extracted knowledge about the level structure. These and other type of complementary changes will be explored with more detail in the next chapter.

7. Automatic Level Generation Algorithms

"Platformer level generation is a more difficult problem than level generation in either RPG *or strategy games, since very small changes can change a whole level from challenging to physically impossible."*

(Compton & Mateas, 2006)

7.1. Introduction

In this chapter, we will dig into the algorithmic details of the approaches that have been implemented in the automatic level generation for platform videogames. Multiple methods have been considered in this topic, wherein some of them have been slightly revealed. In section 7.2, we will cover several of those approaches with more detail, in order to create a steady ground to this topic. In the following sections, from 7.3 through 7.5, we will present our progresses in this area, where we will show our techniques and algorithms for automatic level generation, covering the details behind them. Next, in section 7.6, we will present a categorisation to describe the differences between the several distinct techniques and bridge the concepts of that taxonomy to the level design patterns presented in section 3.3. In section 7.7, we will describe an application that works as a semi-automatic authoring tool. In that application, the users can manually project a level by instantiating and parameterising quest patterns, which will be physically generated by multiple generation algorithms, according to the desired features. To finish this chapter, in section 7.8, we will summarise our achievements and present our concluding remarks.

7.2. Related Work

As we have seen in subsection 3.3.3, the study of platform videogames in academic context was pioneered by Compton and Mateas (Compton & Mateas, 2006). The authors defined a conceptual model to characterise this genre of videogames and identified the main design patterns to structure a level. Besides, the notions of rhythm and rhythmic pattern were introduced as an approach to automatic level generation. It is stated that "[rhythmic] patterns provide the mechanism for grouping individual components into a longer sequence, while still maintaining rhythmic movement for the player". These notions were defined already considering the possible usage of the proposed model as a future mechanism for automatic level generation.

The study referred in the previous paragraph inspired a line of work based on the principles of rhythm, which we henceforth refer as rhythm-based generators. Smith *et al.* (Smith et al., 2008) presented a framework to analyse level content based on a hierarchy of entities, which has been described in subsection 5.3.1. That hierarchy organises information within a level with strong emphasis on *rhythm groups*, which represent the main structures when decomposing levels into sections. The principles of this framework lead to the implementation of a rhythm-based level generator (Smith et al., 2009). Regarding the generation algorithm that is proposed, it is composed by the following two main steps:

- The creation of a set of actions that the user has to perform, constrained to form a rhythmic pattern.
- The conversion of the preceding set of actions into an arrangement of geometric features where those actions are physically possible. This conversion is implemented using a grammar, and the parameterisation of the resulting geometric features is done using a physics model.

The sole usage of these steps in the creation of a level suffers from over-generation, as the authors stated, resulting in undesired levels with excessive repeated patterns. To overcome this issue, it has been proposed in the article to generate multiple levels at the same time, which are then analysed and classified by (virtual) critics according to certain design heuristics, to choose the best generated level from the output set. Finally, to complete the generation process, the authors suggest performing a global pass for level perfecting. In this case, they corrected the platform positions to avoid floating platforms and added collectible coins for scoring purposes. Figure 7.1 sums the architecture of the system that was described. The same principles were used as part of subsequent systems, namely *Launchpad* (Smith & Whitehead, 2010, 2011), *Rathenn* (Smith, Gan, Othenin-Girard, & Whitehead, 2011) and *Endless Web* (Smith & Othenin-Girard, 2012).

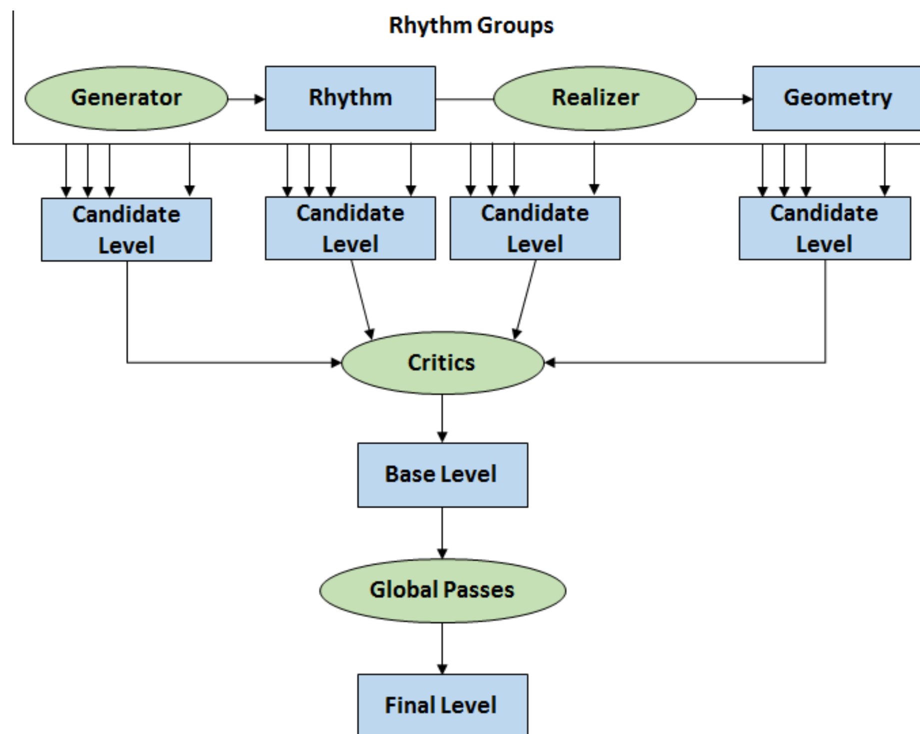


Figure 7.1 – Rhythm-based level generation architecture (Smith & Whitehead, 2011).

In order to evaluate the expressive range of the previous work, the same authors proposed a method to analyse the generated content (Smith & Whitehead, 2010). One important aspect that is referred and should be retained is that it is not only important to examine the number of different levels that are generated and the time needed for the creation. It is also relevant to extract how different and varied the results are. Two main aspects were considered to measure expressivity: linearity and leniency. It was established that a level is linear if the path from the start to the end can be represented as a single line. Linearity is measured with a linear regression considering the central points of each platform. Leniency is a measure related to difficulty, where existing objects contribute to a final score. The goal of identifying leniency is to understand if the range of outputs can reach different user profiles.

In addition, this rhythmic approach tries to achieve fair and challenging results regarding gameplay, two core features of the aforementioned concept of *flow*. A methodology to keep the user in the state of *flow* is to control the task's difficulty, the concept of DDA that was previously explained in sub-section 4.3.2, where we also pointed the system *Polymorph* (Jennings-Teats et al., 2010a, 2010b). In *Polymorph*, the authors tackled the problem of DDA by creating statistical models for levels and players. Level segments are generated at runtime, using the rhythm-based approach that we have been describing, in which the new segments try to match the user's difficulty profile. The difficulty of the segments and the profile of the user are identified using machine learning principles.

As a final point about the subject of rhythm-based level generation, it is important to refer the mixed-initiative proposed in the system *Tanagra* (Smith, Whitehead, & Mateas, 2010a, 2010b,2011), a "design tool for 2D *platformer* level design, in which a human and a computer can work together to produce a level". The user's task is mainly to provide level rhythm and required geometric sections, while it is up to the computer the task of filling spaces between the defined sections. This final part is accomplished with an implementation based on constraint programming.

While the majority of the work on the field focuses essentially on the typical tactile-skill based levels, Nygren *et al.* (Nygren, Denzinger, Stephenson, & Aycock, 2011) directed their work to a different perspective, generating more diverse puzzle-based content, requiring the player to explore the level in order to solve it. Specifically, the authors presented three different types of gameplay that can be selected, namely:

- Combat, in which the level is strongly populated with opponents.
- Flow, in which the challenge is represented mainly by the level geometry.
- Puzzle, where different paths are created through the level, forcing the player to have an exploratory behaviour.

Their approach for level generation is decomposed mainly into two phases, starting with a preference elicitation process that is followed with the effective generation engine. The initial elicitation stage allows the user to select one of the previous gaming styles and to select the desired difficulty. The player's ability is assessed within an initial set of pre-authored levels. The level generation stage is decomposed into the following steps:

- Creation of the level structure, in which a graph is procedurally generated, representing the cells of that level and the links between them. The authors describe cells as "subdivisions of the level in which the style of content is locally consistent with regard to some measure", adopting the principles of Smith *et al.* (Smith et al., 2008). The graph generation was implemented using a feasible-infeasible two-population genetic algorithm (Kimbrough, Lu, Wood, & Wu, 2003). In order to identify good graph structures, a fitness function was implemented taking into account the average number of outgoing edges, the number of outgoing edges for the start node, the number of incoming edges of the end node and the length of the shortest path from the start to the end.
- Conversion of cells into a grid-based representation, in which each cell is assigned to a certain region in the two-dimensional space that represents the level. Restrictions are applied in this generation process in order to make the corresponding regions represent the adjacencies of the initial graph.
- Creation of individual cells using the available assigned space from the previous stage. In this final step, game components are randomly added or removed until the adjacencies of the

graph are completely represented within that cell. The process of addition and removal of entities is biased according to the selected gaming style.

Another example directed to puzzles was presented by Williams-King *et al.* (Williams-King, Denzinger, Aycock, & Stephenson, 2012). The authors presented a mechanism based on two main steps to generate puzzle platform levels for the game *KGoldrunner*. The first step consists in creating static level geometry, using genetic algorithms. In the end of this initial step, a graph is calculated in order to identify the main routes within the level. The second phase of the algorithm consists in using intelligent agents to play the level, each of them exploring different resolution plans that have been identified in the level graph. In a simplified manner, if a large number of those agents can finish the level, it means that it is too easy and, on the contrary, if only a small number or none of those agents can finish the level, then it is considered too difficult.

Still with the focus on the generation of adventure levels, Dormans and Bakkes (Dormans & Bakkes, 2011; Dormans, 2010) proposed a generating approach decomposing the process into two distinct phases, namely missions and spaces. For the generation of missions, the author presented a technique using generative grammars, applying it to the concept of graph, resulting in graph grammars, "a specialised form of generative grammars that does not produce strings but graphs consisting of edges and nodes". The space generation is accomplished recurring to shape grammars. To bridge the two generation phases, each rule of the implemented shape grammar refers to a terminal symbol in the mission grammar.

Moreover, Mawhorter and Mateas (Mawhorter & Mateas, 2010) presented a different approach to level generation. They introduced Occupancy Regulated Extension (ORE), an algorithm to create game spaces based on the composition of small pre-authored level parts designated as chunks. ORE starts with a partial representation of a level, initially filled only with a starting chunk containing an anchor to represent the effective starting position. The rest of the level is composed iteratively, matching the current partial level with the existing chunk set, resulting in new anchors representing potential avatar positions. Ultimately, the results are varied providing several distinct levels with different segments. However, playability is not ensured and possible adaptation to other contexts has not been approached. Even though the authors applied this idea to *Infinite Mario Bros.*, a game where scenarios are typically open spaces, the used concepts suggest additional potential in reduced spaces with more restricted character movements. The referred videogame *Spelunky* uses similar principles to generate this type of closed environments.

Additionally, Dahlskog *et al.* (Dahlskog, Togelius, & Nelson, 2014) presented an alternative approach for level generation based on *n*-grams. Levels of the game *Super Mario Bros.* are treated as left-to-right sequences of vertical level slices. The generation is then performed in a setting with some formal similarities to *n*-gram-based text and music generation. The applicability of the method is limited to the domain of two-dimensional games where the level progress is completely unidirectional. Still, the authors claim 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".

Furthermore, Snodgrass and Ontañón (Snodgrass & Ontañón, 2014a, 2014b) presented an approach for level generation for *Super Mario Bros.*, based on Markov chains. Starting with a set of existing levels, the system learns statistical patterns regarding the displacement of the cells. Specifically, the Markov chain learns the probability distribution of a given tile, based on the tiles that previously occur in the level, interpreting content top-down and from left to right. This is an interesting alternative to the existing generators, showing that Markov chains have potential for such process. However, the presented validation only ensures the creation of playable levels. To incorporate the generator into an actual game, additional rules to detect malformed structures must be considered.

The multiple works presented in this section show the interest of researching alternatives for the automatic generation of game levels, in particular for platform videogames. It is possible to observe that the earlier techniques provide an interesting foundation for such research, but left different aspects open for further improvements. We have verified that the most popular approaches were mainly directed to generate linear rhythmic structures following the principles of *flow*, without the inclusion of global level features. In section 7.3, we will tackle that issue and propose a generation algorithm that produces levels with an evaluation based on overall level features, such as the branching of the existing paths and the frequencies of the used blocks.

Moreover, good game design goes beyond those principles and the creation of a level typically comprises a set of different level design patterns. Considering the level design patterns that we have identified in subsection 3.3.3, it is possible to verify that the existing techniques, presented in this section, are mainly directed to the creation of *checkpoint* situations, and the additional level design patterns are not considered. We will address this issue with two main perspectives. First, in section 7.4, we will propose an adaptation algorithm that produces different types of situations based on the identified patterns. This technique works with a level created beforehand (humanly or by an automatic generation algorithm) and takes advantage of the existing structures to create the different types of situations. Secondly, we will propose a semi-automatic generation process, in section 7.7, where the designer defines a set of quests and the level is generated following those quests.

Furthermore, in this section, we have observed some existing approaches that generate content based on an initial set of levels. We will also propose a method to generate levels based on this idea, in section 7.5, where we include our graph representation and analysis in the process as an efficient method to ensure playability.

7.3. Mapping Design Heuristics into a Genetic Algorithm

The usage of Genetic Algorithms (GA) is a popular way to generate content procedurally. In general, stochastic solutions are a plausible way to generate levels because, on the one hand, they provide different results in different runs and, on the other hand, they offer an adequate sampling on all possible solutions without testing them all. In this section, we present an implementation of a level generator that has been developed in the scope of this work. Our approach consists in the inclusion of general design heuristics into a GA, in order to search for viable levels within those heuristics.

Primarily, before entering the details of our implementation, it is important to refer the main features that one should consider in such type of approach. A GA mimics real life evolution, in particular based on Darwin's theory of natural selection (Darwin, 1859). Briefly, this theory states that living beings that fit best their environment are more willing to survive and reproduce, hence their features are reinforced in forthcoming generations. Those principles are represented algorithmically to solve different types of problems, following the next guidelines:

- Possible solutions are represented as *individuals*, coded with certain data (genotype) that are manifested throughout some features (phenotype).
- To simulate the principle of fit to the environment, it is required that the individuals are evaluable, meaning that it has to be possible to quantify their quality. This value is defined in mathematical terms as a *fitness function*.
- Evolution over time is simulated through a set of operations, applied to the individuals with certain probabilities. It is common to implement a *mutation* operator that randomly changes the features of some individual and a *crossover* operator that mimics reproduction, merging two (or more) individuals into a new one, sharing features from their ancestors.

- A population is defined with a set of individuals. Several generations are simulated by iterating the individuals and applying the possible operators. Individuals are selected from one generation to the next, proportionally to their fitness, possibly with a random factor. Multiple variants have been considered, such as the use of multiple separated populations and the possibility of varying the size of those populations at different iterations. It is also possible to apply additional operators, such as plagues, occasionally discarding a considerable part of the individuals, among several others.

Our implementation was inspired and directed to the classic videogame *Prince of Persia*, because this game has a coarse-grained decomposition into cells, making it suitable for such approach.

7.3.1. Level Representation

For the level representation, we have adopted a direct mapping of the genotype into the phenotype, which means that the coded information represents the features directly. In this case, it consists in representing each cell individually within a two-dimensional array, using the level representation framework that we described previously in section 5.3. The main advantages of this approach are locality, as this allows small changes in a level, and representation capability, as this alternative covers all possible solutions of its domain.

Furthermore, we used a minimal subset of the game blocks in the generation process. The original *Prince of Persia* game presents a set of 31 different blocks to define the level structures. Regarding the genotype of the level, for generation purposes, we have considered only three different types: walls, empty blocks and floor planks. Some of the remaining blocks of the original set have essentially an aesthetic purpose, such as pillars, pillar parts, torches and tapestry elements, and can be considered as variants of the three core elements that we have defined. Using these variants directly in the evolution process generates unnecessary complexity and increases significantly the required time to generate a valid level. Previously, in section 5.4, we have explored the definition of the level blocks for this game, using our framework, and in the context of the block hierarchy, we have also established these three types of blocks as the main elements. This is a natural way to structure the existing blocks and it is applicable to most platform videogames. Moreover, other blocks from the initial set have strong impact in the interpretation of the generated structure, such as gates and trigger buttons to open or close those gates. Analysing a level considering gates and their respective triggering mechanism increases the computational complexity of the process, which in an evolutionary system that requires several analysis at each iteration makes the process slow. Some of the theorems presented by Forišek (Forišek, 2010) and Viglietta (Viglietta, 2012) show that the presence of this type of features increase the computational complexity of finding the best solution of a level from P to PSPACE-complete. Thus, it is preferable to include these entities in the level at a second stage of the processing, as a perfecting step.

7.3.2. Heuristics for Level Evaluation

The evolution mechanism in an evolutionary system requires an evaluation method to identify the best individuals, with impact in their continuity in further generations. In the implemented system, the fitness of the individuals was defined aiming to follow some design heuristics. Liapis *et al.* (Liapis, Yannakakis, & Togelius, 2013) identified three game design patterns for having high generality while being easily quantifiable: area control, exploration and balance. Our heuristics include principles that are aligned with those features, adapted to the context of platform videogames, and are explained next.

Path Structure

The level has to represent an adequate path with some diversity. On the one hand, it is important to have alternative routes to avoid excessive path linearity, which could result in single closed corridors. On the other hand, it is important to prevent excessive path branching, resulting in complex mazes. In order to identify if the structure of a level is more or less branched, one could perform an extensive analysis of the existing routes, for instance using a breadth-first algorithm, and observe the overlapping of the possible alternatives. However, this type of solution is computational expensive and presents an unnecessary level of detail concerning the existing routes. Regarding the heuristic, it is essentially important to have an approximation that quantifies the existence of alternative routes and optional sections. Our solution consists in creating a graph to represent the possible movements of the avatar, following the techniques presented previously in section 6.3.1, and simple agents explore that graph mimicking the possible movements of the game character. Initially, an agent is placed at the starting position. Every time an agent has multiple movement alternatives, it follows one of them, arbitrarily, and new agents are created to explore the remaining alternatives. These new agents keep the movement history of the original one. The existing agents are iterated one movement per cycle and they never move to a node that has already been explored by another agent. When an agent runs out of movement possibilities, it is destroyed. This approach can be seen as an implementation of a best-first search, with an undefined goal and considering that all graph edges have the same cost. The parallelism of the search with the concept of agents provides as estimation of the path branching. The higher the number of agents during the search, the higher the path branching. Therefore, the average number of agents along the iterations ($\bar{a}$) is compared to a reference desired value, which is established as a parameter (k_a). The fitness value for this first heuristics is obtained proportionally to the absolute value of the difference of the previous terms, truncated between zero and one, with the following expression:

$$F_{h1} = \max\left(0, 1 - K_p|\bar{a} - k_a|\right)$$

The value Kp is a parameter (mandatorily positive) that denotes a proportionality constant regarding the penalisation of the existing deviations. We have observed that a unitary value provides adequate results. In Figure 7.2, we exemplify the behaviour of the agents exploring one level.

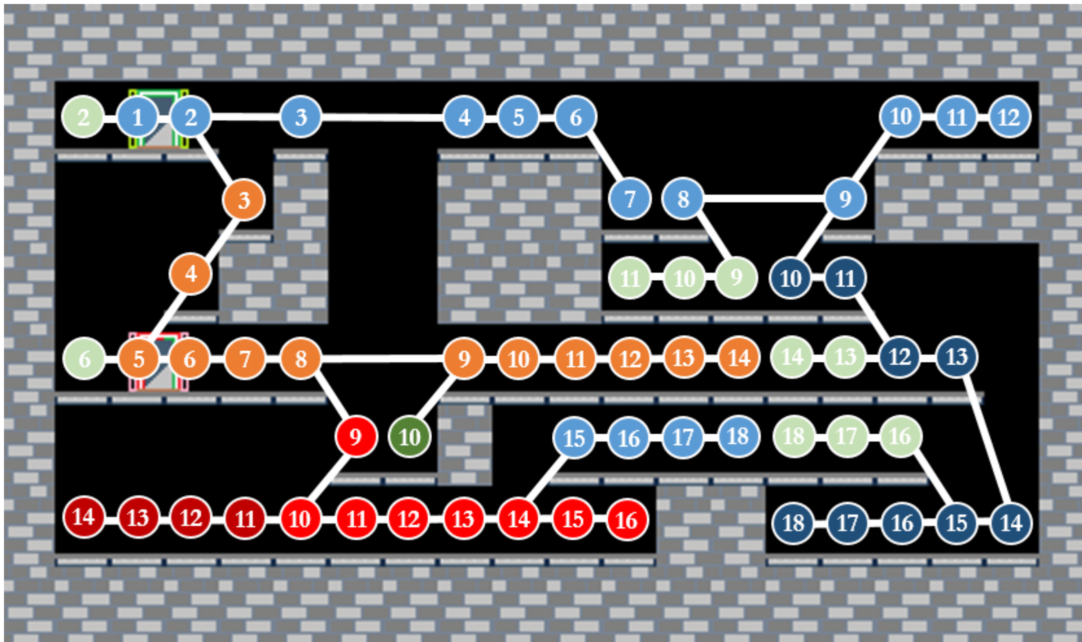


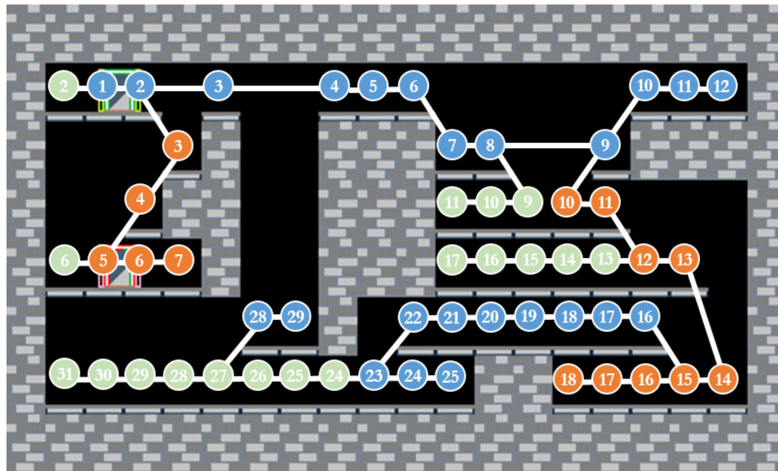
Figure 7.2 – Depiction of the level exploration performed by the agents.

Different agents are marked with circles of different colours and the numbers denote the corresponding iteration for the agent position. Accordingly, in the first iteration, one agent is placed at the door on the top left part of the level. As this agent has two alternative movements, it moves to the right and spawns another agent that moves to the left, resulting in the two agents labelled with the number 2. In the next step, the agent that followed the left alternative reaches a dead-end and it is destroyed. The other agent has once again two alternative movements, resulting in the two agents labelled with the number 3. The process continues until the agents explore the entire level and, in the end, we obtain the following number of agents at each iteration:

$$n = (1, 2, 2, 2, 2, 3, 3, 3, 4, 6, 6, 5, 5, 5, 3, 4, 4, 4)$$

Therefore, the corresponding average number of agents is 3.3. During our experiments, we have observed that, typically, this value should be between 1.5 and 2.5, depending on the type of structure that the designer wants. A value of 1.5 commonly denotes a level with a simple path with a few bifurcations, while a value of 2.5 normally represents a level with various bifurcations and multiple alternative paths. Accordingly, the previous example level is too branched. In Figure 7.3, we present two levels similar to the previous one, where it is possible to observe that slight modifications can have impact over the level structure.

Example A



Example B

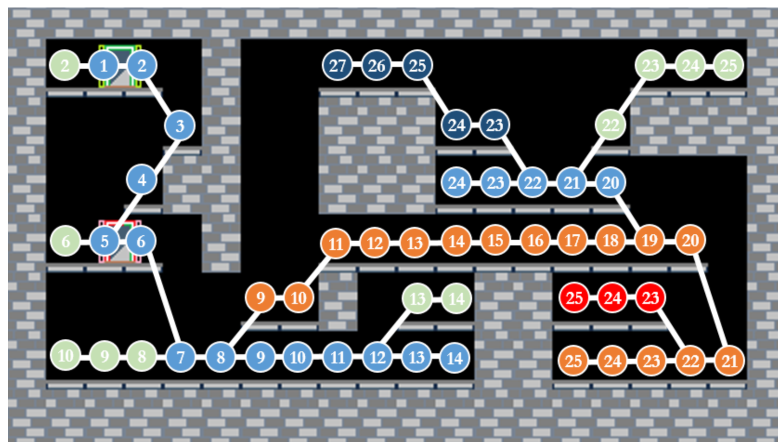


Figure 7.3 – Examples depicting the level exploration performed by the agents.

The presented examples result in the following number of agents at each iteration, with their respective mean value:

$$n_A = (1,2,2,2,2,3,2,1,2,3,3,2,2,2,2,3,3,2,1,1,1,1,1,2,2,1,1,2,2,1,1);\ \overline{n_A} = 1.8$$

$$n_B = (1,2,1,1,1,2,1,2,3,3,2,2,3,4,1,1,1,1,1,2,2,3,5,5,4,1,1);\ \overline{n_B} = 2.1$$

The obtained means in these examples are lower than the initial case and are inside the range that we have proposed. Example A presents lower branch in relation to example B, especially because the existing bifurcations tend to present a dead-end nearby. By that reason, the algorithm never had more than three active agents exploring the level at the same time. Example B is considered more branched mainly because the last part of the level, where multiple alternatives are presented at the same time, resulting in five active agents exploring the level simultaneously.

Placement of the Start and the End Positions

The placement of the start and the ending cells has to ensure, at first, that the level is valid and, secondly, that an appropriate challenge was created, with an acceptable dimension in relation to the whole level. For this purpose, we measure the length of the shortest path from the start to the end of the level, which we can refer as the optimal solution, measured in player's actions. We also measure the maximum number of actions performed by one single agent or its descendants, as described in the former heuristic. The placement of the start and the end positions should result in a level where the length of the optimal solution (l_s) is a high percentage of the maximum identified length (l_{max}). In Figure 7.4, we present an example of two different placements for one level, which shows the impact of such placements. In example A, the start and the end positions of the level are close to each other. The end of the level is reached in five iterations, which is a small value in relation to the longest path covered by an agent, with a length of 31. The length of the solution is 16% of the length of the longest path, which means that a large part of the level is not likely to be used or explored, because the player can finish the level right away. On the right part, the start and the end positions of the level are away from each other. An agent finds the end of the level at iteration number 25, and the maximum length covered by an agent is 27. The length of the solution is about 93% of the length of the longest path. These positions cause a greater part of the level to be actually necessary, and encourage a greater usage of the existing space.

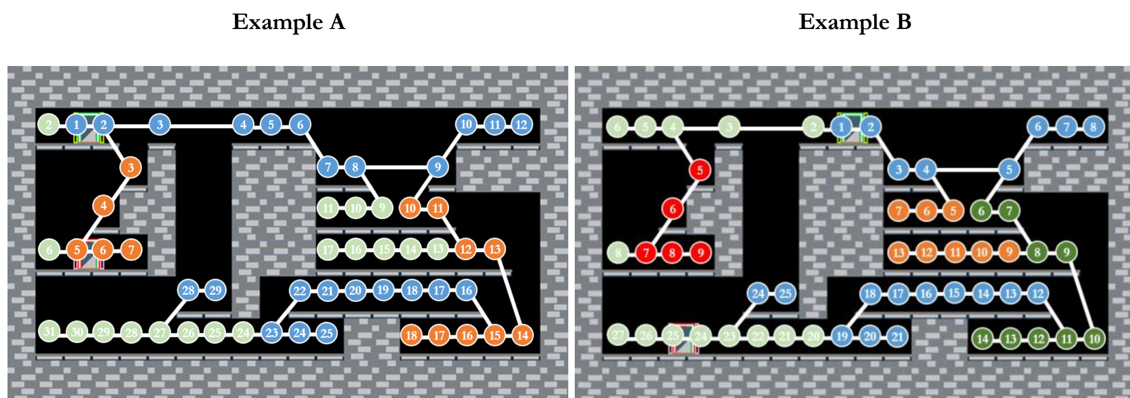


Figure 7.4 – Impact of the placement of the start and the end blocks within the levels regarding the existing paths and their length.

In order to calculate the fitness value for this heuristic, a reference percentage threshold is defined as a parameter T_p. If the length of the level solution in relation to the longest path exceeds the referred threshold, that value is considered high enough, and the fitness value for this component is one.

Otherwise, the fitness value ranges from zero to one linearly, for the domain between zero and the established percentage. The calculation can be summarised with the following expression:

$$F_{h2} = \min\left(\frac{l_s}{l_{max}} \cdot \frac{1}{T_p}, 1\right)$$

We have observed that, typically, a value for T_p over 80% provides adequate placements of the start and the end positions.

Contextual Cell Analysis

Each cell has a particular meaning regarding its neighbours. The system defines good and bad cells as they make sense within its neighbourhood. The following rules have been established in order to verify if a cell is good or bad.

- A wall cell is always considered valid.
- A floor cell is only valid if it is part of any of the possible paths, including routes to a dead-end.
- An empty cell is valid if it is used in a path (for instance, to create a gap to jump over) or if it has aesthetic purposes. For the last, we defined that an empty cell has aesthetic purpose recursively if it has a valid empty cell in the neighbourhood. This specific aspect allows the system to construct levels with open rooms instead of only closed corridors.

In Figure 7.5, we present a level obtained during the generation process, where it is possible to observe some of the cells marked as bad. The floor cells on the top right of the level are marked as bad because they are inaccessible. The marked empty cell is considered bad because it is not part of an existing path and it does not have an adjacent empty cell to be considered aesthetic.

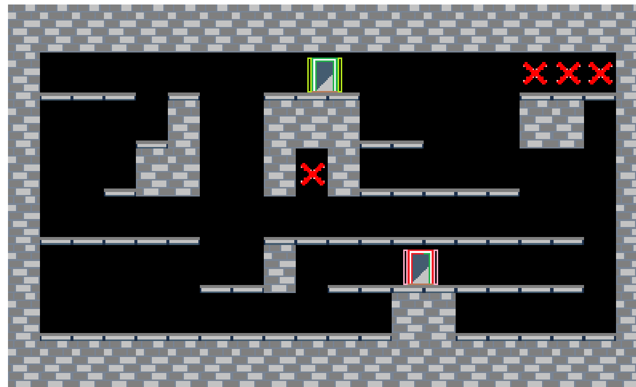


Figure 7.5 – A level obtained during the computation of the genetic algorithm, where cells marked as bad are identified with a cross.

The fitness value of this specific component is obtained by the percentage of the number of good cells (n_g) in relation to the total number of cells of the level (n_t), with the following expression:

$$F_{h3} = \frac{n_g}{n_t}$$

Aesthetic Balance

To keep the visual balance of the generated level, the usage of each particular block should be adequate. As we have stated on the previous heuristic, a wall cell is always valid, which would direct the evolution process to use excessively this type of blocks. Therefore, the frequencies of each block (f_b)

within the level cells should match a set of reference values, established as parameters (Kf_b). The deviation of a level from this reference (d_l) is then measured as the sum of the absolute differences for each block, with the following expression:

$$d_l = \sum_b |Kf_b - f_b|$$

The level has the maximum possible deviation (d_{max}) when it is totally filled with the block that has the lowest value of Kf_b, given by the following expression:

$$d_{max} = 2\left(\sum_b Kf_b - \min(Kf_b)\right)$$

Below a deviation threshold (T_d), established as a parameter, this component has its maximum value of one. From this point, the function decreases linearly until it reaches the value of zero, obtained when the level has the maximum possible deviation. Hence, the value of this heuristic is obtained by the following expression:

$$F_{h4} = \min\left(\left(1 - \frac{d_l}{d_{max}}\right) . \frac{1}{1 - T_d}, 1\right)$$

In practice, we have verified that configuring this feature to allow preponderance of specific elements is a way to direct the output to a certain level style. This is a way of providing some control to the designer within the automatic generation process. In Figure 7.6, we present three examples, each one obtained with a specific parametrisation, biasing the generation to use a certain block more often that the others.

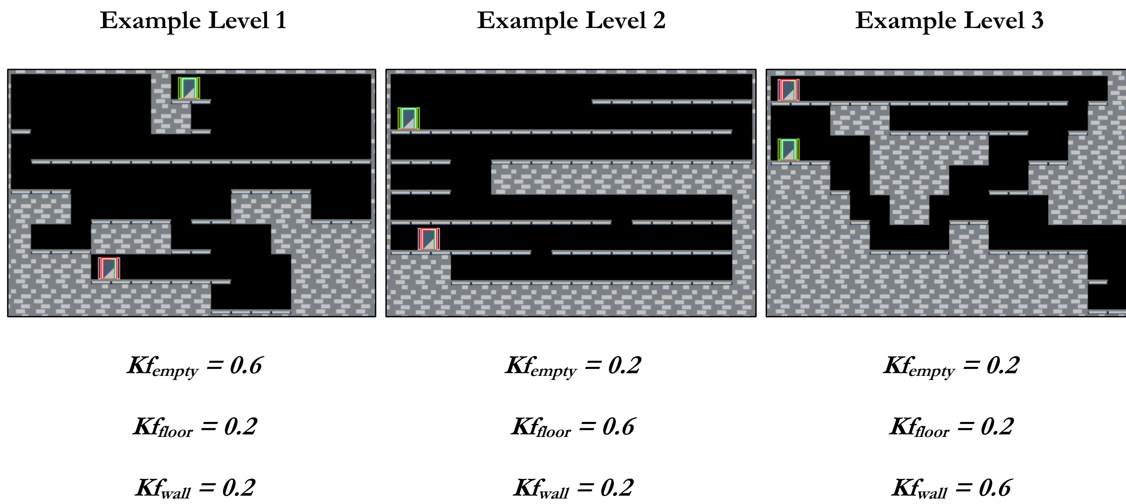


Figure 7.6 – Examples of generated levels using different parameters regarding aesthetic balance.

It is possible to observe that the frequencies of each block vary according to the parametrisation. In the first case, we have defined a higher desired percentage of empty cells, resulting in an output with wider rooms and less corridors. In the second case, we biased the generation to use mainly floor cells, which resulted in long sequences of this type of platform. The first heuristic forces the generation process to avoid excessive branching, so this parametrisation results in a longer path from the beginning to the end of the level, and the player has to weave across the different corridors in order to finish it. In the third example, the output presents a level with a higher frequency of wall blocks in

result of the respective parametrisation. In this case, the algorithm tends to minimise the existence of floor and empty blocks. These blocks are used barely to allow the creation of a valid level from the start to the end of the level, meeting the criteria of the other heuristics, resulting in close space structures.

Heuristic Combination

The defined set of heuristics (h_i) provides partial scores of the level quality, regarding different aspects. As we have observed in the presented expressions, these aspects have been measured in a scale from zero to one. The final fitness value of the individuals is obtained summing these components, pondered with specific weights (w_i), with the following expression:

$$F = \sum_i w_i h_i$$

With a set of weights whose sum is one, the expression is a pondered mean with a scale likewise from zero to one. Roughly, a final value of zero represents the worse possible case and, on the contrary, levels with a score of one are theoretically perfect. In our tests, the algorithm performed best using equal weights for the heuristics, as all of them play an important role in the calculation. With different values, the algorithm tended to spend more time reaching higher fitness values and to be more permissive to represent undesired situations.

7.3.3. Operators

In an evolutionary system, variety lies in the quality of the used operators. On the one hand, these operators should be able to perform changes in the existing data to make them different enough to skip local maxima and, on the other hand, these changes should keep good information from past iterations. At each new generation, mutation occurs and individuals are crossed with each other in a process that we will describe next.

Mutation

Mutation occurs with a certain probability and can be applied in different forms. As stated, it is important that mutations are able to make an individual diverge sufficiently to skip local maxima. In our case, we considered initially the smallest possible mutation as being the change of one particular cell in the grid to another value. The algorithm would randomly pick one cell from the level and set it to a valid random value. In our initial trials, we observed that the change of a single cell represents a minor variation and does not provide enough divergence, thus the implemented mutation operator consists of more than one change at a time. The number of changes in each mutation can be tuned, as it is a system parameter.

We have implemented two types of mutation, defined as random and selective mutation. Random mutation simply changes some of the cells in a level, arbitrarily. In selective mutation, we consider that some cells are more suitable to be changed. Isolated floor cells are not aesthetic so they are more proper to be changed. Besides, cells that are not in the main path and are not accessible by any way are more likely to be mutated to a wall block. These corrections allow a faster evolution of the system, namely in the features regarding the heuristic of contextual cell analysis, presented in the previous subsection.

Crossover

This operator was implemented to cross elements in pairs. Crossing over more than two elements was tested without relevant improvement on the final results, so the descriptions will focus on the pair crossing mechanism. Due to the level structure, based on cells, a simple crossover mechanism

could consist in constructing each new individual by taking random cells from another two individuals. However, cells by themselves do not represent much information and should be considered in relation to the whole level or at least to its neighbourhood. Therefore, it was decided to take mainly into account the more relevant paths that exist in each individual to be crossed, rather than only the isolated cells.

When two levels are crossed, the main path of the first is kept intact, the main path of the second is also kept intact as long as it does not contradict the first one and, finally, other cells are chosen randomly from one or another individual. A visual representation of the crossover mechanism is provided in Figure 7.7. We start by presenting two different levels in the first row and their corresponding path on the second row. The third row presents the overlap of both paths. Cells that correspond to paths in both levels are highlighted and the assigned value corresponds to the first individual. In the fourth row, we added the cells that have the same content in both levels. The final row presents a possible result by filling the remaining cells taking the values randomly from the first or the second individual. This crossover operator performed better than the simple random selection of cells, which had a very similar behaviour to the mutation operator.

7.3.4. Closing Remarks

The algorithm that was presented in this section is part of our primal studies in the automatic generation of platform game levels. The presented approach unveiled some advantages concerning level variety, which lead to further studies. For instance, in the generated levels, the solution is not straight and sometimes not even unique, which allows usage in other variants of platform gaming. The creation of optional areas revealed the importance of defining a non-linear gameplay. Smith and Whitehead (Smith & Whitehead, 2010) pointed the importance of having generators capable of accomplishing different levels of linearity. In this case, we point that linearity is not only a measure regarding the static structure of a level but also a feature that should consider the routes that occur in that same static geometry. This emphasis on path structures and similar features was part of the inspiration of the aforementioned studies in the subject of graphs, presented in chapter 6.

Regarding performance, the implicit rules represented by the calculations in the fitness function make the process converge to a good level in a matter of seconds. In our latest tests, we ran the algorithm one hundred times in a laptop with an *Intel Core i7* processor at 2.3 MHz, and the generation process achieved a level with a fitness value of at least 0.95 in an average time of 8.1 seconds. It is important to refer that the current implementation does not take advantage of multicore processors, and the graph creation uses a naïve pattern matching implementation, as stated in section 6.3.1. Such type of optimisations can reduce the generation times. In addition, the initial iterations of the evolution process are typically faster, because the existing levels have less complex paths to analyse. In this early stage, the fitness evolves faster than in subsequent iterations, normally taking about 2 seconds to converge to a valid level with minor errors, with a fitness value around 0.9. The proceeding iterations perfect the level structure removing those minor errors, resulting in fitness values over 0.95. In Figure 7.8, we represent graphically the described evolution, obtained from an arbitrary run of the algorithm.

Finally, the initial implementation of this approach included a post-processing step that complements the generated level structure with additional gaming content. This content comprises aesthetic elements, such as torches and windows, as well as other gameplay entities, such as enemies and traps. In the next section, we will present an adaptation algorithm that performs this type of post-processing, taking into account the existing structure to create additional types of patterns with buttons and gates, and performing difficulty-based adjustments.

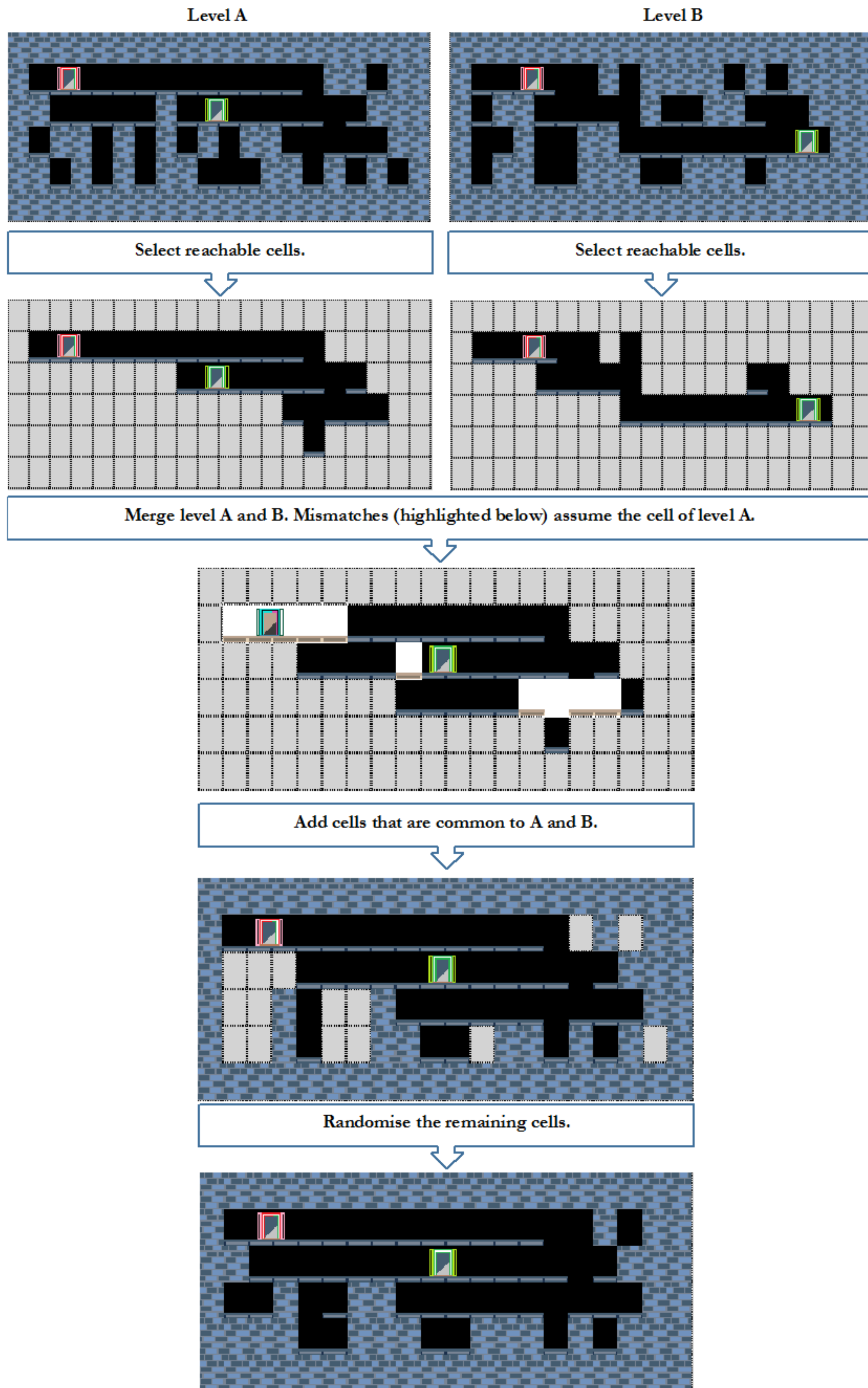


Figure 7.7 – Example of the implemented crossover mechanism.

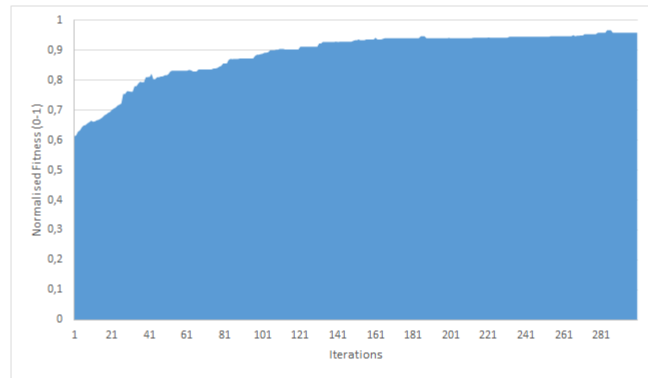


Figure 7.8 – Evolution of the levels' average fitness value.

7.4. Graph-Based Adaptation Algorithm

The construction of a level is a creative process that is performed at multiple stages. Initially, the level is thought as a global structure with certain features and, as the process evolves, more detailed aspects are created and the content is increasingly perfected. With that in mind, we implemented an adaptation algorithm that takes a level draft as input, which might be a sketch created by a designer or have been generated by a PCG algorithm, as the one presented in the previous section, and performs content adjustments. This section covers the details of such algorithm.

7.4.1. Requisites

In its basis, our adaptation algorithm works using graphs that represent a respective level structure. As we have presented in section 6.3, those graphs can be obtained in different manners. The algorithm also uses the processing mechanism and the node classification scheme that we have proposed in section 6.4. In addition, the level completion algorithm adjusts the level based on estimated difficulty and similar related aspects. The algorithm is agnostic regarding the used difficulty metric, so the exact concept of difficulty may remain undetailed and be implemented as the game designer considers best. The key idea that has to be present is that difficulty arises from the base geometry, such as jumps over gaps, from gaming entities, such as enemies and traps, and from the path structure, which forces the user to accomplish additional jumps and to encounter further gaming entities. By design, the algorithm has little control over jumps, as adding or removing gaps require graph recalculations. Hence, it has a higher focus on the control of additional gaming entities and the definition of composed challenges.

Difficulty has been characterised and estimated in distinct ways, as we have presented in chapter 4. For the purpose of this algorithm, the requisites are the following:

- A higher difficulty value means that the level is more difficult than another with a lower value.
- Every level segment can be analysed individually to produce a difficulty value.
- A succession of analysed sections with certain difficulty values produces a final difficulty value for that succession. For instance, a sequence of independent challenges with a certain probability of failure results on the multiplication of such probabilities.

When the path section presents two or more alternatives, the difficulty value that is considered is the lowest of the existing alternatives, as it is plausible to assume that the user picks the easiest possible solution. However, exploratory behaviour can be reinforced for this situation by establishing another decomposition for parallel challenges, such as a pondered mean using the difficulty values of each alternative.

7.4.2. Implemented Adaptations

Types of Adaption

The algorithm is able to apply the following adjustments to the base level:

- **Change of the difficulty of a segment**, which is done by adding or removing an opponent or a trap, or making a gap larger or smaller. This is achieved with a pattern matching rule set. The original level graph remains intact, while a parallel data structure stores the added entities, as well as the changes that have been performed. Two different rule sets are defined, one containing the rules to make the level easier and another one containing the rules to make the level harder. In Figure 7.9, we present examples of such cases. On the left part, we present two rules for the game *Prince of Persia*, the first one to make the level harder (by adding a chopper trap) and the second one to make the level easier (by adding a potion). On the right part, we present similar cases applied to game *Infinite Mario Bros*. Adding an enemy makes the level harder and replacing a brick with a power-up makes the level easier. In Figure 7.10, we present an example of possible gap adaptations in *Infinite Mario Bros*. It is possible to define the type of block that does the filling in order to reduce the gap size.

- **Detour creation**, which consists of identifying a path to a dead-end from the main path, using the vertex classification described in subsection 6.4.2, followed by adding a certain item in the path to the dead-end and making it required on the main path. Again, a pattern matching mechanism is used, in order to search for a certain group of cells. In this case, a set of replacement patterns is defined, consisting of pairs of rules. One is applied in the path to the dead-end and the other one is applied in the mandatory part of the section. Figure 7.11 presents an example of such pair of rules for the game *Prince of Persia*. The left part shows that a button can be included in a section of floor planks regarding a path to a dead-end section, associated to a gate that should be created in the main path according to the pattern presented on the right.

- **Creation of cooperative two-player game situations**, consisting in identifying two parallel alternatives for one particular section and using the prior two principles to adjust each alternative individually for each player. Besides, the game should contain a method to prevent both players from following the easiest alternative. Figure 7.12 contains an example of this type of situation, considering a hypothetic multi-player version of *Prince of Persia*. Considering two players coming from the left side of the level, each one will have to take a different route to open the gate to the other player, in a path of no return. Thus, from that point they will be in different sections of the level.

- **Bonus entity addition**, which consists in adding collectibles or minor power-ups on certain dead-ends, creating secondary goals for the player. Once again, this type of change is accomplished using a pattern search and replace mechanism.

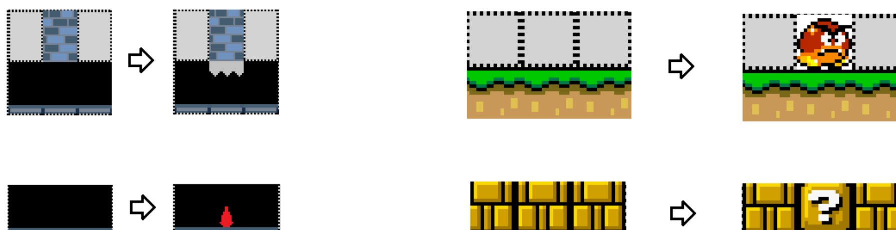


Figure 7.9 – Examples of rules for level adaptation. The two rules on the left apply to the game *Prince of Persia* and the two rules on the right apply to *Infinite Mario Bros*. The two top rules intend to increase the level difficulty and the two bottom rules intend to decrease the level difficulty.

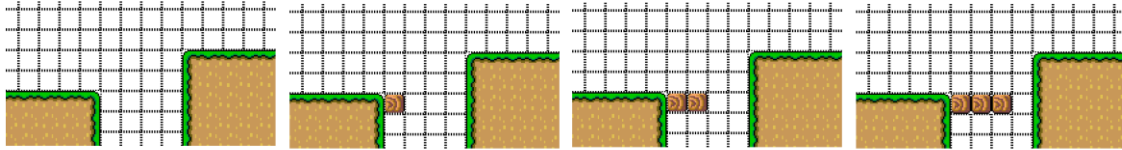


Figure 7.10 – Example of a gap in *Infinite Mario Bros.* and the possible level adaptation to reduce the gap size.

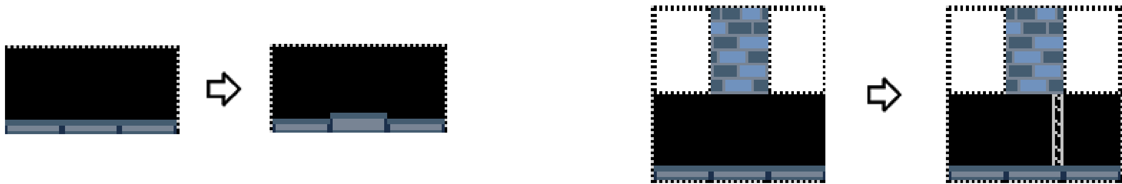


Figure 7.11 – Example of a detour adaptation rule in *Prince of Persia*. The rule on the left is applicable in the route to the dead-end and the rule on the right is applicable in the main path.

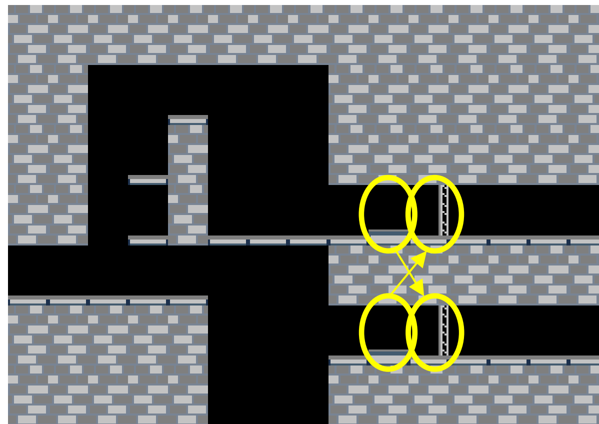


Figure 7.12 – Example of a situation that forces two players to follow distinct paths (the arrows denote the associations between trigger buttons and their respective gates, both marked with ellipses).

Analysed Features

The previous changes occur based on probabilistic coefficients, established by the system after analysing the following values:

- Path size from the start to the end of the level.
- Total desired difficulty value (whole level).
- Mean desired difficulty value (per segment).
- Estimator for the player's state at each vertex of the graph, detecting periods of possible boredom, *flow* and frustration. For this purpose, we have used an estimator based on an anxiety function. Each challenge along the path increases the anxiety value and trivial transitions reduce that same value.

Probabilistic Coefficients

The algorithm works iteratively in multiple adjustment passages, with the level difficulty being analysed for each passage. The process stops when it reaches the desired value or the maximum possible number of iterations. At the end of each step, the referred features are analysed and the following probabilistic coefficients are adjusted for each path section:

- Probability of adding/removing a difficulty item;
- Probability of adding/removing a bonus item;
- Tendency to adapt gaps; and
- Probability of creating/removing a detour.

Adaptation Algorithm
Moreover, the following situations and respective resolutions were identified:

- If the level dimension is lower than desired, the detour creation probability is increased.
- If the level dimension is higher than desired, the probability of removing previously created detours is increased.
- If the single player gameplay is too easy or the level is too easy for both user profiles in the multiplayer case, the following changes are performed:
  - Increase the probability of adding difficulty elements along the path;
  - Increase the probability of removing bonuses along the path; and
  - Make gap adaptation more willing to perform adjustments to make the level harder.
- If the single player gameplay is too hard or the level is too hard for both user profiles in the multiplayer case, the following changes are performed:
  - Reduce the probability of adding difficulty elements along the path;
  - Increase the probability of adding bonus entities regarding gameplay; and
  - Make gap adaptation more willing to perform adjustments to make the level easier.
- If during gameplay, frustration or boredom occurs, the system detects the section with the undesired situation and applies the overall correction case, setting independent probabilistic coefficients from the beginning of the level to that section.
- If the difficulty analysis using the player with the highest skills considers the level too easy, the overall correction case for easiness is applied, specifically to the particular sections that are exclusive to that player.
- If the difficulty analysis using the player with lowest skills considers the level too hard, the overall correction case for hardness is applied, specifically to the particular sections that are exclusive to that player.

The next subsection presents some examples that show how a simple level can be tweaked by the system using these premises.

7.4.3. Examples and Results

Example of Adaptations in *Prince of Persia*
As a first example, we will resume the case presented in subsection 6.4.2, where we have observed how the vertex tagging mechanism works, using the game *Prince of Persia*. We have presented the vertex classifications, identified a set of routes to dead-ends, and presented a tree that depicts the possible paths from the beginning to the end of the level. In Figure 7.13, we present that same level after the application of our adaptation algorithm.

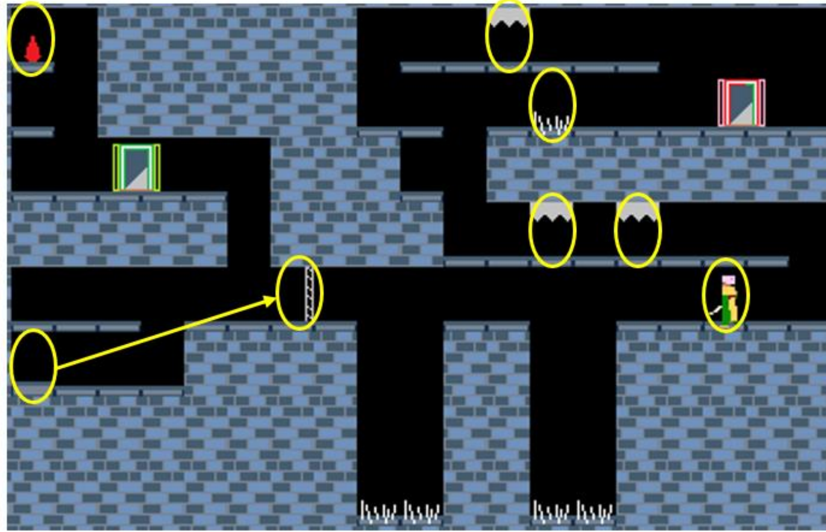


Figure 7.13 – Example of a tuned level for the game *Prince of Persia* (changes are marked with ellipses and arrows denote the associations between trigger buttons and the respective gates).

In order to use the available space to create a longer level, the algorithm added a gate in association with a respective button to open it. Recalling the pattern used to create gates, presented back in Figure 7.9, gates can only be added replacing a floor block with a wall cell over it. In the segment where this change occurred, the only alternative for the placement of this gate was the cell immediately on the right. In the remaining of the level, the algorithm added guards, choppers and spikes to increase the overall difficulty of the level. A potion was also added in the beginning of the level.

Example Integrating the Vertex Classification and the Adaption Algorithm with *Prince of Persia*

Now, we will exemplify with more detail the adaptation process with the complete process, from the graph classification to the calculation of the possible modifications. We have manually created the level structure represented in the background of Figure 7.14. The level does not contain enemies, gates, switches or any traps. These are the ideal conditions to apply the possible modifications. The starting point of the level is the door on the left side and the level ends when the character reaches the door on the right side. The system extracted and compressed the level graph presented as an overlay on that same image. Nodes have been named according to their coordinates on the grid, also marked in the image for convenience.

Using the graph classification that we have presented in subsection 6.4.2, the system classifies the vertices as follows:

- **Mandatory**: C3_4, C4_6, C5_11, C7_11, C9_10, C10_4, C14_4, C14_8, C21_4.
- **Optional**: C5_4, C7_3.
- **Dead-end**: C2_2, C2_9, C2_11, C3_6, C10_10, C10_11, C13_3, C19_6, C21_8.
- **Unreachable**: C15_11, C17_11.
- **Path to dead-end**: C3_11, C19_8.

Furthermore, routes to the existing dead ends are calculated. In this case, the system identifies the following possible routes:

- C4_6 → C3_6.

- C5_11 → C3_11 → C2_9.
- C5_11 → C3_11 → C2_11.
- C7_3 → C2_2.
- C7_11 → C10_11.
- C9_10 → C10_10.
- C14_4 → C13_3.
- C14_8 → C19_8 → C21_8.
- C14_8 → C19_8 → C19_6.

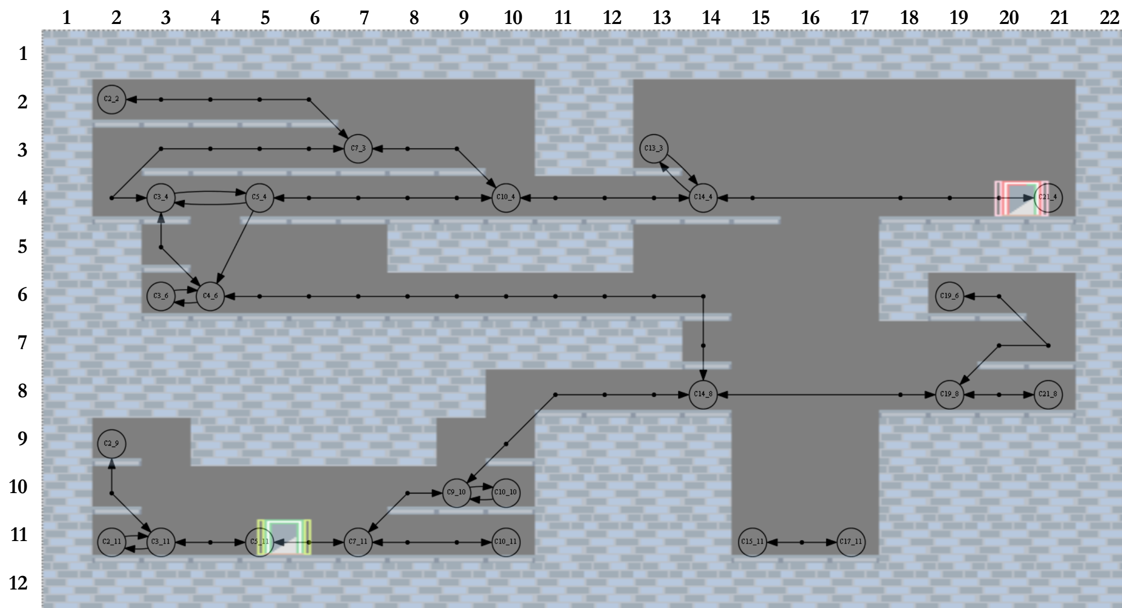


Figure 7.14 – Graph generated from the example level.

Moreover, the tree presented in Figure 7.15 was calculated, which describes the overall level structure, specifically the decomposition into alternatives of the main goal of moving from the initial point (cell *C3_3*) to the final one (cell *C17_2*), along with the length and estimated difficulty for each segment.

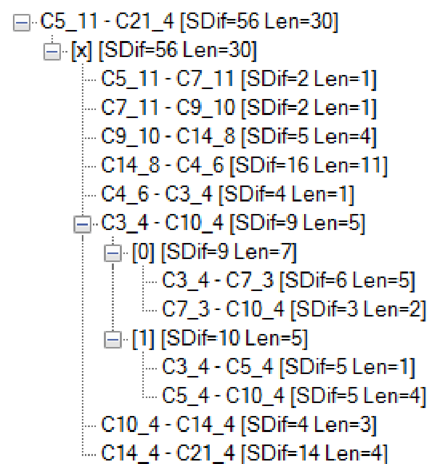


Figure 7.15 – Calculated tree with all possible paths from the beginning to the end of the level. The value *SDif* represents the estimated difficulty and *Len* refers to the segment length in cells.

Considering a simple approach for estimating difficulty implemented as the sum of accumulated challenges, similar to the metric used in the work of Smith *et al.* (Smith & Whitehead, 2011), a value of 56 was obtained and the solution length was computed as 30 cell movements among segments. As a test, we defined the desired difficulty value of 150 with a 10% error margin, therefore the algorithm was expected to increase the existing difficulty. In the same manner, we defined the desired length to be 40 movements. An arbitrary run of the algorithm presented the result (printed to console) showed in Figure 7.16. The changes were then applied to the level, generating the content showed in Figure 7.17.

```
Add spikes @ C7_11 (Difficulty + 10)
Create Bonus Detour. Add potion @ C2_9 ()
Add spikes @ C9_10 (Difficulty + 10)
Create Detour. Add button @ C10_11 and gate @ C8_10 (Length + 3)
Add enemy @ C12_8 (Difficulty + 15)
Create Detour. Add button @ C10_10 and gate @ C11_8 (Length + 1)
Add enemy @ C7_6 (Difficulty + 15)
Create Detour. Add button @ C19_6 and gate @ C12_6 (Length + 5, Difficulty + 20)
Add enemy @ C13_4 (Difficulty + 15)
Current difficulty: 121.
Current length: 39
Create Detour. Add button @ C2_2 and gate @ C11_4 (Length + 5)
Add spikes @ C19_4 (Difficulty + 10)
Add chopper @ C11_6 (Difficulty + 20)
Current difficulty: 151.
Current length: 44
```

Figure 7.16 – Example of a set of computed modifications.

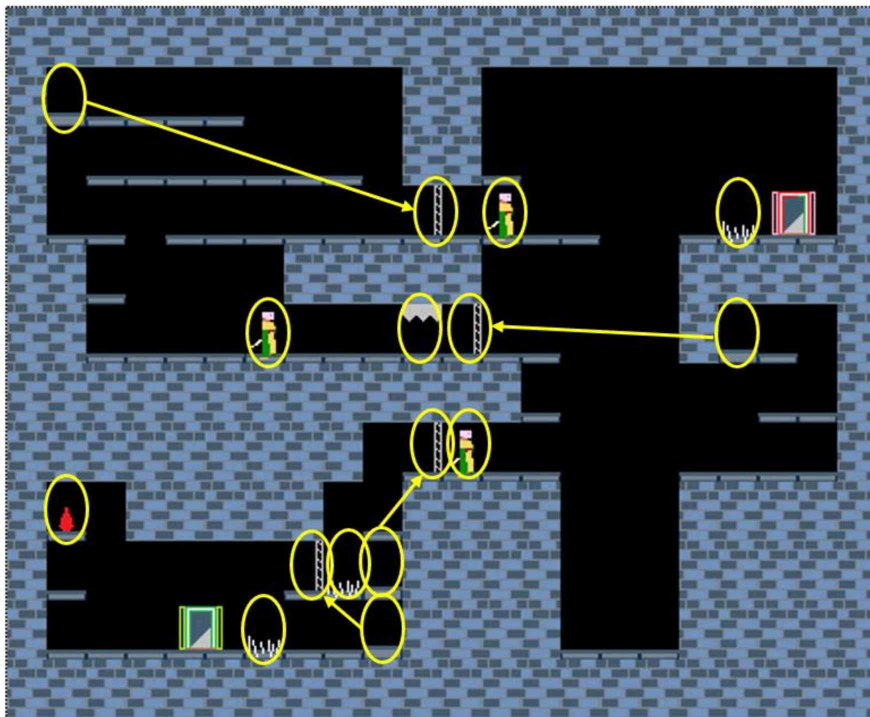


Figure 7.17 – Example of a tuned level for the game *Prince of Persia* (changes are marked with ellipses and arrows denote the associations between trigger buttons and the respective gates).

It is possible to verify that, in this case, two iterations were required to accomplish the desired results. The final value of difficulty is 151 and the total length resulted in 44 cell movements. In the end, four

gates were created, as well as the respective button triggers to open them. Those modifications had their main effect in the length of the level as only the button that was added in the rightmost part of the level forces an additional gap to cross, thus increasing the generated difficulty. This aspect was controlled with more emphasis on the addition of gaming entities, namely the addition of three guards, three spike traps and one chopper traps. Finally, one potion was added as an optional item.

Additional Examples with *Infinite Tux* and *XRick*

Besides *Prince of Persia*, we have also tested this approach with other titles, namely *Infinite Tux* and *XRick*. In the first, the detour principle is unlikely to be applied as the game does not have triggering events or object gathering. Nevertheless, the algorithm is able to adapt level segments with difficulty adjustments and by adding bonus content. Coins are willing to appear as bonuses on dead-ends, power-ups replace brick platforms and the number of enemies vary to match a desired value for the difficulty. In Figure 7.18, we show a sample of a randomly created level that has been perfected using our algorithm, matching a difficulty value defined by the user. The game *XRick* has strong emphasis on triggers, where the detour principle was also applied with success. In Figure 7.19, it is possible to observe a level that was created with that in mind. The main structure was manually created with two obvious dead-ends, accessible with the ladders. The system automatically added the two guards at the bottom, the bonus sphinx on the left dead-end and a trigger on the right dead-end, marked with a stick, which removes the spikes at the bottom, allowing the character to go through in the path from the left entry to the right exit.

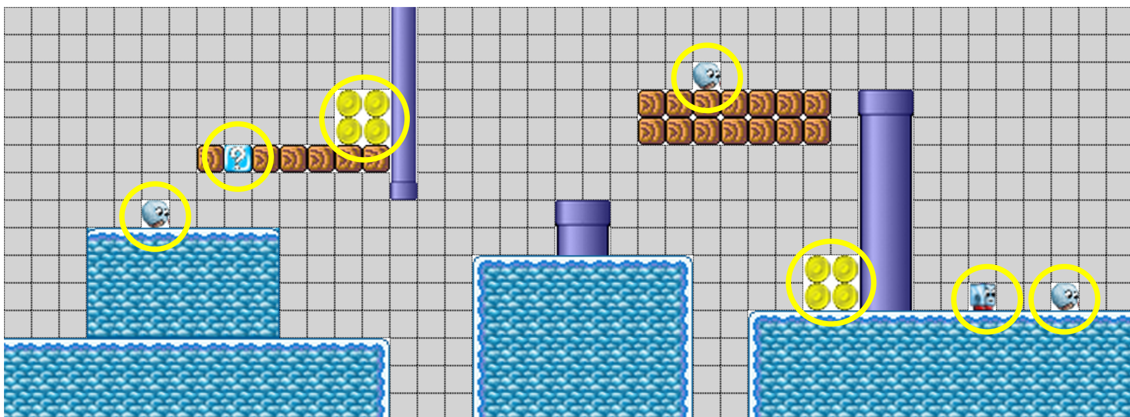


Figure 7.18 – Example of a tuned level for our prototype *Tux Likes You* (changes are marked with ellipses).

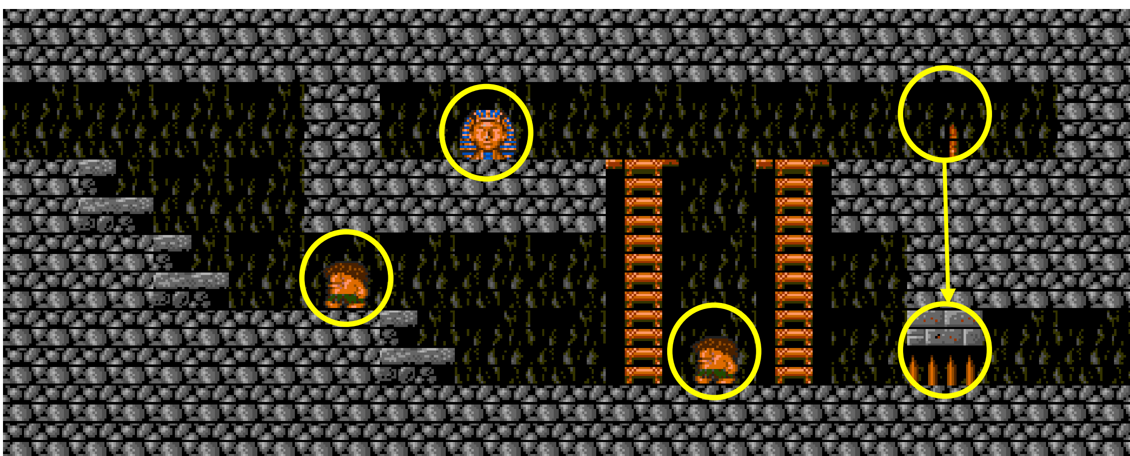


Figure 7.19 – Example of a tuned level for the game *XRick* (changes are marked with ellipses).

7.4.4. Closing Remarks

In this section, we have observed our level adaptation algorithm, a technique that, considering a roughly defined platform level and a corresponding graph, finalises the level by adding optional content and adjusting difficulty. This approach confirms the potential of the graph studies that we have presented in chapter 6, showing that knowing additional semantic concepts about the level structure brings interesting new features to level generation capabilities. In addition, filling a level structure with extra content and adapting paths to force some particular actions produces improvements in content richness to the topic of level generation. This approach brings context to the actions that must be performed in order to capture the user interest and to create less linear gameplay, as seen in more contemporary games.

Naturally, the type of adjustments that we have referred is mostly suitable for games that have somehow in their mechanics the principles of gathering certain objects or triggering some events to unlock passages. For this reason, we directed several examples to the game *Prince of Persia*, where we could see how our algorithm included new content. However, we have also applied successfully the technique in *XRick* in similar situations and in *Infinite Tux* to add enemies and coins.

Finally, we have proposed some adaptations based on multiplayer cooperative gameplay in a shared environment that could not have been tested in practice, as the studied titles do not contain such gaming mechanism. The considered games and most classic platform games are only single player, so this particular concept is still theoretical. Some experiments were done assuming possible versions of *Prince of Persia* and *Infinite Mario Bros.* supporting two players and the output is coherent with those premises. For instance, given the level structure presented in Figure 7.20, an arbitrary appliance of the algorithm resulted in the adaptations marked with ellipses, where we have highlighted the two segments for different players. Segment A was adapted to follow the profile of a player with low skills, while segment B considered the profile of an expert gamer. Therefore, this last segment presents various traps and enemies, whereas the other contains only one enemy to avoid the state of boredom. After the two parallel segments, the two players follow a common path and the algorithm added a balanced quantity of traps and enemies, preventing boredom to the more skilled player and frustration of the less skilled one.

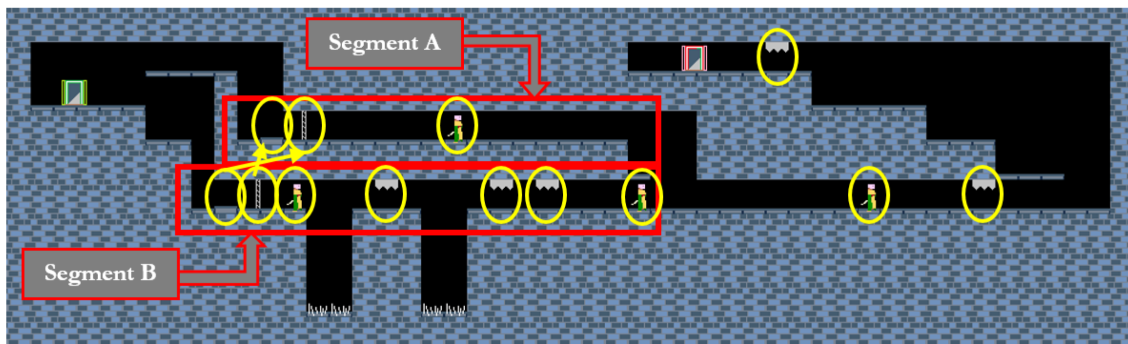


Figure 7.20 – Example of a level with two independent segments, adjusted individually for two players of different skills.

7.5. Chunk Overlap Extractor and Generator

Our third different approach for level generation is suitable for the creation of basic open scenarios, such as those in *Infinite Mario Bros.* and *Infinite Tux*. This composition algorithm is able to build levels based on a set of previously created levels. It is appropriate for the generation of levels where the gameplay consists mostly in side-scrolling action, which is the case of the aforementioned games.

Typically, a small set of initial levels is enough for the algorithm to construct new structures and create diverse levels, as we will observe next.

7.5.1. Algorithm

The creation process consists in generating content horizontally. In the case of *Infinite Mario Bros.* and *Infinite Tux*, the generation process goes from left to right but the algorithm can be applied from right to left as well. Starting in the beginning of a random level, the algorithm copies sequentially its columns, until it reaches a column that exists more than once in the example set. In these cases, which we refer as transition points, the copying process may shift to one of the matches at the corresponding column. For that purpose, a shifting probability is established as a parameter. In Figure 7.21, we present a possible initial set of levels for the game *Infinite Tux*, which can serve the purposes of this algorithm. The identified transition points are displayed in the lower part of the image, with their reference to the level (*L*) and respective column (*C*) where they occur.

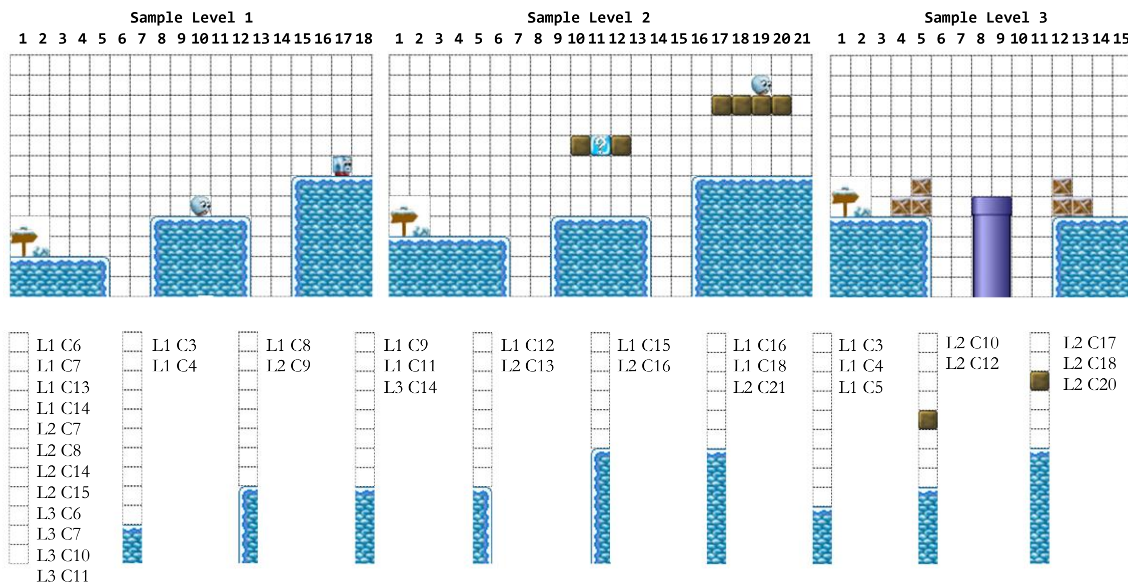


Figure 7.21 – Possible example set of levels for *Infinite Tux* in the context of the chunk overlap generation algorithm (top) and identified transition points (bottom).

As stated, every time the copying process reaches a certain transition point, the copying process may shift to another position that corresponds to the reached transition point. The process continues the copying process from that position, possibly changing several times between the existing base levels, until the result achieves a desired length, within a certain deviation margin. In Figure 7.22, we present an example of a level that was generated from the previous sample set. Transitions between levels are marked with the reference to the destination shifting location. It is possible to observe that the result provides a playable scenario, in which different types of geometry were created, in relation to the initial set.

7.5.2. Output Correction

In our tests, we detected one main issue in the usage of this approach directly that, however, can be fixed recurring to our movement rules that were previously presented in subsection 6.3.1. This problematic case is the creation of impossible game situations due to incompatible transitions. For instance, a common and natural transition point occurs in gaps, as they consist of multiple columns composed only by empty cells. This match represents a wild card situation because every gap column matches another gap column, breaking the main principles of the overlapping approach. Without any additional conditions, the presented algorithm may shift multiple times from level to level in gap parts generating a very large gap that the character cannot jump across.

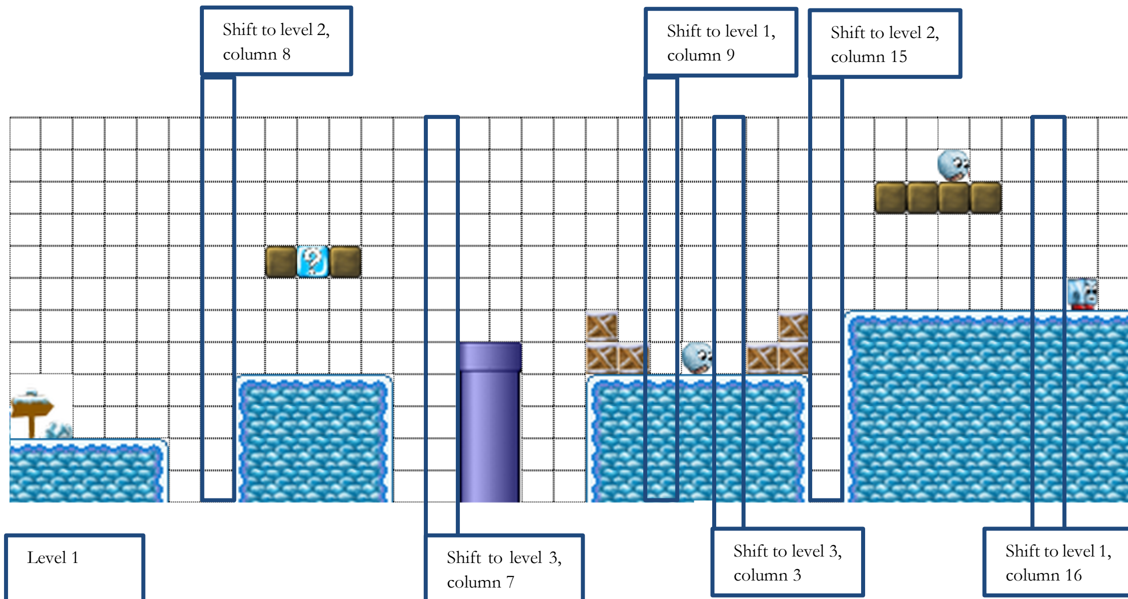


Figure 7.22 – Example of an automatically generated level using the chunk overlap algorithm. Transitions between levels are highlighted.

These situations could be solved by preventing level shifting in gaps. However, this would not be an elegant solution as it consists in a particularisation of something that was meant to be generic. Additionally, this *ad-hoc* solution would not prevent the occurrence of similar situations in implicit gaps, represented in situations where the avatar has obligatorily to jump between two separate platforms. An example of such situation is presented in Figure 7.23, where the two highest platforms represent an implicit gap that requires the user to perform the jump represented with an arrow.

To illustrate how performing a transition in an implicit gap can result in an erratic output, we will consider the previous situation, which we will refer as segment A, and an additional level sample that we will refer as segment B. Both are represented in Figure 7.24, where we have also marked the possible transition points with rectangles. It is possible to verify that columns 1 and 8 in segment A matches the first and the last three columns in segment B. Then, considering possible shifts from A at column 8 to B at column 1, and from B at column 8 to A at column 8, the result is the impossible situation presented in Figure 7.25, where the original implicit gap has been enlarged, now requiring a jump that the user cannot accomplish (represented with an arrow in the same figure).

The usage of graph information allows us to overcome this issue. Every time that a new section is generated, the movement rules should be applied to obtain the new level graph. As the left part of the level is still the same, there is only the need of applying these rules to the newly generated part and an additional buffer of previous content to map the transitions between the new segment and the remaining of the level. As long as new segments allow reaching the right part of the level, the algorithm can continue. Otherwise, the last created segment must be ignored and the process should rewind to the last shifting process.

7.5.3. Closing Remarks

With this third alternative generator, we intended not only to explore the possibility of creating new levels starting from an initial humanly created set, but also to present a complementary approach to our previous proposals, with concepts that are more suitable for the creation of open scenarios.

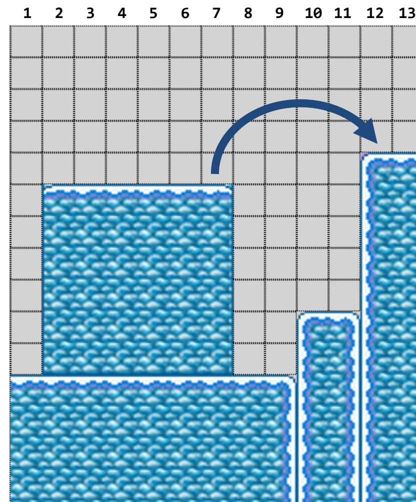


Figure 7.23 – Example of a level situation with an implicit gap.

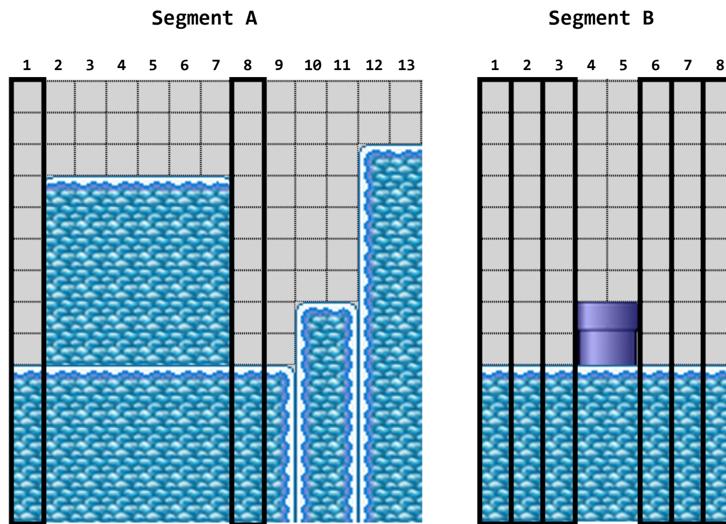


Figure 7.24 – Example of two arbitrary level parts with common columns where shifting is allowed. Equal columns are marked with rectangles.

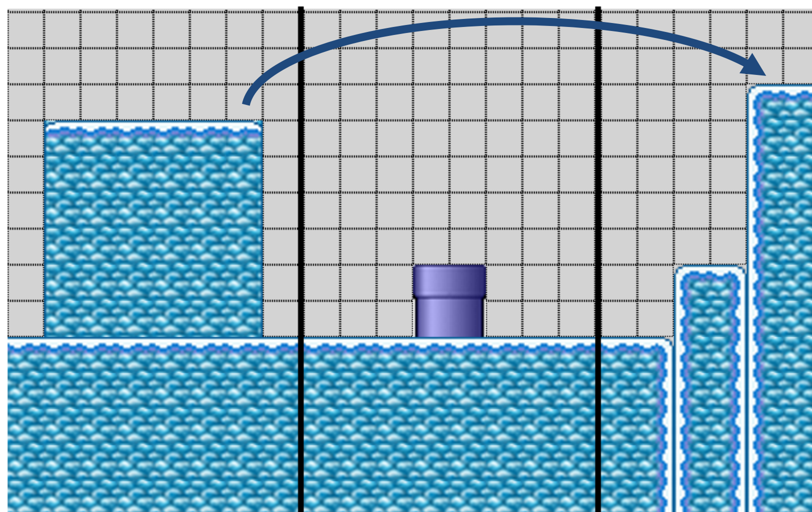


Figure 7.25 – Example of a wrongly created level using basic overlapping, as the level is not traversable. The transitions between chunks are marked with strong lines.

Moreover, the presented alternative represents our contribution to the chunk-based approach for level generation, in which we raise the possibility of obtaining those chunks automatically. In fact, each section generated between two transitions is a large implicit chunk, automatically detected by the system. Although it is possible to think that copying sections from an initial set of results have a limited domain for the output, which can result in repetitive levels, the fact is that, even with a small set of initial levels, the number of possible combinations expands, and the system is capable of creating different situations not thought before. For instance, recalling the example generated level presented in Figure 7.22, the system created a new type of zone, consisting of a valley with an enemy in the middle, between the third and the fourth level shifting. This interesting design situation is common in platform videogames and was not present in the initial level set.

Additionally, there are a few aspects that we would like to point for further research in this area, which might bring some new interesting progresses. To start, it would be interesting to explore with more detail the alternatives for the generation process. For instance, a bidirectional creation process, generating alongside from the start and the end of the level, can increase the control over the length of the output. It is more likely to find an intermediate chunk to connect with the two parallel segments than making the initial unidirectional algorithm converge to the level end when size restrictions are accomplished. Furthermore, the concept of automatic extraction can be expanded by removing the horizontal movement restriction, considering any type of sliding windows over the level area, searching for matches in any possible direction. This type of improvement can take into account the graph representation for the generated level in order to understand the existing possible paths and detect which directions have more potential for further expansions. Finally, we would like to point to the possibility of detecting resizable groups, following the concept of resizable group that we have presented regarding our framework, in subsection 5.3.3. Sliding a window through an existing set of levels is a potential approach to obtain common level parts (for instance, in section 6.3.2, we have used a similar approach to extract rules for graph creation). A similar method to detect the specific case of resizable groups is an interesting point of further research.

7.6. Algorithms' Taxonomy and Integration

In this chapter, we started by presenting an overview of the most relevant techniques for the automatic level generation of platform game levels. In addition, we have proposed three original approaches in the context of level generation. Each one has different purposes and presents distinct output types. Besides, in chapter 3, we have observed a set of design patterns that serve level creation for platform videogames. Therefore, on one side, we have a set of creational patterns and, on the other side, we have a set of techniques that can generate certain type of levels or level parts with certain features. It is thus important to establish an intermediate platform in-between, bridging these two sides in order to integrate multiple generation approaches with the different design patterns that one can find in level creation. In section 2.3, we have presented different taxonomies and categorisations regarding general PCG. At this point, we present a similar type of study, yet specialised to the specific case of automatic level generation for platform videogames and focusing the referred bridging requisite between design patterns and generation algorithms.

7.6.1. Approaches

As an initial point regarding the algorithms' taxonomies, we believe it is important to recall the related work presented in section 7.2 and summarise the different existing ideas into specific categories regarding the considered approach. For that matter, we have identified the following main different features with two opposite sides in generation approaches:

- **Compositional vs. global.** This distinction aims to state the two main different views regarding the granularity of the generation process, which can be directed to the generation of

multiple segments that are assembled together or to an overall creation process without any type of segmentation. Typically, it is possible to interrupt a compositional algorithm and keep the incomplete partial result as a small sample level, which does not occur in global techniques that only allow valid interpretation of the output in the very end of the process.

- **Flat vs. layered**. This differentiation refers to the possibility of having the content interpreted as proposed by Hendrix *et al.* (Hendrikx et al., 2013) and explained in subsection 2.3.3, decomposed into multiple abstraction levels of detail, with specialised techniques for each different layer or, on the opposite side, analysed continuously without decomposing it into different levels of abstraction.

- **Parameterised vs. mixed**. Human interference over the generation process can be done explicitly or implicitly, which are the two sides of this distinction group. In a parametrised algorithm, the user defines global parameters such as the number of gaps and the length of the level. Alternatively, these parameters can be related to the gameplay experience, representing, for instance, the represented difficulty or the implication of a certain emotion. Mixed algorithms provide direct control over the level, allowing the user to state explicitly the content of some parts. One can see it as an incomplete classical level editing process in which the content is finished procedurally.

- **Creative vs. example-based**. This category distinguishes algorithms that have the creational processed embedded in the algorithmic structure, namely by the inclusion of design heuristics or other creational rules within the code, in opposition to more general algorithms that work with a set of examples or templates in order to infer possible combinations and extensions of such initial set.

7.6.2. Output Type

Besides the different approaches presented in the previous section, it is also important to look at the existing algorithms regarding their main goal and the generated type of content. In this case, we propose the following classification for the generation techniques and algorithms:

- **Structural generators**, referring to the algorithms that are capable of generating global level structures without specific detail. The notion of global should not be interpreted specifically as a full size level but as a level part with multiple path alternatives and composed challenges.

- **Segment generators,** which can handle efficiently the task of generating small and linear level sections according to certain parameters.

- **Adapters**, which group the algorithms that change previously created content as a perfecting step. Those can be decomposed into subgroups regarding their goal as follows:
    - **Aesthetic**, directed to adaptations that aim only to the inclusion and the positioning of decorative elements, without any direct or indirect interference over gameplay.
    - **Gameplay**, which configure the positioning of certain gaming elements in order to manage perceptions regarding gameplay, such as difficulty, enjoyment, fun or any other related factors. However, a gameplay adapter does not change the path structures that are represented in the level.
    - **Structural**, changing the initial established movement graph implying differences to the original generated structure.

7.6.3. Bridging Generation Algorithms with Quest Patterns

Lastly, we recall the quest (and side-quest) patterns for platform videogames, previously presented in subsection 3.3.4, and extend them regarding the possible approaches for their implementation by proposing a set of variables that are required for such implementation, as follows:

- **Checkpoint**. Should be implemented as a segment generator, where aesthetic and gameplay adapters can be used. Alternatively, it can be implemented with a structural generator restricted to a certain size if it is possible to ensure that a single path solution is generated, for instance, recurring to a graph-based analysis. Structural adapters are not suitable as they can generate variants that result in other quest patterns. Generally, algorithms for this type of quest should allow establishing a length value. Specifying the main type of goal(s) represented by the quest implies different sets of additional variables, defined in the following manner:
  - Talk to: *whom* (mandatory).
  - Deliver: *which item* (mandatory) and to *whom/what* (mandatory).
  - Grab: *which item* (mandatory).
  - Milestone: a reference *landmark* (optional).
  - Escort: *whom/what* (mandatory).
  - Escape: *whom/what* (mandatory).
  - Interact with: *what* (mandatory) and *with each item* (optional).
- **Hostile zone**. Can be implemented either with structural or segment generators, as the zones are not required to have any specific format. Still, the dimensions of the action zone are an obvious variable to establish. In addition, any type of adapters can be used as long as they can apply the adaptations within the restricted zone. For each of the possible concretisations of this quest, a different set of variables are also required, which are described next:
  - Defeat *n* opponents*: how many* (mandatory) and a set of *possible opponents* (one or more).
  - Defeat a horde of opponents until a certain condition is reached: the *item to gather* (optional) and the *entity to modify* (optional), in which it is mandatory to pick at least one.
  - Defeat one particular opponent: *who/what* (mandatory).
- **Multiple checkpoints**. As a non-linear path is required to be established in this quest, the exclusive usage of a segment generator is not possible. Thus, to achieve this type of quest, such generator could only be used with a further structural adapter. In alternative, a structural generation algorithm might be used if it can receive and process information regarding dependencies of gaming entities and restrict their usage. Regarding the possible alternatives for multiple checkpoints, they require different sets of variables, described as follows:
  - Key/lock: a *key item* (mandatory) and a *lock entity* (mandatory).
  - Composed key/lock: a set composed by two or more pairs with a *key item* and a *lock entity* (mandatory).
  - Switch/door: a *switch entity* (mandatory) and a *door entity* (mandatory).
  - Composed switch/door: a set composed by two or more pairs with a *switch entity* and a *door entity* (mandatory).

  - Character modifier: a *modifier item* (mandatory) and a reference to a *movement rule set* (mandatory), according to our framework.

- **Grab collectible points**. This side-quest pattern can be seen as an adaptation of the checkpoint pattern, in which an optional scoring mechanism is added. Therefore, it can be equally implemented as a segment generator, where aesthetic and gameplay adapters can be used, namely to the inclusion of the scoring mechanism. Its variable is a mandatory *scoring item*.

- **Reach a secret zone**. The principles of this side-quest denote two specific zones: the main path and the hidden zone that contains the secret item. It can be seen as an adaptation of the multiple-checkpoint pattern, with the specificity that the secondary area is hidden. Thus, likewise that pattern, this one can be implemented as a segment generator followed by a structural adapter, or directly as a structural generator. According to the type of secret zone, a different set of variables is required, as presented next:
  - Collectibles data: *game item* (mandatory) and *collectible item* (mandatory).
  - Collectible power: *power-up item* (mandatory).

Moreover, in the definition of each of the previously presented patterns, it should be possible to state explicitly a certain difficulty value or refer to a certain difficulty profile. It is up to the algorithm to use that information in an appropriate manner. An inappropriate difficulty measurement or ignoring such data does not imply that it is not possible to describe and generate levels from a set of quest, but will imply less control over that type of features. In the next section, we will show how this quest description was applied into a semi-automatic generator for a practical usage.

7.7. Semi-Automatic Level Generator

7.7.1. Approach

We have seen, in subsection 3.3.4, a set of quest patterns for platform videogames. We have also presented an additional characterisation of these patterns, in subsection 7.6.3, where we have also defined a set of variables for each quest variation. From a PCG perspective, one can see a quest as an abstraction, describing the motivation of a game character to go through a specific part of a level. In this section, we present a semi-automatic level generator based on quests. As in the mixed-initiative (Smith et al., 2010a; Smith, Whitehead, et al., 2011), already referred in subsection 7.2, the users and the computer cooperate in the process of level design. However, in our case, the users and the computer work in different levels of abstraction. The user defines a set of quests that state the main content of the level, and it is up to the computer to generate the physical structures that materialise those quests. For instance, the user can define a level to start with a *checkpoint* pattern of the type *grab*, meaning that the first part of that level will consist in a path with the specified item in the end. The quest does not state the direction of the path or other physical features. The automatic generator will define those automatically, when the level is being created.

7.7.2. Interface and Main Features

Figure 7.26 shows the current UI of the application in the creation of a level for the videogame *Prince of Persia*. The designer manually defines the sequence of quests with their respective goals and a set of side-quests, as observable on the left part of the figure. Also in the presented screenshot, the user defined a level to contain initially a trap zone, followed by a side-quest with a hidden potion, and finalising with a multiple checkpoint. The automatic generation process takes those quests as input and generates the geometry using a set of algorithms that differ according to the used patterns, as explained in subsection 7.6.3. It is possible to generate a concretisation of the quest structure, presented on the right part of the window, where the generated segments have been highlighted for

comprehension aid. In this case, the application was configured to also use predefined segments for the start and the end of the level.

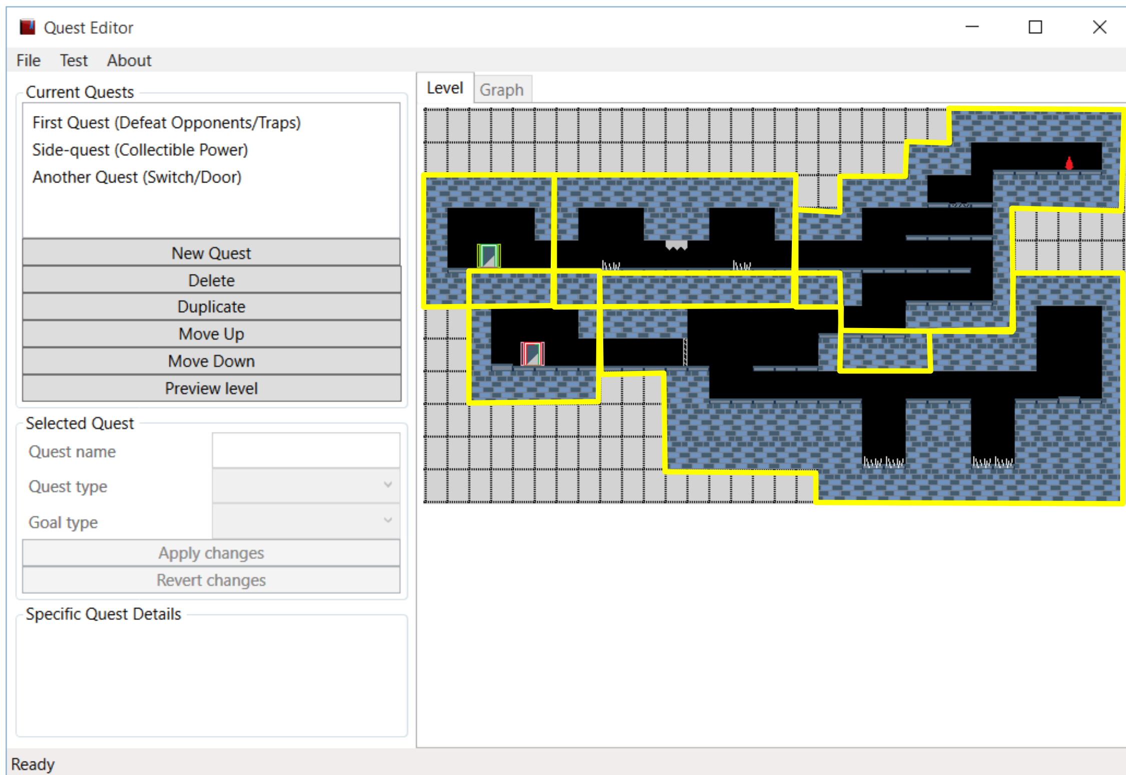


Figure 7.26 – Interface of the implemented semi-automatic level generator based on quests.

The patterns can be set according to certain parameters, as described in the previous section. In Figure 7.27, we exemplify the configuration of the patterns used in the creation of the presented level. The first configuration contains the definition of a *hostile zone*, where the player faces a set of traps. In this case, the user defined that this zone should contain three traps as a combination of choppers and spikes. In the second configuration, we show the creation of a side-quest, a *secret zone* containing a collectible power. It allows the definition of the power-up item, which in the example is a life potion. The last example depicts the creation of a *multiple checkpoint* pattern with a gate and its respective trigger. These are the configurable features of the quest.

The quests are processed sequentially, consisting in the generation of the level segments and their respective subgraphs, depicting the movements of the avatar, using the graph extraction mechanism based on rules, presented in subsection 6.3.1. The generation of a segment for a quest has the following requirements:

- The placement of the new segment must ensure that it has a node that is horizontally adjacent to a node of the previously generated segment. This guarantees that all segments are consecutively connected.
- The new segment can overlap previously created segments only if the overlapping cells are equal to the existing ones, or if they are their descendants, according to the defined block hierarchy, established in our framework, as presented in subsection 5.3.2. Back in Figure 7.26, it is possible to observe various segments overlapping in wall cells.

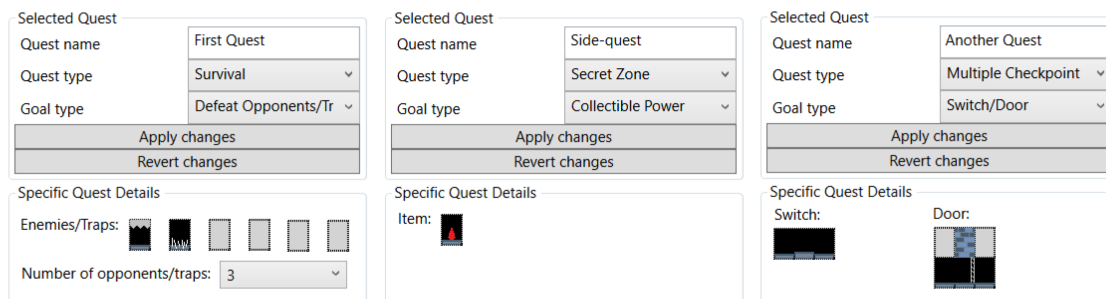


Figure 7.27 – Configurations for different patterns, namely *secret zone* of *collectible powers*, *multiple checkpoint* with *switch/door* and *hostile zone* to *defeat enemies*.

The generation is a trial-and-error process, performed automatically. At each iteration, a candidate segment is generated, and the possible adjacency combinations of nodes between this segment and the previously generated one are analysed. If the segment fulfils the previous requirements, it is added to the level and the process moves to the next quest. Otherwise, the segment is discarded and the procedure is repeated with a new candidate. The algorithm rolls back the generation to a prior state if too many candidates are generated unsuccessfully. For this purpose, a limit number of candidates is established as a parameter.

It is possible to rebuild the whole level from scratch or only certain quests, allowing a designer to regenerate some parts while keeping others intact. Figure 7.28 shows a sample of multiple variants generated for the pattern *checkpoint* of the type *milestone*, in different runs of the algorithm. For the generation of this specific type of quest in *Prince of Persia*, a composition algorithm was implemented in association with an automatic chunk detection process. The segments were thus created by retrieving common linear segments from the original set of levels in this videogame, with a variant of the algorithm presented in section 7.5. We did not define any specific entity to denote the milestone, as this is optional to the quest pattern. Therefore, the represented segments have an implicit milestone at their end without a physical manifestation.

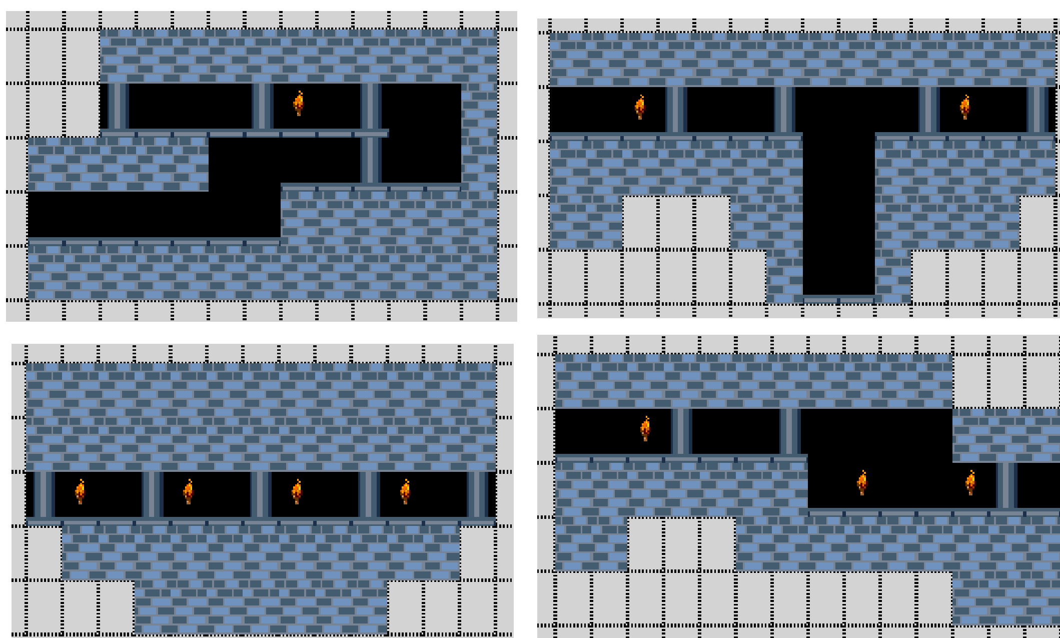


Figure 7.28 – Alternative geometries generated for a *checkpoint* pattern of the type *milestone*, considering the game *Prince of Persia*.

In the same manner, the multiple checkpoint pattern can be materialised into different configurations. Some examples are presented in Figure 7.29. This pattern has been implemented using a predefined set of humanly authored bifurcation chunks, which are expanded automatically into the alternative directions as independent *checkpoint* patterns. Then, the adaption algorithm, explained in section 7.4, is applied in order to add the button and the gate in appropriate locations.

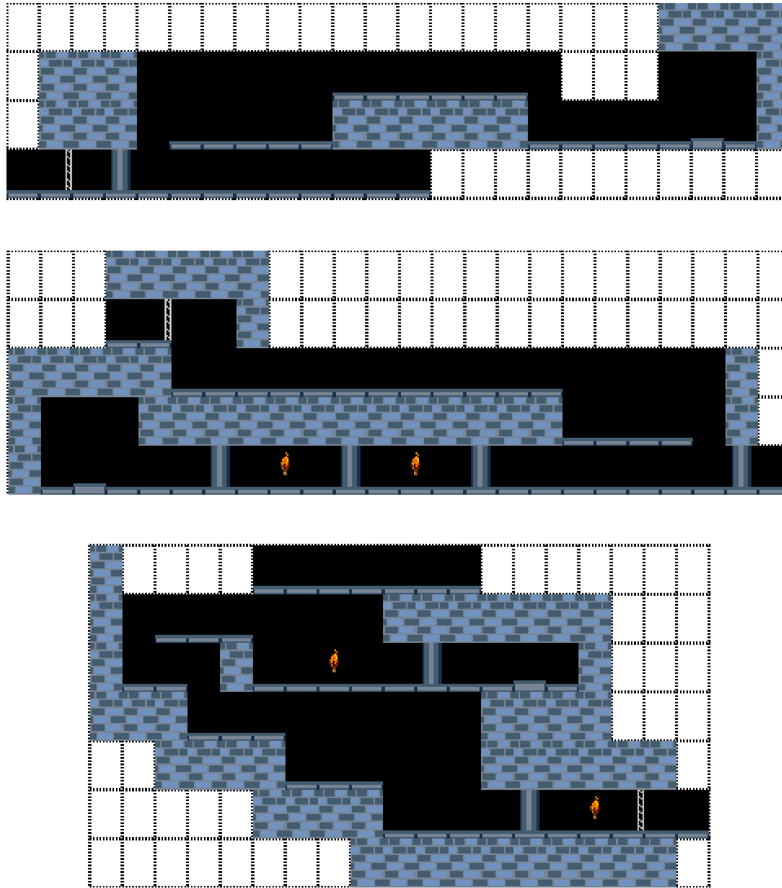


Figure 7.29 – Alternative geometries generated for a *multiple checkpoint* pattern of the type *switch/door*, considering the game *Prince of Persia*.

Furthermore, some patterns allow the definition of difficulty based variables. An example is the *hostile zone* pattern, which we have implemented to map user skills to the frequency of opponents and respective strength. Figure 7.30 shows different generated alternatives for the same segment with distinct difficulty profiles, in this case using the videogame *Infinite Tux*. A simple terrain displacement generator was used followed by a random inclusion of opponents, regulated by the difficulty estimation constraints (briefly, a certain difficulty profile has to match a respective frequency of opponents).

7.7.3. Closing Remarks

The implemented semi-automatic generator, based on quests, shows the potential of using quests as an initial authoring mechanism to feed lower abstraction generation algorithms. As we are decomposing the level structure into quests and goals, the generation algorithms are responsible for small level parts at each time, reducing the computational complexity. For instance, evolving complete level structures with genetic algorithms as a global process is a time-consuming task, but generating a small portion to match a certain quest can be achieved promptly. Besides, this decomposition into smaller regions potentiates simpler algorithms to be used integrated with techniques that are more complex. As an example, the referred *checkpoint* pattern typically consists of a systematic sequence of jumps and/or traps, which can be generated with ease using an *ad-hoc* algorithm. Still, these parts can be

merged with other segments representing quests that require more complex algorithms, such as the *multiple checkpoint* pattern, where paths and routes must be analysed to create those situations.

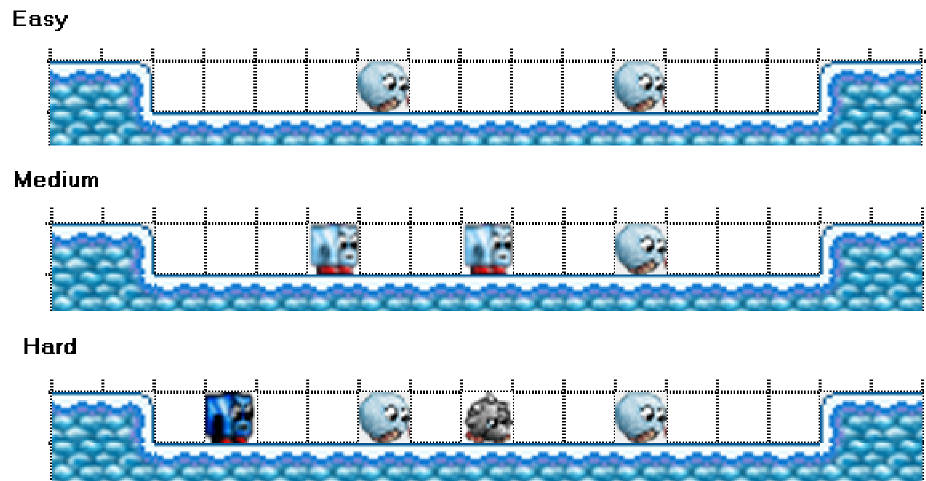


Figure 7.30 – Alternative configuration of a certain level part, for the game *Infinite Tux*, configured for different difficulty settings.

We envision this approach for two different specific usage scenarios. First, we believe it is a valid alternative to speed up the process of level design. The designer can use the quest-based approach to create an initial level structure, with a predefined structure, and then tune it in a level editor, including some specific details that he/she desires. In section 8.3, we will present our arguments in favour of this proposal. Secondly, a game can have its levels defined only as the quest structure, and the generation takes place at runtime. In this case, the players will always have a different level to play, which is one of the strongest arguments in favour of PCG, but they will have a fixed background set of quests to ensure coherence with a narrative line. For instance, in the first level of the game *Prince of Persia*, the user has to find a sword before defeating an enemy, and then conclude the level. The quest-based generation allows the definition of this important set of events without restricting the content to one specific level.

Lastly, we consider important to address, as a point for further improvements, the development of a more efficient mechanism for the generation process. As we have stated, in the previous subsection, the generation is a trial-and-error mechanism, which works as a blind search over the possible combinations of segments. At a certain iteration, the algorithm has no evidence about the potential of the ongoing solution. A possible method is to apply heuristics to assess the potential of the current state. Then, the generation mechanism can be replaced with a more efficient search algorithm, such as A*, examining the alternatives in a less randomised manner.

7.8. Concluding Remarks

In this chapter, we have presented our main contributions regarding the effective implementation of generation algorithms for platform videogames, namely:

- A generator embedding a set of design heuristics into a genetic algorithm, described in section 7.3. This generator is suitable for the creation of general level structures with the core details about paths.
- An adaptation algorithm, explained in section 7.4, that uses overall level structures, such as those provided by the previous algorithm, and produces an enhanced version, comprising

some design patterns that encourage exploratory gameplay, and including procedures of difficulty adjustment.

- A generator based on the combination of previously created level segments, presented in section 7.5. This generator is able to create open scenarios of side scrolling gameplay.

The referred implementations benefited several different concepts explored along this dissertation. First, most of the experiments and the implemented techniques were sustained with the architecture proposed in section 5.2, and using the level representation framework described in section 5.3. Furthermore, we have applied the quest patterns that we have identified previously in subsection 3.3.4, in order to develop the semi-automatic tool for level generation presented in section 7.7, in which the proposed algorithms were included. In addition, the difficulty measurements that were proposed in chapter 4 were applied in those algorithms and the structural studies based on graphs presented in chapter 6 were also applied in some of the represented quests. Moreover, the examples of the current chapter show that our approaches are able to build coherent level structures, including new design patterns. Still, it is also important to analyse the quality of the generated levels. We address that topic in the next chapter.

8. Results and Evaluation

"In the same way that simple computational models can serve to elucidate the dynamics of otherwise complex natural phenomena (Humphreys, 2008), it is possible that models of fun will serve to illustrate fundamental principles of game design."

(Sorenson & Pasquier, 2010b)

8.1. Introduction

Throughout this document, we have presented some tests where our architecture was implemented. Primarily, those tests allowed us to understand the viability of using such architecture in an automatic generation system. Furthermore, we have performed some tests to validate the proposed metrics regarding game difficulty. Additionally, the proposed generation algorithms were tested in order to understand their plausibility and, likewise, to detect the main requirements for further improvements. To reinforce the validity of the proposed metrics and the applicability of the designed algorithms, we have implemented a more extensive evaluation study, which will be presented in this chapter. This final study aims to assess the following aspects of this research:

- The quality of the procedurally generated levels using the proposed alternatives for the generation process.
- The effectiveness of the proposed metrics to estimate difficulty and the plausibility of the proposed hypothesis, regarding the players' behaviours and strategies facing common challenges in platform videogames.
- The usefulness of the semi-automatic generation approach using humanly authored quests and the main scenarios in which it can be used.

In the next section, we will describe the main features of this final study. Next, in section 8.3, we will analyse the obtained data from such study. Lastly, in section 8.4, we will review the main conclusions about the obtained results.

8.2. Experiment Details

In order to assess the aspects referred in the previous section, we developed a prototype of a platform game, entitled *The Platformer*[1] (Figure 8.1). It was implemented with *Unity* and was made available to the users through the Web. It consists of a game similar to the original version of *Prince of Persia*, which can be played directly inside the browser. The player controls a character in a dungeon-themed environment, searching for the exit door while avoiding different types of traps, which we will cover in the next section. The main movements are running (Figure 8.2), jumping (Figure 8.3) and climbing (Figure 8.4). Along the level, the player may encounter closed gates (Figure 8.5) that are opened by

[1] http://platformer.gearmind.com

stepping over their respective buttons (Figure 8.6). The exit door (Figure 8.7) is also opened with one these buttons, allowing the player to enter and finish the level.

Figure 8.1 – Screenshot of our prototype game, *The Platformer*.

Figure 8.2 – Screenshot of our prototype, *The Platformer*. The character is running in the scenario.

Figure 8.3 – Screenshot of our prototype, *The Platformer*. The character is jumping across a gap.

Figure 8.4 – Screenshot of our prototype, *The Platformer*. The character is climbing to a platform above.

Figure 8.5 – Screenshot of our prototype, *The Platformer*. The character is facing a closed gate.

Figure 8.6 – Screenshot of our prototype, *The Platformer*. The character is stepping over a button that opens a gate.

Figure 8.7 – Screenshot of our prototype, *The Platformer*. The character is entering the exit door, finishing the level.

In the same way as in *Little Big Planet*, the players are potential level creators within our system. Therefore, a simplified version of our *Platform Level Editor* (described in section 5.4) is available for the users to download. A version of our *Quest Editor* (presented in section 7.7), configured to this game, is also available for the users to download.

The users can create levels using the previous authoring tools and submit them to the system. Moreover, the system also comprises levels generated with a fully automated process, using a composed generator based on the algorithm presented in section 7.3, to create level structures, and the algorithm presented in section 7.4, to perfect those same structures. Therefore, for the sake of convenience, we will refer to the different types of levels in the following manner:

- **Procedural levels** are those that were generated with the fully automated process.
- **Authored levels** refer to the levels that were created by the users using the *Platform Level Editor*.
- **Quest levels** are the descriptions created with the *Quest Editor*.

With the purpose of keeping the proportionality between the existences of the different types of levels, the system generates one procedural level for each two levels that are uploaded. Whereas it is not plausible to force the users to upload an equal number of authored and quest levels, this process ensures that one third of the existing levels are procedural and that the remaining two thirds are divided between authored and quest levels.

Every time the system receives a level created with the *Quest Editor*, five different level configurations of the respective quest structure are generated. The existence of multiple variants of a quest level illustrates the potential of the semi-automatic generation, providing the same base content with geometric variations. With a limited set of alternatives, we ensure that the different players share one single domain of existing levels, and each level is played eventually more than once.

When the user plays the game, he/she receives a random level from the existing ones, with even probabilities of being procedural, authored or quest levels. In the end of the level, the user can rate its quality, in a scale from 1 to 5, and can post his/her speculation about its author (human or computer). Moreover, gameplay data is recorded regarding the different challenges.

We have also retrieved additional information about the process of level design, in particular to obtain feedback from professionals and experts in the topic. As this further experiment required more complex and time-consuming tasks, a convenience sample was established. First, the participants were identified within one of the following categories:

- **Game designer**, for those who work/worked directly in game design tasks.
- **Non-designer level creators with experience**, referring to the users that do not work as game designers but have experience creating content in videogames, independently of the genre.
- **Non-designer level creators without experience**, for the users that do not have experience in tasks of content creation in videogames.

In further discussions, the profiles will be referred under the designations of game designer, experienced non-designer and inexperienced non-designer.

After being familiarised with the game and its main entities, the participants were asked to perform the following tasks:

- Conceive a mental sketch of a possible level for the game.
- Create a sketch of the level with the *Platform Level Editor*, keeping track of the required time.
- Perfect the level to make it a final version, keeping track of the required time.
- Define the corresponding version of the level with the *Quest Editor*, keeping track of the required time.

8.3. Data Analysis

The prototype was initially tested by students, spanning different degree courses, and then made available openly. Two weeks after its release, the gathered gameplay information resulted in a study dataset with the following features:

- 93 registered players;
- 52 submitted levels (28 authored levels and 24 quest levels);
- 26 procedurally generated levels; and
- 1587 level attempts.

In this section, we will analyse the data that we have obtained after implementing the experiment described in the previous section.

8.3.1. Players' Strategies and Behaviours

Regarding players' strategies and behaviours to tackle challenges, we have extended the experiment of our prior study (Mourato et al., 2014), described in section 4.4, where we also pointed the most relevant features that we have obtained. At this point, we extend that study by observing the players' behaviours with a larger and more heterogeneous sample.

In the implemented game, challenge arises from the following situations:

- Spatial challenges, which are materialised as gaps. The character has to perform jumps at appropriate locations in order to cross such gaps. This challenge is depicted in Figure 8.8.

The game has two main types of platforms: stone and wooden. When the level is rendered, wooden platforms are randomly translated horizontally and vertically, not more than half the size of the cells. This displacement creates multiple various gap combinations, additional to the grid resolution, which is useful for our studies regarding difficulties, as we will observe in the next section.

- Temporal challenges, represented as choppers that periodically move out of a wall, killing the game character. One needs to cross this contraption within the time interval when the chopper is retracted. In Figure 8.9, we present a screenshot of such challenge. Each blade has a fixed attack time, between one and three seconds, randomly established when the player starts the level.
- Dynamic challenges, represented as moving wooden platforms, creating a gap with variable dimension over time, as presented in Figure 8.10.
- Combined spatial and temporal challenges, merging the first two types of trials, which consist of choppers placed within gaps, as presented in Figure 8.11.

Figure 8.8 – Screenshot of our prototype, *The Platformer*. The player is attempting a jump over a gap.

Figure 8.9 – Screenshot of our prototype, *The Platformer*. The player is trying to move to the right avoiding the blade.

Figure 8.10 – Screenshot of our prototype, *The Platformer*. The player is jumping to a moving platform.

Figure 8.11 – Screenshot of our prototype, *The Platformer*. The player is jumping between platforms while avoiding a blade trap.

Each of these types of challenges has specific features with influence over the concept of difficulty and will be analysed next.

Spatial Challenges

In the same manner as we did in our initial study, we have started by analysing the players' behaviour in the most common situation that one can encounter in platform videogames, jumps over gaps.

Following our premise that the players aim their jumps to occur slightly before the edge of the platform, we measured at which distance the jumps were performed and obtained the histogram represented in Figure 8.12.

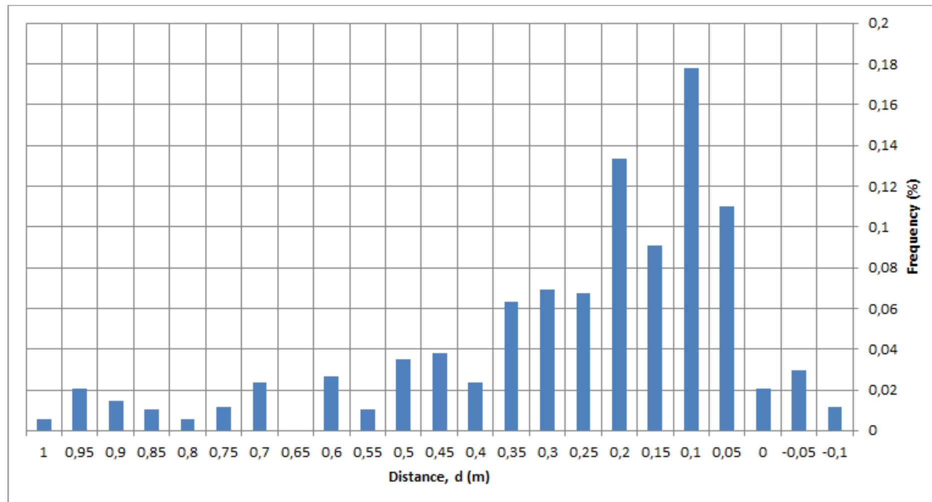


Figure 8.12 – Frequencies for the distance in relation to the platform edge in the jump origin, when jumping between static platforms.

It is possible to notice a trend in the data and it is possible to observe that the jumps occur following a distribution centred in a value corresponding to a safety distance. Even though the data scattering presents resemblance with a normal distribution, such premise could not be confirmed. We have applied a K-S test to the data and the obtained *p*-value does not allow us to assume normality. However, applying the same test to the individual distributions of each player, typically results in *p*-values over 0.05, which is consistent with normality for a confidence value of 95% (in our sample, about 75% of the individual distributions presented a *p*-value over 0.05). Accordingly, the individual performances have specificities that are stronger than the overall behaviour. In addition, to verify the differences amongst the players, we have measured each player's success rate and analysed their individual average (Figure 8.13) and standard deviation (Figure 8.14) regarding *d*. For the matter, we have excluded the players with less than 30 attempts in this type of challenge, as a small set of tries makes the percentages of success susceptible of biasing.

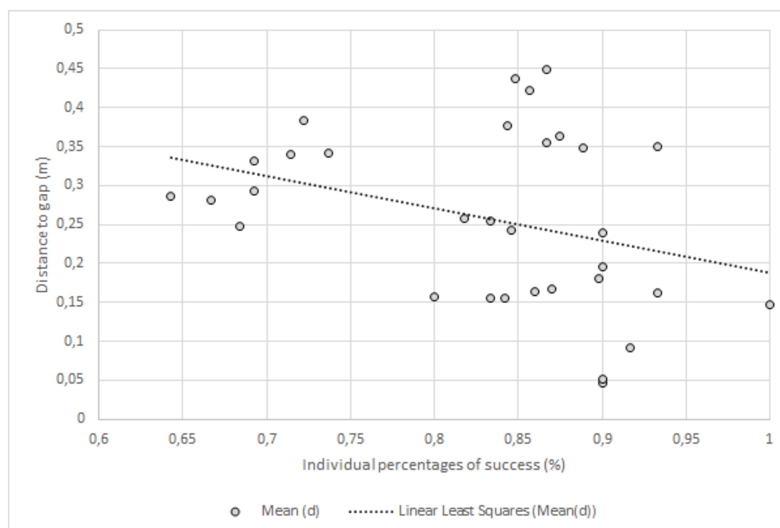


Figure 8.13 – Mean of the distance to the gap's edge in relation to the player's ranking for a spatial challenge.

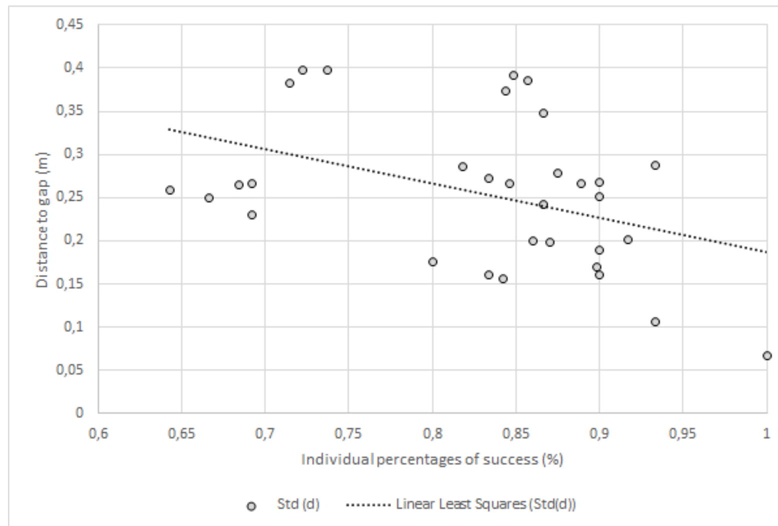


Figure 8.14 – Standard deviation of the distance to the gap's edge in relation to the player's ranking for a spatial challenge.

It is possible to observe that higher ranked players tend to perform the jumps nearer to the platform edge (lower mean) and with more precision (less standard deviation). While obtaining lower mean values is a natural consequence regarding the features of the jump, it is particularly interesting to observe a similar trend regarding the standard deviation. This confirms the hypothesis that the players with more skills are more precise in their actions, as proposed in our estimator based on distributions, presented in subsection 4.4.2. Such confirmation supports the usage of the estimator for difficulty prediction with different types of user profiles, corresponding to distinct distribution parameters.

Time-Based Challenges

In the same manner as in the previous case, we have analysed how the players tackle time-based challenges. For this particular case, we have observed the instants at which the players crossed chopper contraptions. Those instants were normalised into the percentage of time between two chopper attacks. Figure 8.15 presents the results of such data, displaced within a histogram.

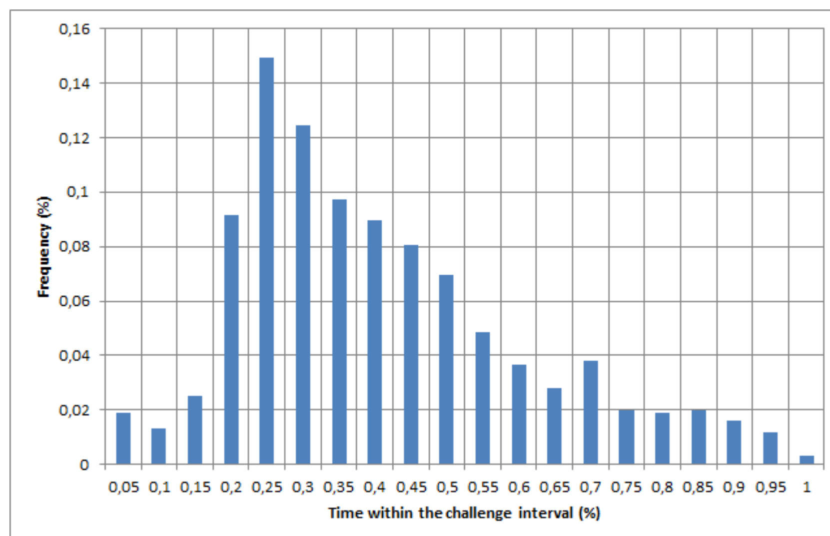


Figure 8.15 – Frequencies for the chosen instant to cross a timed-based challenge within one period.

In this case, it is still noticeable a trend in the data, with more occurrences in the middle part of the interval. However, the trend is weaker than in the previous case and the values are more scattered.

Two possible arguments can justify such situation. The first possible justification is that the players are not aiming to the middle of the interval. They wait until the harmful window passes and perform the move right after it, as a reflex. Therefore, the graphic is almost an indirect representation of the players' reaction time and, for that reason, there is a slight trend for the value to occur in the initial part of the interval. The second reason is that, in this type of challenge, the players tend to have a less cautious behaviour, because a careful approach means a break in the game's action. This is more common in earlier stages of the levels, where the consequence of failing is less significant than in later stages.

A K-S test was applied to verify if the distribution is normal, resulting in a *p*-value under 0.05, which does not allow us to take that assumption. However, similarly to the case of jumps across gaps, applying the same test to the players' distributions independently results in *p*-values over 0.05 for the most cases (in our sample, about 80% of the individual distributions have a *p*-value over 0.05), meaning that, with 95% of confidence, those distributions can be considered normal.

Once again, we also want to analyse the behavioural differences between the more and the less experienced players. We have displaced their success rate in this type of challenge and measured the time instants in which they tried to cross the chopper. The result can be observed in Figure 8.16, where we display the mean value of those instants, for each player, and in Figure 8.17, where the standard deviations of the instants are presented.

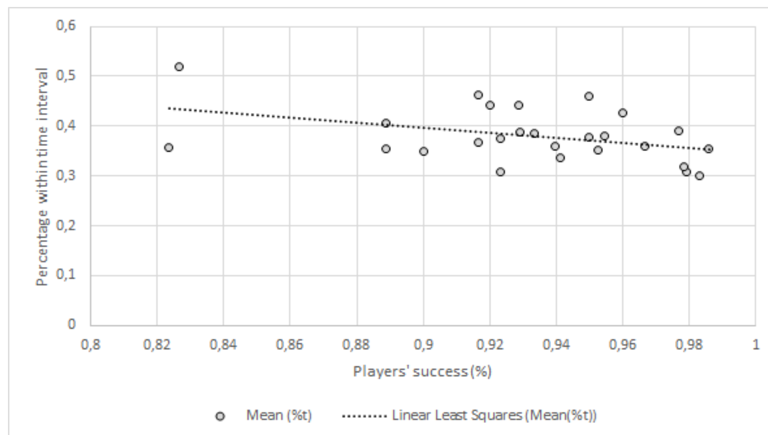


Figure 8.16 – Means of the chosen instant to cross the time-based challenge, in relation to the player's ranking.

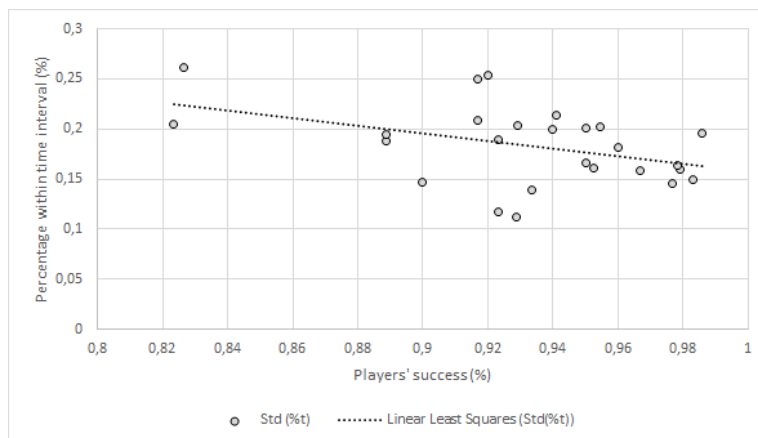


Figure 8.17 – Standard deviations of the chosen instant to cross the time-based challenge, in relation to the player's ranking.

We can also verify that the more successful players tend to have a lower standard deviation, reflecting their higher precision, even though the effect is less noticeable in comparison to the former case. When we observed the frequencies presented in Figure 8.15, we have argued that the players tend to start the movement as a reflex. This graph sustains that hypothesis, as we can observe that the players with higher success also start their movement earlier in the interval, meaning that they have a better reaction time according to the gameplay mechanics. As in the previous case of the spatial challenges, the confirmation of our hypothesis makes the estimator based on a parametrised distribution for time based-challenges, presented in subsection 4.4.2, a valid approach to estimate difficulty for different profiles.

Dynamic Challenges

Regarding moving platforms, we have also started by measuring the distance *d* to the origin platform edge for every performed jump, and represented the data as the histogram presented in Figure 8.18.

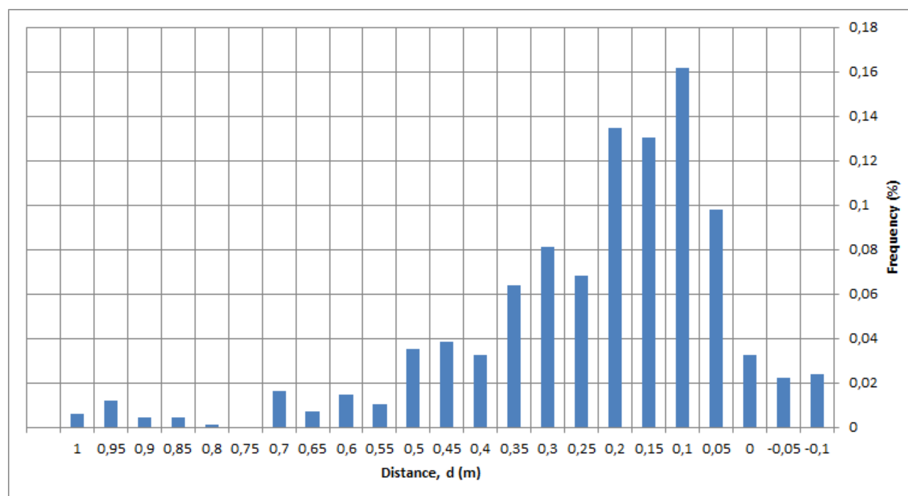


Figure 8.18 – Frequencies for the distance in relation to the platform edge in the jump origin, when jumping to a moving platform.

The same conclusions about normality can be extracted. The overall case cannot be considered normal, but when observing the individual distributions of each player such hypothesis is usually valid (in our sample, about 80% of the individual distributions presented a *p*-value over 0.05).

Regarding the time related component of the challenge, we have registered at which distance the moving platform was when the player attempted the jump, in relation to the minimum possible distance. Figure 8.19 shows a graphical representation of such data (negative values for *d* mean that the platform was moving towards the player and positive values represent the opposite).

In this case, it is possible to verify the intuitive notion that the players aim to perform the jump with the minimum gap distance. A K-S test for normality was also applied, resulting in a *p*-value of 0.057, confirming our hypothesis of normality. Moreover, it is important to refer that the obtained mean value for the distribution was -0.42, which represents an additional motion compensation. This means that, in general, the players perform the jumps slightly before the platform reaches the nearest position, aiming to land at the minimum distance.

In summary, the tests confirm an overall behaviour of the players, jumping to the moving platform at the shortest distance, whereas the trajectories follow distributions with different parameters for each different player. Therefore, it is possible to apply the difficulty estimator based on a parametrised distribution for dynamic challenges, as presented in subsection 4.4.2. One should consider global

parameters for the timing of the jump and specific player's parameters regarding the distance to the platform's edge.

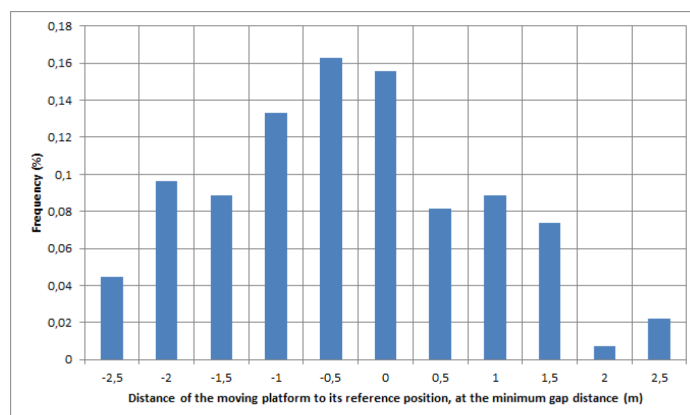


Figure 8.19 – Frequencies of moving platform positions for jumps towards those platforms.

Combinations of Spatial and Temporal Challenges

In subsection 4.4.2, we defined the combinations of spatial and temporal challenges under the hypothesis of independency. To corroborate that hypothesis, we have analysed success cases in this type of challenges and decomposed the failure cases into those caused by falling into the gap and those in which the player lost due to the chopper challenge. These values were compared to the existence of those same challenges independently. In Figure 8.20, we can observe failure in gaps in relation to the gap size, independently and in combination with a time-based challenge. In Figure 8.21, we observe failure in time-based challenges in relation to the duration of the interval, again independently and in combination.

In both situations, we obtain similar functions, a result that is coherent with our hypothesis of independency. In addition, the trend functions are almost identical, reinforcing that the overall behavioural differences are undistinguishable in the created subsets.

Furthermore, we have applied Student's *t*-tests to compare the two situations (gaps with and without the presence of a chopper and choppers with and without the presence of a gap), resulting in *p*-values of 0.721 and 0.823. Two sample groups can only be considered different regarding their mean for *p*-values under 0.05, which is not the case. Therefore, it is valid to assume the hypothesis of independency and calculate the combined probability of success by multiplying the separate estimators for the spatial and the time-based components of the challenge. Even though we applied the test without the premise of normality, the results of the previous tests are acceptable because we are working with a large sample, containing over 1000 attempts. In related literature, it is stated that 500 samples provide reliable results in such type of tests, even with non-normal distributions (Lumley, Diehr, Emerson, & Chen, 2002).

8.3.2. Difficulty Prediction

In the previous subsection, we have analysed how the players tackle the different types of challenges that they encounter in a platform videogame. Such analysis allows us to extract some parameters that are then used in our difficulty prediction models. In this subsection, we analyse the accuracy of those models, comparing the predicted failure probabilities with the effective obtained values. The described models define the probabilities of success and failure sustained by two important dimensions: space and time. Those can be combined in order to obtain alternative challenge types (for instance moving platforms, in which space varies over time). Therefore, space and time are the two main aspects that will be analysed.

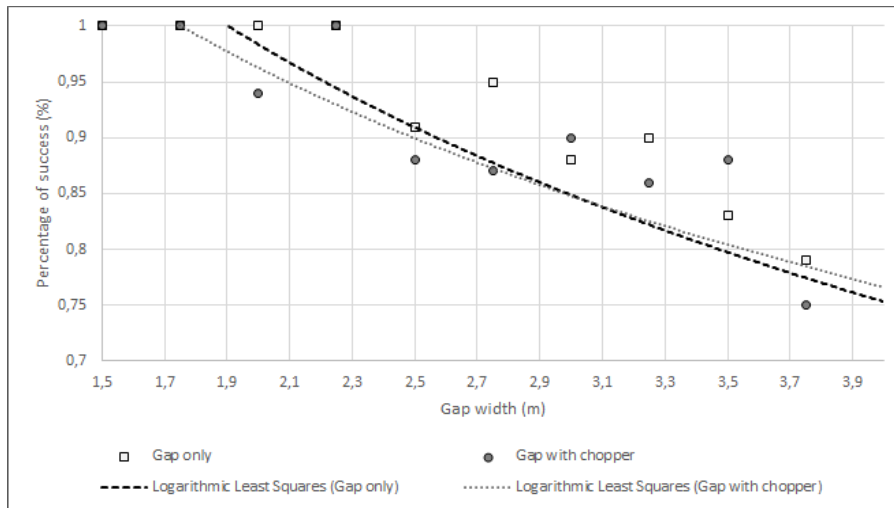


Figure 8.20 – Comparison of the percentages of failure using a gap challenge individually and in combination with a time-based challenge.

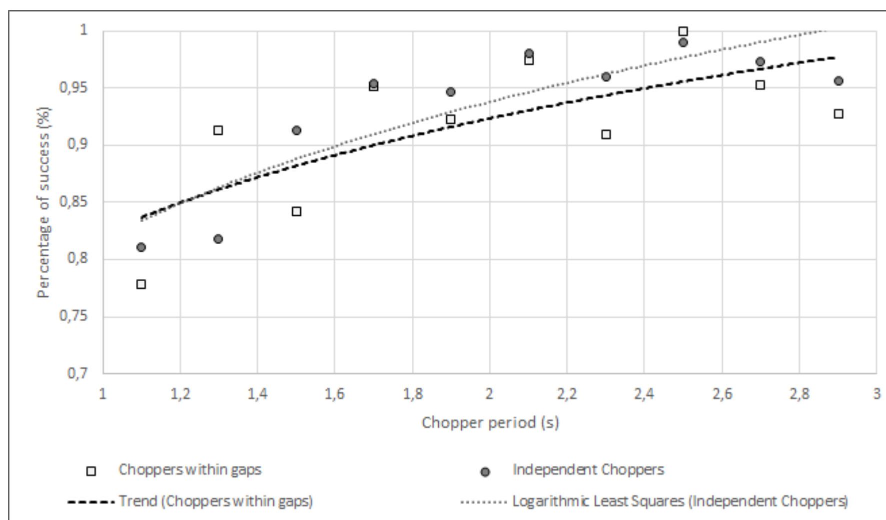


Figure 8.21 – Comparison of the percentages of failure using a time-based challenge individually and in combination with a spatial challenge.

Time-Based Challenges

We will start by examining the difficulty estimator for time-based challenges, as these have a more straightforward interpretation. While time-based challenges have only one independent variable (t), spatial challenges have two (gap displacement in x and y). In Table 8.1, we present the estimated and the measured difficulty in such type of challenges for different attack times. The measured probabilities of success are grouped within an arbitrary set of intervals of equal size, and the estimated probabilities consider the central point of the referred intervals.

The data of that table is depicted as a chart, in Figure 8.22. It is possible to observe that the metric presents values that are near the effective measured probabilities of success. For the majority of the cases, the predicted values do not differ more than 2% in relation to the obtained probabilities, resulting in an average error of 2.4%. The less accurate prediction can be observed for periods within the interval [1.2; 1.4[, where we have registered an error of about 10%. It is still important to notice that this estimator is defined for the overall population, using the overall mean and standard deviation, where the individual players' profiles are not considered. Taking that into account and recalling the fact that the overall population does not follow a normal distribution, such precision should be

considered a valid approximation. Moreover, individual models can be defined for each player, using their specific mean value and the corresponding standard deviation (which follow a normal distribution, as we have observed in the previous section). The individual models should provide more accurate error predictions, even though the size of our sample does not allows confirming it. As the probabilities of success are being estimated with an error around 2%, such confirmation requires having, for each of the individually analysed player, a sample that allows measuring the real probability of success at least with that precision. Preferably, this precision should be higher. For instance, in order to measure the probability of success with a precision of 1%, one has to analyse 100 attempts for each of the defined intervals, resulting in a total of 1000 attempts from each player. Having a set of players performing that number of challenge attempts, with the desired commitment and endeavoured to succeed, requires an experiment with much larger dimensions, preferentially in a popular game.

Time-window (seconds)	Estimated probability of success	Measured probability of success
[1.0; 1.2[	0.835	0.811
[1.2; 1.4[	0.894	0.818
[1.4; 1.6[	0.923	0.913
[1.6; 1.8[	0.939	0.953
[1.8; 2.0[	0.948	0.946
[2.0; 2.2[	0.953	0.981
[2.2; 2.4[	0.957	0.96
[2.4; 2.6[	0.959	0.99
[2.6; 2.8[	0.961	0.973
[2.8; 3.0[	0.963	0.957

Table 8.1 – Probabilities of success (estimated and measured) for time-based challenges.

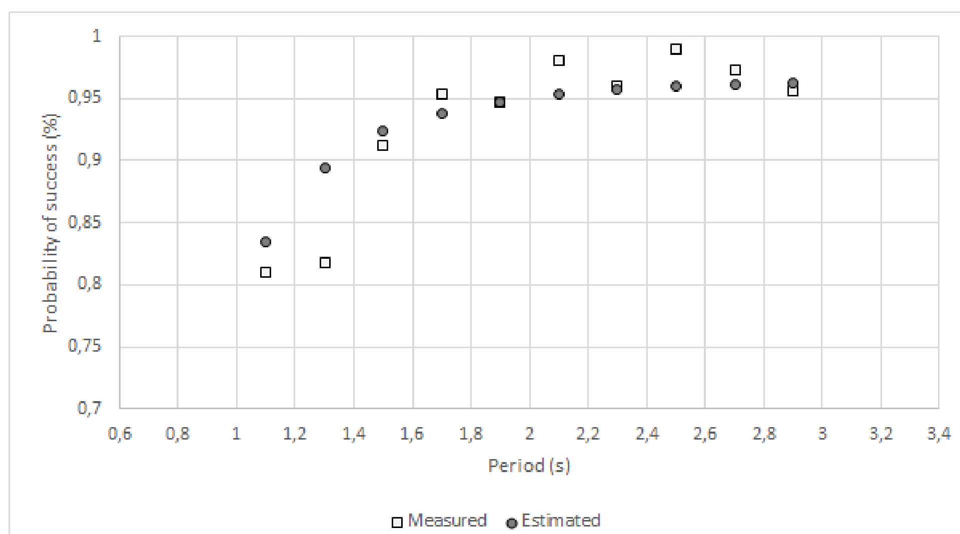


Figure 8.22 – Probabilities of success (estimated and measured) for time-based challenges.

Spatial Challenges

Regarding jumps over gaps, we have proposed two different estimators, back in subsection 4.4.2, one based on an error margin and another one based on a parametrised distribution model according to the players' strategies. In Table 8.2, we present the predicted probabilities of success with both estimators (P_{em} refers to the estimator based on an error margin and P_{dm} refers to the estimator based on the jump distribution model) for different gap displacements, in conjunction with the effective measured probabilities (P_m).

Vertical difference of the gap, g_v (m)	Width / Height	Horizontal distance of the gap, g_h (m)					
		1,5	**2**	**2,5**	**3**	**3,5**	**4**
	-1	$P_{em}=0.97$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.96$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.95$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.93$ $P_{dm}=0.82$ $P_m=0.89$	$P_{em}=0.91$ $P_{dm}=0.82$ $P_m=0.78$	$P_{em}=0.89$ $P_{dm}=0.82$ $P_m=0.91$
	-0,75	$P_{em}=0.97$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.96$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.94$ $P_{dm}=0.82$ $P_m=0.87$	$P_{em}=0.93$ $P_{dm}=0.82$ $P_m=0.93$	$P_{em}=0.91$ $P_{dm}=0.82$ $P_m=0.91$	$P_{em}=0.88$ $P_{dm}=0.82$ $P_m=0.92$
	-0,5	$P_{em}=0.97$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.96$ $P_{dm}=0.82$ $P_m=0.79$	$P_{em}=0.94$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.92$ $P_{dm}=0.82$ $P_m=0.88$	$P_{em}=0.9$ $P_{dm}=0.82$ $P_m=0.86$	$P_{em}=0.88$ $P_{dm}=0.82$ $P_m=0.84$
	-0,25	$P_{em}=0.97$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.95$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.94$ $P_{dm}=0.82$ $P_m=0.73$	$P_{em}=0.92$ $P_{dm}=0.82$ $P_m=0.92$	$P_{em}=0.9$ $P_{dm}=0.82$ $P_m=0.88$	$P_{em}=0.85$ $P_{dm}=0.82$ $P_m=0.87$
	0	$P_{em}=0.95$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.95$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.93$ $P_{dm}=0.82$ $P_m=0.93$	$P_{em}=0.91$ $P_{dm}=0.82$ $P_m=0.89$	$P_{em}=0.87$ $P_{dm}=0.82$ $P_m=0.81$	$P_{em}=0.81$ $P_{dm}=0.82$ $P_m=0.56$
	0,25	$P_{em}=0.92$ $P_{dm}=0.82$ $P_m=1$	$P_{em}=0.92$ $P_{dm}=0.82$ $P_m=0.86$	$P_{em}=0.91$ $P_{dm}=0.82$ $P_m=0.91$	$P_{em}=0.88$ $P_{dm}=0.82$ $P_m=0.83$	$P_{em}=0.84$ $P_{dm}=0.82$ $P_m=0.82$	$P_{em}=0.76$ $P_{dm}=0.77$ $P_m=0.47$
	0,5	$P_{em}=0.88$ $P_{dm}=0.82$ $P_m=0.83$	$P_{em}=0.89$ $P_{dm}=0.82$ $P_m=0.89$	$P_{em}=0.87$ $P_{dm}=0.82$ $P_m=0.94$	$P_{em}=0.84$ $P_{dm}=0.82$ $P_m=0.91$	$P_{em}=0.79$ $P_{dm}=0.81$ $P_m=0.9$	$P_{em}=0.69$ $P_{dm}=0.55$ $P_m=0.17$
	0,75	$P_{em}=0.83$ $P_{dm}=0.82$ $P_m=0.9$	$P_{em}=0.84$ $P_{dm}=0.82$ $P_m=0.71$	$P_{em}=0.83$ $P_{dm}=0.82$ $P_m=0.89$	$P_{em}=0.79$ $P_{dm}=0.82$ $P_m=0.66$	$P_{em}=0.71$ $P_{dm}=0.69$ $P_m=0.73$	$P_{em}=0.49$ $P_{dm}=0.08$ $P_m=0.13$
	1	$P_{em}=0.69$ $P_{dm}=0.82$ $P_m=0.66$	$P_{em}=0.77$ $P_{dm}=0.82$ $P_m=0.77$	$P_{em}=0.76$ $P_{dm}=0.82$ $P_m=0.23$	$P_{em}=0.7$ $P_{dm}=0.71$ $P_m=0.09$	$P_{em}=0.51$ $P_{dm}=0.13$ $P_m=0.07$	$P_{em}=0$ $P_{dm}=0$ $P_m=0$

Table 8.2 – Probabilities of failure (estimated and measured) for a gap challenge.

The overall behaviour of the effective measured data can be observed in Figure 8.23. A steady pattern can be observed for the easiest cases, where the platforms are more proximate, presenting values over 80% of success. As the displacements increase, the probabilities of success start to decrease gradually and, as we move towards extreme cases in either of the dimensions, the probabilities decay more rapidly, converging to low values. Both the proposed estimators can represent this type of behaviour, as it is possible to observe in Figure 8.24, where we present the graphical depictions of such estimators. Regarding the global accuracy, both present similar values. The average absolute

error with the estimator based on an error margin is 9.9%, whereas the estimator based on the distribution model presents an absolute average error of 12.4%. The main issue in the later lays on the cases regarding jumps between nearby platforms. As the estimator uses the overall distribution, which has a wide standard deviation than spans different player profiles, there is a tendency to deflate the probabilities of success. That effect is more noticeable in the referred cases of nearby platforms. However, in the remaining cases, this approximation tends to be more effective.

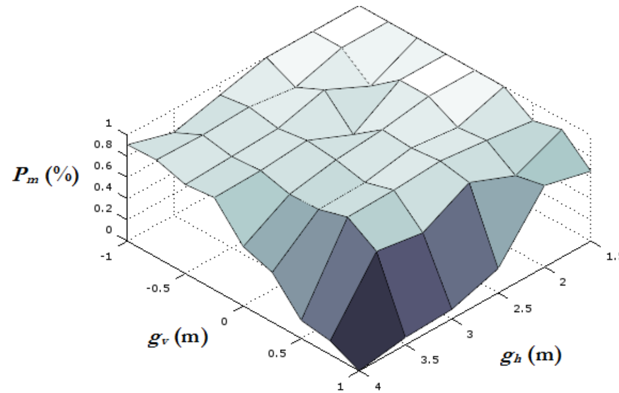


Figure 8.23 – Measured probabilities of success for gaps with different values of x and y.

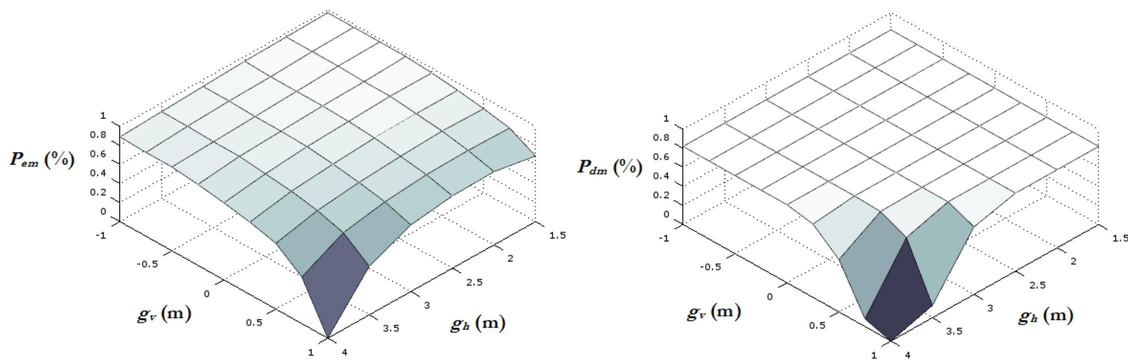


Figure 8.24 – Estimated probabilities of success for gaps with different values of x and y (at the left, estimator based on error margins and, at the right, estimator based on the distribution model).

In addition, similarly to the case of time-based challenges, considering the individual distributions of each player, in order to build their personalised estimators, the predictions should be more accurate, even though the size of our sample does not allow us to confirm that. In this case, the numerous combinations between horizontal and vertical gap distances make the requirements of such confirmation even higher than in the previous case. For instance, estimating difficulty with 2% precision requires 50 attempts for each of the combinations, resulting in 2700 jumps per player. Obtaining such sample requires devoted players frequently playing the game over time.

8.3.3. Level Evaluation

Lastly, the implemented experiment allows us to assess the level value using different approaches. We intend to retrieve effective arguments for the following questions in the context of this work:

- Do users prefer humanly authored or procedurally generated levels?
- Can the users tell apart the levels created by humans from the levels generated procedurally?
- What is the potential of the quest-based generation, regarding design time, overall level quality and design controllability?

Level Quality

One first important aspect that we want to understand is if procedural generated levels can match the quality of humanly designed content. As we have seen in section 8.2, at the end of each level, the users were asked to rate it in a scale from 1 to 5. In Table 8.3, we sum up the values of such process, showing the differences for each individual design method. The data was obtained from a total of 1123 classifications. Those referred to levels created by 6 designers, 9 experienced non-designers and 6 inexperienced non-designers.

<table>
<tr><th colspan="2" rowspan="2">Design method (approach and profile)</th><th colspan="2">Level evaluation</th></tr>
<tr><th>Mean, μ</th><th>Std. deviation, σ</th></tr>
<tr><td colspan="2">Procedural</td><td>3.474</td><td>1.149</td></tr>
<tr><td rowspan="3">Quest Editor</td><td>Game-designer</td><td>3.442</td><td>1.127</td></tr>
<tr><td>Experienced non-designer</td><td>3.433</td><td>1.146</td></tr>
<tr><td>Inexperienced non-designer</td><td>3.214</td><td>1.423</td></tr>
<tr><td rowspan="3">Platform Level Editor</td><td>Game-designer</td><td>3.707</td><td>1.062</td></tr>
<tr><td>Experienced non-designer</td><td>3.428</td><td>1.147</td></tr>
<tr><td>Inexperienced non-designer</td><td>2.722</td><td>1.407</td></tr>
</table>

Table 8.3 – Level evaluation for different authoring approaches and profiles.

Regarding the human process of level design, it is possible to notice differences in the average level rates for the different profiles. In this aspect, the game designers are the preferred level creators, as their levels have higher rates than the levels created by the other users. It is also possible to notice that the experience is an important factor in the process. The users with previous experience in edition tasks present a higher classification value in comparison to the users that do not have such type of experience. The later are also more inconsistent, presenting higher deviations for both editing approaches.

The results regarding the levels created with the *Quest Editor* are more alike amongst the different profiles. The overall average rates represent satisfactory values, matching the edition capabilities of a non-designer with experience in content creation. In particular, such type of approach is useful for beginners in such type of process. While these users created levels with an average rate of 2.72 with the classic level editor, they achieved an average of 3.21 with the *Quest Editor*. However, for game designers, the single use of this type of approach limits their creativity and the results are not as good as with the *Platform Level Editor* (mean value of 3.44 vs. 3.71).

The implemented procedural generator creates levels with a satisfying average rating, yet not as high as the average ratings of the levels created by the designers (mean value of 3.47 vs. 3.71). In order to understand the real differences among the possible alternatives, we have performed a one-way ANOVA followed by a Tukey's test to verify the effective differences between the calculated values. Considering a 95% confidence interval, different groups shall present a *p*-value under 0.05 to be considered significantly different. This is the case for the three different human author profiles. The mean regarding experienced non-designers present a *p*-value of 0.01 in comparison with designers, and a *p*-value of 0 in comparison with inexperienced non-designers. Relating the two opposite sites, designers and inexperienced non-designers, results in a *p*-value of 0. Therefore, it is valid to state that

the three established profiles have a statistically significant difference. However, such type of conclusion is not possible concerning the procedural generated levels. Comparing the corresponding mean results in *p*-values of 0, 0.984 and 0.198 for inexperienced non-designers, experienced non-designers and game designers, respectively. Consequently, the ratings of our procedural levels have statistically significant differences only when compared to the levels created by inexperienced non-designers. In comparison to experienced non-designers and designers, the differences are not statistically significant, showing that the generator presents similar ratings to these profiles.

The same test was also performed to compare the differences for the various profiles using the *Quest Editor* and none of the groups can be considered significantly different as all obtained *p*-values are over 0.05. This reinforces that the semi-automatic generation approach, based on quests, flattens the different profiles regarding the quality of the levels.

Level Believability
Another aspect that we believe it is important to understand is if the users can (easily) distinguish humanly designed levels from the ones generated automatically. In the described experience, the players were asked to speculate about who created the levels after they played them. They could estimate if the level was created by a human or by a computer algorithm, or present a neutral answer stating that the level does not provide evidences to any side. In Table 8.4, we present the percentages of each response for the different generation methods.

In the obtained data, it is noticeable that the answers present a considerable confusion in the perception of the level creation process, which favours the hypothesis that the users cannot easily distinguish humanly created levels from the ones generated automatically. It is possible to notice a trend for the users to think that a level was generated procedurally. This might be due to the context of the project, in which it is known that some of the levels are certainly generated with such approach. However, when the levels are indeed generated procedurally, the users are more likely to think the opposite, reinforcing the randomness of the answers.

Classification / Generation method	Human	Untraceable	Procedural
Created with the *Platform Level Editor*	21.34%	23.43%	55.23%
Created with the *Quest Editor*	11.76%	22.06%	66.18%
Procedural	31.82%	27.27%	40.91%

Table 8.4 – Users' speculations regarding the level creation method (humanly created or procedurally generated).

In addition, we have performed a one-way ANOVA followed by a Tukey's test to verify the effective differences among the calculated values, considering the classification as a quantitative variable. None of the means is significantly different from the others (all *p*-values in the comparisons are over 0.05), so it is not possible to separate any of the previous groups.

Applicability of the Quest-Based Approach
Moreover, it is important to understand the usefulness of a semi-automatic approach such as the quest-based generation. This type of approach is intended as a compromise between level controllability and design time. Table 8.5 shows the registered required time for level creation by the individuals of our sample.

One first interesting aspect that can be observed in the table is that, for game designers, the process of level creation takes more time than for non-designers. This aspect is especially noticeable in comparison to those who do not have prior experience in content creation. While these users tend to simply define a path that makes sense with a random sequence of challenges, a designer takes into account the aesthetic aspects of the level, in particular with the placement of decorative entities. Users with experience in content edition tend to think more like a designer and look at the process in a similar way, even though they present minor efforts in level perfecting. Naturally, these specificities have repercussions in the level quality, as we have seen previously.

Approach/Task Profile	***Platform Level Editor*** **Sketch (min)**	***Platform Level Editor*** **Perfecting (min)**	***Quest Editor*** **(min)**
Game-designer	$\mu = 14.5$ $\sigma = 3.1$	$\mu = 26.83$ $\sigma = 16.08$	$\mu = 8.5$ $\sigma = 5.44$
Experienced non-designer	$\mu = 15.13$ $\sigma = 9.23$	$\mu = 7$ $\sigma = 4.64$	$\mu = 5.5$ $\sigma = 1.12$
Inexperienced non-designer	$\mu = 6.33$ $\sigma = 3.73$	$\mu = 8.5$ $\sigma = 5.5$	$\mu = 8$ $\sigma = 4.97$

Table 8.5 – Level creation times using the *Platform Level Editor* and the *Quest Editor* (mean – μ, and standard deviation – σ).

Secondly, it can be confirmed that the process of level creation using the quest-based generator is faster in relation to the common edition mechanism. From the perspective of the game designer and the experienced non-designers, this approach has potential for the faster creation of level sketches. Regarding the inexperienced non-designers, the creation process has balanced required times, so the main potential of this approach is the construction of better levels without requiring design skills. As we have seen previously, these users created levels with higher ratings using this method (mean value of 3.21 vs. 2.72).

8.4. Concluding Remarks

In this chapter, we have presented our final evaluation study about the different metrics, algorithms and approaches that we propose in this dissertation. Using the architecture that we have described in chapter 5, we have implemented a game that gathers gameplay information from the users while playing levels that might be generated by a human or by an automatic generation process. Humanly authored levels are divided in three main profiles, namely designers, non-designers with experience in content creation and non-designers without experience in content creation. Two different types of authoring processes were also considered. One is the classical method of level editing, using an adapted version of the *Platform Level Editor* that we have presented in section 5.4. The other is based on the semi-automatic generation process after an initial definition of a set of quests that should be comprised by the level, as described in section 7.7. The procedural generated levels use the graph principles presented in chapter 6, and the algorithms presented in sections 7.3 and 7.4. Furthermore, we used the gathered data to assess the capabilities of the proposed estimators for difficulty, based on probabilities of success and failure, as proposed in section 4.4.

First, it was particularly interesting to verify that the users had difficulties in distinguishing our procedural generated levels from those created by humans. After finishing a level, either by completing it or by failing a challenge, the game asked the users to wonder if a human or a computer algorithm created the level. The answers exhibited high randomness without any particular trend, denoting the

fact that the users could not find any particular flaws or patterns in the generation algorithms to easily identify procedural generated levels.

Secondly, they rated highly the levels generated procedurally, similarly to the levels created by non-designers with experience in content creation tasks and those created by game designers. The later still presented levels with a higher average in comparison to those generated procedurally, but the applied statistical tests sustain the hypothesis that they do not present significant differences. The population of procedural levels is not statistically different from the population composed by the levels created by designers, regarding the obtained classifications.

Finally, concerning our difficulty estimators, it was possible to observe that the predicted values for success and failure were approximate to the effective rates. The two proposed estimators, one based on the concept of margin of error and the other one based on distribution models, show similar overall values. Regarding jumps over gaps, both allow predicting the probabilities of success and failure in challenges with an average error margin around 10%. Whereas the estimator established after a margin of error requires an *ad-hoc* parametrisation of an exponential factor, the estimator defined with a distribution model presents a more efficient tuning mechanism, consisting in setting the distribution variables (mean and standard deviation) following the effective measured data. As we presented in subsection 8.3.2, we believe that tuning the estimator individually for each player using his/her specific mean and standard deviation may improve the error margin, even though such confirmation requires an extensive gameplay analysis with players committed to the game over time. Another advantage of the estimator based on a distribution model it that the principles are applicable to other types of challenges. Also in subsection 8.3.2, we have seen that, for time-based challenges, it allows predicting success and failure as a function of the time windows represented in those challenges, with small error margins of only about 2%.

9. Conclusions and Future Work

"A PCG-based game is one in which the underlying PCG system is so inextricably tied to the mechanics of the game, and has so greatly influenced the aesthetics of the game, that the dynamics – player strategies and emergent behaviour – revolve around it."

(Smith, Gan, et al., 2011)

9.1. Synthesis

This dissertation aimed to expand the boundaries of the research about automatic level generation for platform videogames. This final chapter synthesises the current work and sums up the main conclusions of this research. In addition, we raise new questions for further developments and unveil new challenges to be addressed within the topic.

Computer games gained a significant role in quotidian life in which a large market emerged, shared by the most diverse type of development models, from individual game developers to studios with staffs that comprise hundreds of people, spanning all types of expertise areas. Contemporary game development may include analysts, designers, developers, testers, project managers and also several less common software personnel like writers, artists, and musicians (McConnell, 2001). All this intense mass production brings the risk of a market collapse, bringing into the memory the videogame crash that occurred back in 1983. To avoid this risk, different lines of investigation present us continuous search for new types of experiences, focused on the user and directed to the user experience. Moreover, it is important to recall that videogames are inherent learning tools (Koster, 2004), which can serve not only entertainment purposes but also serious aspects of different domains.

The transcription of videogames into engineering concepts is a task that is often not trivial. While classic game theory provide interesting algorithms to study mathematical models of conflict and cooperation between intelligent rational decision makers (Myerson, 1997), action videogames are still out of this domain, as they do not present trivial logical rules plausible to be interpreted as mathematical searches for optimal results. Several other notions have to be taken into account, such as aesthetic perceptions and users' emotion models, among several others. In that way, game research is increasingly becoming "a discipline in its own right, with several different internal research stances"(Björk, 2008).

PCG is an active research topic with interesting factors for different aspects of videogame development. On the one hand, this is a useful approach for small development teams and independent developers to overcome the issue of lack of resources in the global market of mass-production game-design. On the other hand, the process of automatically creating a level potentiates the creation of experiences centred in the player, an interesting approach towards personalisation.

This specific work applies PCG techniques designed for the automatic level creation for platform videogames. An important part of our studies aimed to improve the bridging between different abstract layers of game content. For that purpose, we have related the algorithms for game space generation with contextual information in which those spaces are intended to be used, as a background

story or a set of missions and quests. In a *platformer*, levels represent quests in hero-centred stories, where a set of actions are taken depicting those quests. Thus, the automatic level generation algorithms should consider a bridging mechanism between these notions and the geometry that is ought to be created. Following Togelius' overview about the future of PCG research (Togelius et al., 2013), our work goes towards the goal of multi-level multi-content PCG, addressing the generation of "multiple types of quality content at multiple levels of granularity". In addition, this idea goes towards some of the principles that Smith has proposed for the future of PCG in games (Smith, 2014a). She emphasises that we should "pay more attention to both the designer and the design process when creating systems, rather than relying on design knowledge to be embedded in building blocks or applied as an evaluation function on an otherwise design-blind process".

Furthermore, we have implemented different generation algorithms, focusing different goals in the process of level generation, each with distinct advantages. Namely, we have implemented a generator where we mapped global design heuristics into a genetic algorithm, developed an automatic chunk-detection generator for open scenarios, and created an adaptation mechanism to provide content richness and the creation of composite design patterns.

Moreover, creating levels automatically strengthens the generation of content that is specific for a certain player's profile. In that context, we have explored different human factors related to gameplay and identified the need of understanding and quantifying the notion of difficulty. We have defined and tested different difficulty estimators for distinct situations that one can come across in action videogames. Existing game levels were analysed in order to perceive the empiric notion that gameplay becomes gradually more difficult in succeeding levels. Furthermore, we gathered data from multiple playing sessions, which allowed testing the estimated probabilities of success in comparison to its effective values.

Integrating the previous concepts, we projected a global architecture for a PCG system in which a generic level representation language was created. Content from different platform videogames can be described using this framework. This served as a base for the implementation of a level editor in which it is possible to edit and analyse levels of multiple existing games and in which it is also possible to test different generation algorithms.

9.2. Achievements

In the end, our main achievements can be summed up as follows:

- Identification of a set of quest patterns for the specific genre *platformer* (Mourato et al., 2013a), presented in section 3.3.
- Definition of a framework to represent content in platform videogames, applied to a level editor and used in multiple automatic generation algorithms (Mourato et al., 2012b, 2012c), as described in chapter 5.
- Definition of difficulty models for specific challenges, allowing the estimation of difficulty based on general entity features (Mourato et al., 2014; Mourato & Próspero dos Santos, 2010). These contributions were addressed in section 4.4.
- Creation of a tagging mechanism for the graph representation of game levels, depicting the roles of each path segment (Mourato et al., 2012a, 2013b), described in section 6.4.
- Development of three different types of level generators, with different objectives, namely:

- An overall structure generator, in which design heuristics were embedded within a genetic algorithm (Mourato et al., 2011), described in section 7.3.
- A content adaptation algorithm, based on the referred graph tagging mechanism, capable of perfecting level content (Mourato et al., 2012a, 2013b), explained in section 7.4.
- A sequential level generator, which takes a set of existing levels as input and provides different variants for such levels (Mourato et al., 2013b), described in section 7.5.

- A semi-automatic level generator, presented in section 7.7, using an author-centric approach in which levels are created after the definition of a sequence of desired quest design patterns, explained in subsection 3.3.4 and further expanded in section 7.6. This application allows the integration of the different implemented algorithms and proves the potential of bridging different layers of a quest hierarchy.

9.3. Further Developments

The achievements presented in the previous section represent a stable state in our research. Still, the accomplished results open new doors for additional investigation.

As the bridging mechanism for a semi-automatic generation that was presented shows the potential of integrating multiple levels of abstraction, an interesting point for further research is to expand this type of integration in order to apply automatic generators for quests. This topic has been addressed recently (Lee & Cho, 2012; Lima, Feijó, & Furtado, 2014) and its possible integration with lower abstraction principles goes towards the objective of multi-level multi-content PCG, in which fully automated generation processes are decomposed into multiple levels of abstraction.

One additional interesting step is the research towards PCG-based game design, one major goal of the topic, aiming to make the rules part of the automatic generation process. As the presented approach for gameplay and level analysis is based on general representations, in the form of graphs, independent of the game, further studies might aim to generate the graph rules automatically and the corresponding physical constraints within the game engine.

Moreover, another major goal of PCG research is the automatic generation of complete games. Merging the two previous lines of further study, integrating automatic generation of quests and game rules, is a route towards the automatic generation of complete games.

9.4. Concluding Statement

Summing the different progresses that we have presented, we can state that we have accomplished our research goals. The final experiment, presented in chapter 8, demonstrated the viability of such approach for small teams and independent developers, spanning different aspects, from a design perspective to a gameplay standpoint.

The generation algorithms that we have developed include new features and design patterns that were not addressed in prior alternatives, making them suitable for platform games with an exploratory theme. Despite the fact that these differences do not allow direct comparisons with other approaches, we have verified that the algorithms produce levels that can impersonate humanly designed levels and with quality that can arguably dispute content created by game designers or users with experience in content creation tasks.

Our metrics for estimating difficulty provide a novel mechanism in the topic, expanding the existing proposed metrics for gaps in platform videogames and evidencing applicability for other types of

challenges. The examples of time-based challenges and moving platforms show that using predictions based on distributions built according to the players' behaviours is a valid way to perceive probabilities of success and failure. Moreover, is opens prospects of similar analysis for other types of challenges, inclusively in other genres.

The implemented *Quest Editor* is a concretisation of our ideas regarding the integration of content in different abstraction layers. A semi-automatic generation process based on quests is as an interesting approach to bridge story related concepts with structural generation algorithms, with potential in the generation of varied content and to speed up human authored creation.

Finally, observing the path taken since the early days of initial ideas and prospects to the obtained results, it is rewarding to verify that we have fulfilled out main objectives. With the novel aspects brought to the automatic level generation for platform videogames and the different contributions to the topic, we finish this dissertation with a sense of gratification.

A. *Platform Level Editor* - Implementation Details

A.1. Introduction

This appendix presents additional implementation details about our *Platform Level Editor*, presented in section 5.4.

A.2. Generator Plugin Interface (C#)

```
public interface IAlgorithm
{
    /// <summary>
    /// Processes the generation process.
    /// </summary>
    /// <param name="levels">
    ///  A set of levels, obtained from a previous algorithm
    /// (can be empty, if this algorithm is the 1st of the list).
    /// </param>
    /// <returns>
    ///  A set of generated levels.
    /// </returns>
    Level[] Generate(Level[] levels);


    /// <summary>
    /// Calculate the number of levels that
    /// the generator will produce.
    /// Notice that the number of levels can be a
    /// parameter, thus this number is not fixed.
    /// </summary>
    /// <returns>
    /// The number of levels to generate with
    /// the current configuration.
    /// </returns>
    int CalculateOutputSize();


    /// <summary>
    /// Calculate the number of levels that
    /// the generator needs as input for
    /// the current configuration.
    /// </summary>
    /// <returns>
    /// The required input size for the
    /// current configuration.
    /// </returns>
    int CalculateInputNeeded();


    /// <summary>
    /// The name of the algorithm.
    /// </summary>
    string Name { get; }

}
```

A.3. Game Plugin Interface (C#)

```
    public interface IPlugin
    {

        /// <summary>
        /// Exports a level into the game.
        /// </summary>
        /// <param name="level">The level to export to the game.</param>
        void Export(Level level);


        /// <summary>
        /// Import a level from to the game to the level editor.
        /// Can provide an interface to select the level.
        ///
        /// Should call this method to update the level in the editor:
        /// PluginUtil.MainForm.PluginUpdateLevel (Level level)
        /// </summary>
        void Import();


        /// <summary>
        /// Gets the game ontology.
        /// </summary>
        /// <returns>The name (and path) of the XML file containing
        /// the game ontology</returns>
        string GetCellSet();


        /// <summary>
        /// Launches the game with the current selected level.
        ///
        /// If needed, call Export to send the level to the game.
        /// If needed, access PluginUtil.MainForm.CurrentLevel to obtain
        /// the level that is currently selected within the editor.
        /// </summary>
        void Play();

    }
```

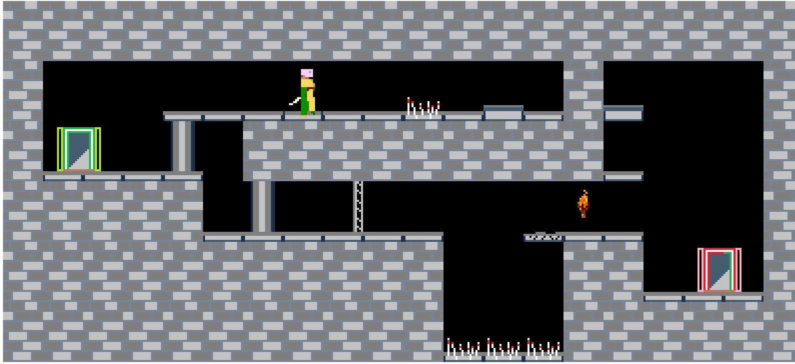
A.4. Game Ontology XML – Example for *Prince of Persia*

```
<CellSet>

  <Images ThumbnailWidth="20" ThumbnailHeight="30" />

  <Blocks>
    <Block Name="Normal Cells" Parent="" Path="Normal Cells" Image="0" />
    <Block Name="Wall" Parent="Normal Cells" Path="Normal Cells\Wall" Image="3" />
    <Block Name="Empty" Parent="Normal Cells" Path="Normal Cells\Empty" Image="1" />
    <Block Name="Floor" Parent="Normal Cells" Path="Normal Cells\Floor" Image="2" />
    <Block Name="Landable" Parent="Normal Cells\Floor"
       Path="Normal Cells\Floor\Landable" Image="2" />
    <Block Name="Big Pillar" Parent="Normal Cells\Empty"
       Path="Normal Cells\Empty\Big Pillar" Image="13" />
    <Block Name="Tapestry" Parent="Normal Cells\Wall"
    Path="Normal Cells\Wall\Tapestry"
    Image="20" />
    <Block Name="Mirror" Parent="Normal Cells\Wall"
    Path="Normal Cells\Wall\Mirror" Image="10" />
    <Block Name="Lattice Small" Parent="Normal Cells\Empty"
       Path="Normal Cells\Empty\Lattice Small" Image="27" />
```

```
  <Block Name="Lattice Right" Parent="Normal Cells\Empty"
     Path="Normal Cells\Empty\Lattice Right" Image="26" />
  <Block Name="Lattice Left" Parent="Normal Cells\Empty"
     Path="Normal Cells\Empty\Lattice Left" Image="24" />
  <Block Name="Lattice Support" Parent="Normal Cells\Empty"
     Path="Normal Cells\Empty\Lattice Support" Image="28" />
  <Block Name="Tapestry Top" Parent="Normal Cells\Wall"
     Path="Normal Cells\Wall\Tapestry Top" Image="29" />
  <Block Name="Unlandable" Parent="Normal Cells\Floor"
     Path="Normal Cells\Floor\Unlandable" Image="2" />
  <Block Name="Floor" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Floor" Image="2" />
  <Block Name="Balcony Right" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Balcony Right" Image="23" />
  <Block Name="Lattice Pillar" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Lattice Pillar" Image="25" />
  <Block Name="Balcony Left" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Balcony Left" Image="22" />
  <Block Name="Big Pillar" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Big Pillar" Image="12" />
  <Block Name="Torch" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Torch" Image="21" />
  <Block Name="Sword" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Sword" Image="19" />
  <Block Name="Skeleton" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Skeleton" Image="15" />
  <Block Name="Potion" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Potion" Image="14" />
  <Block Name="Spikes" Parent="Normal Cells\Floor\Unlandable"
     Path="Normal Cells\Floor\Unlandable\Spikes" Image="16" />
  <Block Name="Pillar" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Pillar" Image="11" />
  <Block Name="Chopper" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Chopper" Image="4" />
  <Block Name="Gate" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Gate" Image="8" />
  <Block Name="ExitL" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\ExitL" Image="5" />
  <Block Name="StartR" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\StartR" Image="18" />
  <Block Name="StartL" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\StartL" Image="17" />
  <Block Name="ExitR" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\ExitR" Image="6" />
  <Block Name="Guard" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Guard" Image="7" />
  <Block Name="Breakable" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Breakable" Image="9" />
  <Block Name="Button" Parent="Normal Cells\Floor\Landable"
     Path="Normal Cells\Floor\Landable\Button" Image="30" />
  <Block Name="Drop" Parent="Normal Cells\Floor\Landable\Button"
     Path="Normal Cells\Floor\Landable\Button\Drop" Image="31" />
  <Block Name="Raise" Parent="Normal Cells\Floor\Landable\Button"
     Path="Normal Cells\Floor\Landable\Button\Raise" Image="30" />
</Blocks>

<Groups>
  <Group Name="Start" Width="2" Height="1"
     Blocks="Normal Cells\Floor\Landable\StartL,Normal Cells\Floor\Landable\StartR"
     MinWidth="2" MinHeight="1" MaxWidth="2" MaxHeight="1" />
  <Group Name="End" Width="2" Height="1"
     Blocks="Normal Cells\Floor\Landable\ExitL,Normal Cells\Floor\Landable\ExitR"
     MinWidth="2" MinHeight="1" MaxWidth="2" MaxHeight="1" />
</Groups>
```

```
  <Categories>
    <Category Name="Floor" Image="2" Expression="&quot;Normal Cells\Floor&quot;" />
    <Category Name="Empty" Image="1" Expression="&quot;Normal Cells\Empty&quot; |
       &quot;Normal Cells\Floor\Landable\Breakable&quot;" />
    <Category Name="Wall" Image="3" Expression="&quot;Normal Cells\Wall&quot;" />
    <Category Name="Stable Floor" Image="2" Expression="&quot;Normal
       Cells\Floor&quot; &amp; !&quot;Normal Cells\Floor\Landable\Breakable&quot;" />
    <Category Name="Landable Floor" Image="2"
       Expression="&quot;Normal Cells\Floor\Landable&quot;" />
    <Category Name="Breakable Floor" Image="9"
       Expression="&quot;Normal Cells\Floor\Landable\Breakable&quot;" />
  </Categories>

  <MovementAids>
    <MovementAid Name="Basic Step" GraphEntries="0,0,1,0,1&#xD;&#xA;1,0,0,0,1"
       PatternRows="1" PatternColumns="2" Pattern="Floor,Floor" />
    <MovementAid Name="Jump 1" GraphEntries="0,0,2,0,5&#xD;&#xA;2,0,0,0,5"
       PatternRows="1" PatternColumns="3" Pattern="Floor,Empty,Floor" />
    <MovementAid Name="Jump 2" GraphEntries="0,0,3,0,10&#xD;&#xA;3,0,0,0,10"
       PatternRows="1" PatternColumns="4" Pattern="Floor,Empty,Empty,Floor" />
    <MovementAid Name="Jump 3" GraphEntries="0,0,4,0,20&#xD;&#xA;4,0,0,0,20"
       PatternRows="1" PatternColumns="5" Pattern="Floor,Empty,Empty,Empty,Floor" />
    <MovementAid Name="DropRight" GraphEntries="0,0,1,1,1&#xD;&#xA;1,1,0,0,1"
       PatternRows="2" PatternColumns="2"
       Pattern="Stable Floor,Empty,,Stable Floor" />
    <MovementAid Name="Drop Left"
       GraphEntries="1, 0, 0, 1, 1&#xA;&#xD;&#xA;0, 1, 1, 0, 1&#xA;"
       PatternRows="2" PatternColumns="2"
       Pattern="Empty,Stable Floor,Stable Floor," />
    <MovementAid Name="Vertical Right"
       GraphEntries="0, 0, 0, 1, 3&#xD;&#xA;0, 1, 0, 0, 3" PatternRows="2"
       PatternColumns="2" Pattern="Floor,Empty,Floor,Empty" />
    <MovementAid Name="Vertical Left"
       GraphEntries="1, 0, 1, 1, 3&#xD;&#xA;1, 1, 1, 0, 3&#xD;&#xA;"
       PatternRows="2" PatternColumns="2" Pattern="Empty,Floor,Empty,Floor" />
    <MovementAid Name="Jump 4 Right" GraphEntries="1,0,6,0,50" PatternRows="1"
       PatternColumns="7"
       Pattern="Floor,Floor,Empty,Empty,Empty,Empty,Stable Floor" />
    <MovementAid Name="Jump 4 Left" GraphEntries="5, 0, 0, 0, 50&#xD;&#xA;"
       PatternRows="1" PatternColumns="7"
       Pattern="Stable Floor,Empty,Empty,Empty,Empty,Floor,Floor" />
    <MovementAid Name="Jump 2 Below Left" GraphEntries="3,0,0,1,10"
       PatternRows="2" PatternColumns="4"
       Pattern="Empty,Empty,Empty,Floor,Floor,Empty,Empty," />
    <MovementAid Name="Jump 2 Below Right" GraphEntries="0, 0, 3, 1, 10&#xD;&#xA;"
       PatternRows="2" PatternColumns="4"
       Pattern="Floor,Empty,Empty,Empty,,Empty,Empty,Floor" />
    <MovementAid Name="Drop Down 2 Left" GraphEntries="0, 0, 1, 2, 5" PatternRows="3"
       PatternColumns="2" Pattern="Stable Floor,Empty,Wall,Empty,,Stable Floor" />
    <MovementAid Name="Drop Down 2 Right" GraphEntries="1, 0, 0, 2, 5"
       PatternRows="3" PatternColumns="2"
       Pattern="Empty,Stable Floor,Empty,Wall,Stable Floor," />
    <MovementAid Name="Drop Down 2 Left Alternative" GraphEntries="0, 0, 1, 2, 5"
       PatternRows="3" PatternColumns="2"
       Pattern="Stable Floor,Empty,Empty,Empty,,Stable Floor" />
    <MovementAid Name="Drop Down 2 Right Alternative" GraphEntries="1, 0, 0, 2, 5"
       PatternRows="3" PatternColumns="2"
       Pattern="Empty,Stable Floor,Empty,Empty,Stable Floor," />
  </MovementAids>

</CellSet>
```

A.5. Game Content XML – Example Level for *Prince of Persia*



```
<PlatformLevel Width="20" Height="6">
  <Cell X="0" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="0" Y="1" Name="Normal Cells\Wall" Params="" />
  <Cell X="0" Y="2" Name="Normal Cells\Wall" Params="" />
  <Cell X="0" Y="3" Name="Normal Cells\Wall" Params="" />
  <Cell X="0" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="0" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="1" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="1" Y="1" Name="Normal Cells\Empty" Params="" />
  <Cell X="1" Y="2" Name="Normal Cells\Floor\Landable\StartL" Params="" />
  <Cell X="1" Y="3" Name="Normal Cells\Wall" Params="" />
  <Cell X="1" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="1" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="2" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="2" Y="1" Name="Normal Cells\Empty" Params="" />
  <Cell X="2" Y="2" Name="Normal Cells\Floor\Landable\StartR" Params="" />
  <Cell X="2" Y="3" Name="Normal Cells\Wall" Params="" />
  <Cell X="2" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="2" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="3" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="3" Y="1" Name="Normal Cells\Empty" Params="" />
  <Cell X="3" Y="2" Name="Normal Cells\Floor\Landable\Floor" Params="" />
  <Cell X="3" Y="3" Name="Normal Cells\Wall" Params="" />
  <Cell X="3" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="3" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="4" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="4" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
  <Cell X="4" Y="2" Name="Normal Cells\Floor\Landable\Pillar" Params="" />
  <Cell X="4" Y="3" Name="Normal Cells\Wall" Params="" />
  <Cell X="4" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="4" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="5" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="5" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
  <Cell X="5" Y="2" Name="Normal Cells\Empty" Params="" />
  <Cell X="5" Y="3" Name="Normal Cells\Floor\Landable\Floor" Params="" />
  <Cell X="5" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="5" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="6" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="6" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
  <Cell X="6" Y="2" Name="Normal Cells\Wall" Params="" />
  <Cell X="6" Y="3" Name="Normal Cells\Floor\Landable\Pillar" Params="" />
  <Cell X="6" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="6" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="7" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="7" Y="1" Name="Normal Cells\Floor\Landable\Guard"
       Params="Guard Type: Normal;&#xD;&#xA;Guard Direction: Left" />
  <Cell X="7" Y="2" Name="Normal Cells\Wall" Params="" />
  <Cell X="7" Y="3" Name="Normal Cells\Floor\Landable\Floor" Params="" />
  <Cell X="7" Y="4" Name="Normal Cells\Wall" Params="" />
```

```
<Cell X="7" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="8" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="8" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="8" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="8" Y="3" Name="Normal Cells\Floor\Landable\Gate" Params="" />
<Cell X="8" Y="4" Name="Normal Cells\Wall" Params="" />
<Cell X="8" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="9" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="9" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="9" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="9" Y="3" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="9" Y="4" Name="Normal Cells\Wall" Params="" />
<Cell X="9" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="10" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="10" Y="1" Name="Normal Cells\Floor\Unlandable\Spikes" Params="" />
<Cell X="10" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="10" Y="3" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="10" Y="4" Name="Normal Cells\Wall" Params="" />
<Cell X="10" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="11" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="11" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="11" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="11" Y="3" Name="Normal Cells\Empty" Params="" />
<Cell X="11" Y="4" Name="Normal Cells\Empty" Params="" />
<Cell X="11" Y="5" Name="Normal Cells\Floor\Unlandable\Spikes" Params="" />
<Cell X="12" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="12" Y="1" Name="Normal Cells\Floor\Landable\Button\Raise"
     Params="Trigger Type: Open;&#xD;&#xA;Trigger Coordinate: (8, 3)" />
<Cell X="12" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="12" Y="3" Name="Normal Cells\Empty" Params="" />
<Cell X="12" Y="4" Name="Normal Cells\Empty" Params="" />
<Cell X="12" Y="5" Name="Normal Cells\Floor\Unlandable\Spikes" Params="" />
<Cell X="13" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="13" Y="1" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="13" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="13" Y="3" Name="Normal Cells\Floor\Landable\Breakable" Params="" />
<Cell X="13" Y="4" Name="Normal Cells\Empty" Params="" />
<Cell X="13" Y="5" Name="Normal Cells\Floor\Unlandable\Spikes" Params="" />
<Cell X="14" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="14" Y="1" Name="Normal Cells\Wall" Params="" />
<Cell X="14" Y="2" Name="Normal Cells\Wall" Params="" />
<Cell X="14" Y="3" Name="Normal Cells\Floor\Landable\Torch" Params="" />
<Cell X="14" Y="4" Name="Normal Cells\Wall" Params="" />
<Cell X="14" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="15" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="15" Y="1" Name="Normal Cells\Floor\Landable\Button\Raise"
     Params="Trigger Type: Open;&#xD;&#xA;Trigger Coordinate: (17, 4)" />
<Cell X="15" Y="2" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="15" Y="3" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="15" Y="4" Name="Normal Cells\Wall" Params="" />
<Cell X="15" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="16" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="16" Y="1" Name="Normal Cells\Empty" Params="" />
<Cell X="16" Y="2" Name="Normal Cells\Empty" Params="" />
<Cell X="16" Y="3" Name="Normal Cells\Empty" Params="" />
<Cell X="16" Y="4" Name="Normal Cells\Floor\Landable\Floor" Params="" />
<Cell X="16" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="17" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="17" Y="1" Name="Normal Cells\Empty" Params="" />
<Cell X="17" Y="2" Name="Normal Cells\Empty" Params="" />
<Cell X="17" Y="3" Name="Normal Cells\Empty" Params="" />
<Cell X="17" Y="4" Name="Normal Cells\Floor\Landable\ExitL" Params="" />
<Cell X="17" Y="5" Name="Normal Cells\Wall" Params="" />
<Cell X="18" Y="0" Name="Normal Cells\Wall" Params="" />
<Cell X="18" Y="1" Name="Normal Cells\Empty" Params="" />
<Cell X="18" Y="2" Name="Normal Cells\Empty" Params="" />
```

```xml
  <Cell X="18" Y="3" Name="Normal Cells\Empty" Params="" />
  <Cell X="18" Y="4" Name="Normal Cells\Floor\Landable\ExitR" Params="" />
  <Cell X="18" Y="5" Name="Normal Cells\Wall" Params="" />
  <Cell X="19" Y="0" Name="Normal Cells\Wall" Params="" />
  <Cell X="19" Y="1" Name="Normal Cells\Wall" Params="" />
  <Cell X="19" Y="2" Name="Normal Cells\Wall" Params="" />
  <Cell X="19" Y="3" Name="Normal Cells\Wall" Params="" />
  <Cell X="19" Y="4" Name="Normal Cells\Wall" Params="" />
  <Cell X="19" Y="5" Name="Normal Cells\Wall" Params="" />
</PlatformLevel>
```

References

Aarseth, E. (2005). From Hunt the Wumpus to Everquest: Introduction to Quest Theory. In *Proceedings of the 4th International Conference on Entertainment Computing* (pp. 496–506). Berlin, Heidelberg: Springer-Verlag. doi:10.1007/11558651_48

Aarseth, E. (2012). A Narrative Theory of Games. In *Proceedings of the International Conference on the Foundations of Digital Games* (pp. 129–133). New York, NY, USA: ACM. doi:10.1145/2282338.2282365

Andersen, E., Liu, Y.-E., Apter, E., Boucher-Genesse, F., & Popović, Z. (2010). Gameplay Analysis Through State Projection. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games* (pp. 1–8). New York, NY, USA: ACM. doi:10.1145/1822348.1822349

Andrew Doull. (2008). The Death Of The Level Designer - Procedural Content Generation Wiki. Retrieved from http://pcg.wikidot.com/the-death-of-the-level-designer

Aponte, M.-V., Levieux, G., & Natkin, S. (2011a). Difficulty in Videogames: An Experimental Validation of a Formal Definition. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology* (pp. 49:1–49:8). New York, NY, USA: ACM. doi:10.1145/2071423.2071484

Aponte, M.-V., Levieux, G., & Natkin, S. (2011b). Measuring the Level of Difficulty in Single Player Video Games. *Entertainment Computing*, *2*(4), 205–213. doi:10.1016/j.entcom.2011.04.001

Ashmore, C., & Nitsche, M. (2007). The Quest in a Generated World. In *Proceedings of the 2007 DiGRA Internation Conference: Situated Play* (pp. 503–509). The University of Tokyo.

Baeza-Yates, R., & Régnier, M. (1993). Fast Two Dimensional Pattern Matching. *Information Processing Letters*, *45*, 41–45.

Baker, T. P. (1978). A Technique for Extending Rapid Exact-Match String Matching to Arrays of More Than One Dimension. *SIAM Journal of Computing*, *7*(4), 533–541.

Barendregt, W. (2012). You Have to Die! Parents and Children Playing Cooperative Games. In *Proceedings of the 11th International Conference on Interaction Design and Children - IDC '12* (p. 288). New York, New York, USA: ACM Press. doi:10.1145/2307096.2307147

Barrenho, F., Romão, T., Martins, T., & Correia, N. (2006). InAuthoring Environment: Interfaces for Creating Spatial Stories and Gaming Activities. In *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. New York, NY, USA: ACM. doi:10.1145/1178823.1178835

Bauer, A., & Popović, Z. (2012). RRT-Based Game Level Analysis, Visualization, and Visual Refinement. In *Proceedings of the Eighth Conference on Artificial Inteligence and Interactive Digital Entertainment (AIIDE 2012)* (pp. 8–13).

Björk, S. (2008). Games, Gamers, and Gaming: Understanding Game Research. In *Proceedings of the 12th International Conference on Entertainment and Media in the Ubiquitous Era* (pp. 64–68). New York, NY, USA: ACM. doi:10.1145/1457199.1457213

Björk, S., Lundgren, S., & Holopainen, J. (2003). Game Design Patterns. In *Proceedings of Digital Games Research Conference*.

Bondy, J., & Murty, U. (1976). *Graph Theory with Applications*. Elsevier Science Ltd/North-Holland.

Booth, M. (2009). The AI Systems of Left 4 Dead. In *Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2009).*

Brathwaite, B., & Schreiber, I. (2008). *Challenges for Game Designers*. Cengage Learning.

Calot, E. (2008). *Prince of Persia Specifications of File Formats*.

Chen, J. (2007). Flow in Games (and Everything Else). *Communications of the ACM*, *50*(4), 31. doi:10.1145/1232743.1232769

Chiba, N., Muraoka, K., & Fujita, K. (1998). An Erosion Model Based on Velocity Fields for the Visual Simulation of Mountain Scenery. *The Journal of Visualization and Computer Animation*, *9*(4), 185–194. doi:10.1002/(SICI)1099-1778(1998100)9:4<185::AID-VIS178>3.0.CO;2-2

Chomsky, N. (1956). Three Models for the Description of Language. *IRE Transactions on Information Theory*.

Colton, S. (2002). Automated Puzzle Generation. In *Proceedings of the AISB'02 Symposium on AI and Creativity in the Arts and Sciences*.

Compton, K., & Mateas, M. (2006). Procedural Level Design for Platform Games. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2006)* (pp. 109–111).

Cook, M., & Colton, S. (2011). Multi-faceted Evolution of Simple Arcade Games. In *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games* (pp. 289–296). IEEE. doi:10.1109/CIG.2011.6032019

Cooper, S., El Rhalibi, A., Merabti, M., Wetherall, J., & Rhalibi, A. (2010). Procedural Content Generation and Level Design for Computer Games. In *Proceedings of the 3rd International Symposium on AI & Games*.

Correia, N., Alves, L., Correia, H., Romero, L., Morgado, C., Soares, L., … Jorge, J. (2005). InStory: a System for Mobile Information Access, Storytelling and Gaming Activities in Physical Spaces. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology* (pp. 102 – 109). ACM.

Csikszentmihalyi, M. (1991). *Flow: The Psychology of Optimal Experience.* (H. Collins, Ed.) (Vol. 54). Harper & Row. doi:10.1145/1077246.1077253

Dahlskog, S., & Togelius, J. (2012). Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1. In *Proceedings of the FDG Workshop on Design Patterns in Games (DPG).*

Dahlskog, S., & Togelius, J. (2013). Patterns as Objectives for Level Generation. In *Proceedings of the Second Workshop on Design Patterns in Games, DPG 2013.* ACM.

Dahlskog, S., & Togelius, J. (2014a). A Multi-level Level Generator. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.

Dahlskog, S., & Togelius, J. (2014b). Procedural Content Generation Using Patterns as Objectives. In *Applications of Evolutionary Computation* (pp. 325–336). Springer Berlin Heidelberg. doi:10.1007/978-3-662-45523-4_27

Dahlskog, S., Togelius, J., & Nelson, M. J. (2014). Linear Levels through N-grams. In *Proceedings of the 18th International Academic MindTrek Conference*.

Dalsgaard, T., Skov, M. B. M., Stougaard, M., & Thomassen, B. (2006). Mediated Intimacy in Families: Understanding the Relation Between Children and Parents. In *Proceedings of the 2006 conference on Interaction design and children* (pp. 145–152). New York, NY, USA: ACM. doi:10.1145/1139073.1139110

Darken, C. (2007). Level Annotation and Test by Autonomous Exploration: Abbreviated Version. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2007).*

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection, Or The Preservation of Favoured Races in the Struggle for Life.* J. Murray.

Desurvire, H., Caplan, M., & Toth, J. (2004). Using Heuristics to Evaluate the Playability of Games. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems* (pp. 1509–1512). New York, NY, USA: ACM. doi:10.1145/985921.986102

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, *1*(1), 269–271. doi:10.1007/BF01386390

Dormans, J. (2010). Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*.

doi:10.1145/1814256.1814257

Dormans, J., & Bakkes, S. (2011). Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 216–228.

Drachen, A., & Canossa, A. (2009). Analyzing User Behavior via Gameplay Metrics. In *Proceedings of the 2009 Conference on Future Play* (pp. 19–20). New York, NY, USA: ACM. doi:10.1145/1639601.1639613

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., & Worley, S. (2002). *Texturing and Modeling: A Procedural Approach* (3rd ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Federoff, M. A. (2002). *Heuristics and Usability Guidelines for the Creation and Evaluation of Fun in Video Games*. Indiana University, Bloomington.

Ferwerda, J. (2003). Three Varieties of Realism in Computer Graphics. In *Proceedings of the SPIE - Human Vision and Electronic Imaging 2003* (pp. 290–297). doi:10.1.1.58.6039

Forišek, M. (2010). Computational Complexity of Two-dimensional Platform Games. In *Fun with Algorithms* (pp. 214–227). Springer Berlin Heidelberg. doi:10.1007/978-3-642-13122-6_22

Fournier, A., Fussell, D., & Carpenter, L. (1982). Computer Rendering of Stochastic Models. *Communications of the ACM ACM*, *25*(6), 371–384. doi:10.1145/358523.358553

Frasca, G. (1999). Ludology Meets Narratology: Similitude and Differences between (Video)games and Narrative. *The Ludologist*. Retrieved from http://www.ludology.org/articles/ludology.htm

Fry, B. (2004). *Computational Information Design*. Massachusetts Institute of Technology.

Gardner, M. (1991). *The Unexpected Hanging and Other Mathematical Diversions*. University Of Chicago Press.

Gonzalez, R., & Woods, R. (2007). *Digital Image Processing* (3rd ed.). Prentice Hall.

Grammenos, D., Savidis, A., & Stephanidis, C. (2009). Designing Universally Accessible Games. *Computers in Entertainment*, *7*(1), 1. doi:10.1145/1486508.1486516

Granic, I., Lobel, A., & Engels, R. C. M. E. (2014). The Benefits of Playing Video Games. *The American Psychologist*, *69*(1), 66–78. doi:10.1037/a0034857

Habel, R., Kusternig, A., & Wimmer, M. (2009). Physically Guided Animation of Trees. *Computer Graphics Forum*, *28*(2), 523–532. doi:10.1111/j.1467-8659.2009.01391.x

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*. doi:10.1109/TSSC.1968.300136

Hartsook, K., Zook, A., Das, S., & Riedl, M. O. (2011). Toward Supporting Stories with Procedurally Generated Game Worlds. In *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games (CIG),* (pp. 297–304).

Hassoun, M. (2003). *Fundamentals of Artificial Neural Networks*. A Bradford Book.

Hastings, E. J., Guha, R. K., & Stanley, K. O. (2009). Evolving Content in the Galactic Arms Race Video Game. In *Proceedings of the 5th International Conference on Computational Intelligence and Games* (pp. 241–248). Piscataway, NJ, USA: IEEE Press.

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural Content Generation for Games. *ACM Transactions on Multimedia Computing, Communications, and Applications*, *9*(1), 1–22. doi:10.1145/2422956.2422957

Hilgard, E. R. (1980). The Trilogy of Mind: Cognition, Affection, and Conation. *Journal of the History of the Behavioral Sciences*, *16*(2), 107–117. doi:10.1002/1520-6696(198004)16:2<107::AID-JHBS2300160202>3.0.CO;2-Y

Hoobler, N., Humphreys, G., & Agrawala, M. (2004). Visualizing Competitive Behaviors in Multi-user Virtual Environments. *IEEE Visualization 2004*, 163–170. doi:10.1109/VISUAL.2004.120

Howard, J. (2008). *Quests: Design, Theory, and History in Games and Narratives*. A K Peters Ltd.

Hullett, K. (2012). The Science of Level Design: Design Patterns and Analysis of Player Behavior in First-Person Shooter Levels.

Hullett, K., & Whitehead, J. (2010). Design Patterns in FPS Levels. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games - FDG '10* (pp. 78–85). New York, New York, USA: ACM Press. doi:10.1145/1822348.1822359

Humphreys, P. (2008). Mathematical Modeling in the Social Sciences. In *The Blackwell Guide to the Philosophy of the Social Sciences* (pp. 166–184). Blackwell Publishing Ltd. doi:10.1002/9780470756485.ch7

Hunicke, R. (2005). The Case for Dynamic Difficulty Adjustment in Games. In *ACE '05 Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology* (pp. 429–433).

Hunicke, R., & Chapman, V. (2004). AI for Dynamic Difficulty Adjustment in Games. In *Proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence (AAAI '04).*

Ibáñez-Martínez, J., & Delgado-Mata, C. (2009). From Competitive to Social Two-Player Videogames Flow. In *Proceedings of the 2Nd Workshop on Child, Computer and Interaction* (pp. 18:1–18:5). New York, NY, USA: ACM. doi:10.1145/1640377.1640395

Jenkins, H. (2004). Game Design as Narrative Architecture. *Computer*, *44*(3), 118–130.

Jennings-Teats, M., Smith, G., & Wardrip-Fruin, N. (2010a). Polymorph: A Model for Dynamic Level Generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*.

Jennings-Teats, M., Smith, G., & Wardrip-Fruin, N. (2010b). Polymorph: Dynamic Difficulty Adjustment through Level Generation. In *PCGames '10 Proceedings of the 2010 Workshop on Procedural Content Generation in Games*.

Jungnickel, D. (2008). *Graphs, Networks and Algorithms*. *Vasa*. Springer.

Juul, J. (2009). Fear of Failing? The Many Meanings of Difficulty in Video Games. *The Video Game Theory Reader*, 1–13.

Juul, J., & Norton, M. (2009). Easy to Use and Incredibly Difficult: On the Mythical Border Between Interface and Gameplay. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (pp. 107–112). New York, NY, USA: ACM. doi:10.1145/1536513.1536539

Karakovskiy, S., & Togelius, J. (2012). The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 55–67. doi:10.1109/TCIAIG.2012.2188528

Karp, R., & Rabin, M. (1987). Efficient Randomized Pattern-matching Algorithms. *IBM Journal of Research and Development*, *31*(2).

Kelly, G., & McCabe, H. (2006). A Survey of Procedural Techniques for City Generation. *ITB Journal*, (14), 87–130.

Kimbrough, S. O., Lu, M., Wood, D. H., & Wu, D. J. (2003). Exploring a Two-population Genetic Algorithm. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartI* (pp. 1148–1159). Berlin, Heidelberg: Springer-Verlag.

Klimmt, C., Blake, C., Hefner, D., Vorderer, P., & Roth, C. (2009). Player Performance, Satisfaction, and Video Game Enjoyment. In *Proceedings of the 8th International Conference on Entertainment Computing* (pp. 1–12). Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-642-04052-8_1

Komulainen, J., & Takatalo, J. (2008). Psychologically Structured Approach to User Experience in Games. In *NordiCHI '08 Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges* (pp. 487–490).

Koster, R. (2004). *A Theory of Fun for Game Design*. Paraglyph Press.

Lagae, A., Kaplan, C. S., Fu, C.-W., Ostromoukhov, V., & Deussen, O. (2008). Tile-based Methods for Interactive Applications. *ACM SIGGRAPH 2008 Classes on - SIGGRAPH '08*, 1.

doi:10.1145/1401132.1401254

LaValle, S., & Kuffner Jr, J. (2000). Rapidly-exploring Random Trees: Progress and Prospects.

LaValle, S., & Kuffner Jr, J. (2001). Randomized Kinodynamic Planning. *The International Journal of Robotics Research*, *20*(5).

Lee, Y.-S., & Cho, S.-B. (2012). Dynamic Quest Plot Generation Using Petri Net Planning. In *Proceedings of the Workshop at SIGGRAPH Asia* (pp. 47–52). New York, NY, USA: ACM. doi:10.1145/2425296.2425304

Lewis, C., Wardrip-Fruin, N., & Whitehead, J. (2012). Motivational Game Design Patterns of 'Ville Games. In *Proceedings of the International Conference on the Foundations of Digital Games* (pp. 172–179). doi:10.1145/2282338.2282373

Liapis, A., Yannakakis, G. N., & Togelius, J. (2013). Towards a Generic Method of Evaluating Game Levels. In G. Sukthankar & I. Horswill (Eds.), *Proceedings of the Ninth Conference on Artificial Inteligence and Interactive Digital Entertainment (AIIDE 2013)*. AAAI.

Lima, E. S. de, Feijó, B., & Furtado, A. (2014). Hierarchical Generation of Dynamic and Nondeterministic Quests in Games. In *Proceedings of the 11th International Conference on Advances in Computer Entertainment Technology*.

Lindenmayer, A. (1968). Mathematical Models for Cellular Interactions in Development.

Lumley, T., Diehr, P., Emerson, S., & Chen, L. (2002). The Importance of the Normality Assumption in Large Public Health Data Sets. *Annual Review of Public Health*, *23*, 151–169. doi:10.1146/annurev.publhealth.23.100901.140546

Malone, T. (1980). What Makes Things Fun to Learn? Heuristics for Designing Instructional Computer Games. In *SIGSMALL '80 Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems* (pp. 162–169). doi:10.1145/800088.802839

Mandelbrot, B. B. (1977). *Fractals: Form, Chance and Dimension*. W.H.Freeman & Company.

Mandryk, R. L., & Atkins, M. S. (2007). A Fuzzy Physiological Approach for Continuously Modeling Emotion During Interaction with Play Technologies. *International Journal of Human-Computer Studies*, *65*(4), 329–347. doi:10.1016/j.ijhcs.2006.11.011

Markov, A. (1971). Extension of the Limit Theorems of Probability Theory to a Sum of Variables Connected in a Chain. In *Dynamic Probabilistic Systems, Volume I: Markov Models* (pp. 552 – 577).

Mateas, M., & Stern, A. (2003a). Façade: An experiment in building a fully-realized interactive drama. *Game Developers Conference, Game …*, *2*.

Mateas, M., & Stern, A. (2003b). Integrating Plot, Character and Natural Language Processing in the Interactive Drama Façade. In *Proceedings of the Technologies for Interactive Digital Storytelling and Entertainment Conference*.

Mateas, M., & Stern, A. (2004). Natural Language Understanding in Façade: Surface-text Processing. *Technologies for Interactive Digital Storytelling and …*, 1–12. doi:10.1007/978-3-540-27797-2_2

Mateas, M., & Stern, A. (2005). Procedural Authorship: A Case-study of the Interactive Drama Façade. *Digital Arts and Culture (DAC)*.

Mawhorter, P., & Mateas, M. (2010). Procedural Level Generation Using Occupancy-regulated Extension. *2010 IEEE Conference on Computational Intelligence and Games*, 351–358. doi:10.1109/ITW.2010.5593333

McConnell, S. (2001). Who Needs Software Engineering? *Software, IEEE*, *18*(1), 5–8. doi:10.1109/MS.2001.903148

Mechner, J. (1989). Prince of Persia - Technical Information.

Miller, G. S. P. (1986). The Definition and Rendering of Terrain Maps. *SIGGRAPH Comput. Graph.*, *20*(4), 39–48. doi:10.1145/15886.15890

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT press.

Mourato, F., Birra, F., & Próspero dos Santos, M. (2012a). Enhancing Level Difficulty and Additional Content in Platform Videogames through Graph Analysis. In *Proceedings of the 9th International Conference on Advances in Computer Entertainment Technology*. doi:10.1007/978-3-642-34292-9_6

Mourato, F., Birra, F., & Próspero dos Santos, M. (2012b). Integrated System for Automatic Platform Game Level Creation with Difficulty and Content Adaptation. In *Entertainment Computing-ICEC 2012* (pp. 409–412). Springer Berlin Heidelberg. doi:10.1007/978-3-642-33542-6_40

Mourato, F., Birra, F., & Próspero dos Santos, M. (2012c). Sistema Integrado de Geração Automática de Conteúdo para Videojogos de Plataformas. In *20º Encontro Português de Computação Grafica*.

Mourato, F., Birra, F., & Próspero dos Santos, M. (2013a). The Challenge of Automatic Level Generation for Platform Videogames Based on Stories and Quests. *Advances in Computer Entertainment*. doi:10.1007/978-3-319-03161-3_24

Mourato, F., Birra, F., & Próspero dos Santos, M. (2013b). Using Graph-Based Analysis to Enhance Automatic Level Generation for Platform Videogames. *International Journal of Creative Interfaces and Computer Graphics*, *4*(1), 49–70. doi:10.4018/ijcicg.2013010104

Mourato, F., Birra, F., & Próspero dos Santos, M. (2014). Difficulty in Action Based Challenges: Success Prediction, Players' Strategies and Profiling. In *Proceedings of the 11th International Conference on Advances in Computer Entertainment Technology*.

Mourato, F., & Próspero dos Santos, M. (2010). Measuring Difficulty in Platform Videogames. In *4ª Conferência Nacional Interacção humano-computador* (pp. 173–180). Aveiro, Portugal.

Mourato, F., Próspero dos Santos, M., & Birra, F. (2011). Automatic Level Generation for Platform Videogames using Genetic Algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology (ACE 2011)* (pp. 8:1–8:8). New York, NY, USA: ACM. doi:10.1145/2071423.2071433

Myerson, R. B. (1997). *Game Theory: Analysis of Conflict*. Harvard University Press.

Nelson, M., & Mateas, M. (2007). Towards Automated Game Design. *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*. doi:10.1007/978-3-540-74782-6_54

Nicollet, V. (2004). Difficulty in Dexterity-based Platform Games. *GameDev. Net*, 1–7.

Nielsen, J. (1993). *Usability Engineering*. Academic Press, Oxford, UK. 1.

Nierhaus, G. (2009). *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer.

Niesz, A., & Holland, N. (1984). Interactive Fiction. *Critical Inquiry*, *11*(1), 110–129.

Norman, D. A. (2002). *The Design of Everyday Things*. Basic Books.

Norris, J. R. (1998). *Markov Chains*. Cambridge University Press.

Nygren, N., Denzinger, J., Stephenson, B., & Aycock, J. (2011). User-preference-based Automated Level Generation for Platform Games. *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, 55–62. doi:10.1109/CIG.2011.6031989

Olesen, J. K., Yannakakis, G. N., & Hallam, J. (2008). Real-time Challenge Balance in an RTS Game Using rtNEAT. In *IEEE Symposium On Computational Intelligence and Games* (pp. 87–94).

Oren, M. (2007). Speed Sonic across the Span: Building a Platform Audio Game. *CHI EA'07 CHI'07 Extended Abstracts on Human Factors in Computing Systems*, 2231–2236. doi:10.1145/1240866.1240985

Özkar, M., & Stiny, G. (2009). Shape Grammars. *ACM SIGGRAPH 2009 Courses on - SIGGRAPH '09*, 1–176. doi:10.1145/1667239.1667261

Pedersen, C., Togelius, J., & Yannakakis, G. (2009a). Modeling Player Experience in Super Mario Bros. *5th International Conference on Computational Intelligence and Games*, 132–139. doi:10.1109/CIG.2009.5286482

Pedersen, C., Togelius, J., & Yannakakis, G. (2009b). Optimization of Platform Game Levels for Player Experience.

Pedersen, C., Togelius, J., & Yannakakis, G. (2010). Modeling Player Experience For Content Creation. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(1), 1–14. doi:10.1109/TCIAIG.2010.2043950

Pereira, G., Santos, P., & Prada, R. (2009). Self-adapting Dynamically Denerated Maps for Turn-based Strategic Multiplayer Browser Games. In *ACE '09 Proceedings of the International Conference on Advances in Computer Enterntainment Technology* (pp. 353–356). doi:10.1145/1690388.1690457

Perlin, K. (1985). An Image Synthesizer. *ACM SIGGRAPH Computer Graphics*, *19*(3), 287–296. doi:10.1145/325165.325247

Piaget, J. (1962). *Play Dreams & Imitation in Childhood*. W. W. Norton & Company.

Re, A., Abad, F., Camahort, E., & Juan, M. C. (2009). Tools for Procedural Generation of Plants in Virtual Scenes. In G. Allen, J. Nabrzyski, E. Seidel, G. Albada, J. Dongarra, & P. A. Sloot (Eds.), *Computational Science – ICCS 2009 SE - 89* (Vol. 5545, pp. 801–810). Springer Berlin Heidelberg. doi:10.1007/978-3-642-01973-9_89

Risi, S., Lehman, J., D'Ambrosio, D., Hall, R., & Stanley, K. (2012). Combining Search-Based Procedural Content Generation and Social Gaming in the Petalz Video Game. In *AIIDE*.

Rocha, J., Mascarenhas, S., & Prada, R. (2008). Game Mechanics for Cooperative Games. *ZON Digital Games 2008*, 72–80.

Ryan, M. (2006). *Avatars of Story*. University of Minnesota Press.

Scales, D., & Thompson, T. (2014). SpelunkBots API - An AI Toolset for Spelunky. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on* (pp. 1–8). doi:10.1109/CIG.2014.6932872

Seif El-Nasr, M., Aghabeigi, B., Milam, D., Erfani, M., Lameman, B., Maygoli, H., & Mah, S. (2010). Understanding and Evaluating Cooperative Games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 253–262). New York, New York, USA: ACM. doi:10.1145/1753326.1753363

Shaker, N., Togelius, J., & Nelson, M. J. (2014). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Shaker, N., Togelius, J., Yannakakis, G., Weber, B., Shimizu, T., Hashiyama, T., … Baumgarten, R. (2011). The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(4), 332–347. doi:10.1109/TCIAIG.2011.2166267

Shaker, N., Yannakakis, G., & Togelius, J. (2010). Towards Automatic Personalized Content Generation for Platform Games. *AIIDE*.

Shaker, N., Yannakakis, G., & Togelius, J. (2011). Feature Analysis for Modeling Game Content Quality. *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, (4), 126–133. doi:10.1109/CIG.2011.6031998

Shaker, N., Yannakakis, G., & Togelius, J. (2012). Digging Deeper into Platform Game Level Design: Session Size and Sequential Features. *Applications of Evolutionary Computation*, *7248*, 275–284. doi:10.1007/978-3-642-29178-4_28

Shneiderman, B. (1987). Human-Computer Interaction. In R. M. Baecker & W. A. S. Buxton (Eds.), (pp. 461–467). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Siyahhan, S., Barab, S. A., & Downton, M. P. (2010). Using Activity Theory to Understand Intergenerational Play: The Case of Family Quest. *International Journal of Computer-Supported Collaborative Learning*, *5*(4), 415–432. doi:10.1007/s11412-010-9097-1

Smelik, R., Kraker, K. J. De, Groenewegen, S., Tutenel, T., & Bidarra, R. (2009). A Survey of Procedural Methods for Terrain Modelling. In *Proc. of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*.

Smelik, R. M., Tutenel, T., Bidarra, R., & Benes, B. (2014). A Survey on Procedural Modelling for Virtual Worlds. *Computer Graphics Forum*, *33*(6), 31–50. doi:10.1111/cgf.12276

Smelik, R., Tutenel, T., Kraker, K. J. De, & Bidarra, R. (2011). A Declarative Approach to Procedural

Modeling of Virtual Worlds. *Computers & Hraphics*, *35*(2), 352–363. doi:10.1016/j.cag.2010.11.011

Smith, G. (2014a). The Future of Procedural Content Generation in Games. In *Proceedings of the AIIDE Workshop on Experimental AI in Games*.

Smith, G. (2014b). 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* (pp. 917–926). New York, NY, USA: ACM. doi:10.1145/2556288.2557341

Smith, G., Anderson, R., Kopleck, B., Lindblad, Z., Scott, L., Wardell, A., … Mateas, M. (2011). Situating Quests: Design Patterns for Quest and Level Design in Role-Playing Games. In M. Si, D. Thue, E. André, J. Lester, J. Tanenbaum, & V. Zammitto (Eds.), *Interactive Storytelling* (Vol. 7069, pp. 326–329). Springer Berlin / Heidelberg. doi:10.1007/978-3-642-25289-1_40

Smith, G., Cha, M., & Whitehead, J. (2008). A Framework for Analysis of 2D Platformer Levels. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games - Sandbox '08* (p. 75). New York, New York, USA: ACM Press. doi:10.1145/1401843.1401858

Smith, G., Gan, E., Othenin-Girard, A., & Whitehead, J. (2011). PCG-based Game Design: Enabling New Play Experiences through Procedural Content Generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* (pp. 5–8). doi:10.1145/2000919.2000926

Smith, G., & Othenin-Girard, A. (2012). *PCG-based Game Design: Creating Endless Web. Foundations of Digital Games 2012 (FDG '12).* doi:10.1145/2282338.2282375

Smith, G., Treanor, M., Whitehead, J., & Mateas, M. (2009). Rhythm-based Level Generation for 2D Platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games - FDG '09* (p. 175). New York, New York, USA: ACM Press. doi:10.1145/1536513.1536548

Smith, G., & Whitehead, J. (2010). Analyzing the Expressive Range of a Level Generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames '10* (pp. 1–7). New York, New York, USA: ACM Press. doi:10.1145/1814256.1814260

Smith, G., & Whitehead, J. (2011). Launchpad: A Rhythm-Based Level Generator for 2-D Platformers. *Int'l Conference on the Foundations of Digital Games*, *3*(1), 1–16. doi:10.1109/TCIAIG.2010.2095855

Smith, G., Whitehead, J., & Mateas, M. (2010a). Tanagra: A Mixed-Initiative Level Design Tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games - FDG '10* (pp. 209–216). New York, New York, USA: ACM Press. doi:10.1145/1822348.1822376

Smith, G., Whitehead, J., & Mateas, M. (2010b). Tanagra: An Intelligent Level Design Assistant for 2D Platformers. *Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Smith, G., Whitehead, J., & Mateas, M. (2011). Tanagra: Reactive Planning and Constraint Solving for Mixed-initiative Level Design. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 201–215. doi:10.1109/TCIAIG.2011.2159716

Snodgrass, S., & Ontañón, S. (2014a). A Hierarchical Approach to Generating Maps Using Markov Chains. In *Proceedings of the Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014)* (pp. 59–65).

Snodgrass, S., & Ontañón, S. (2014b). Experiments in Map Generation using Markov Chains. In *Foundations of Digital Games 2014 (FDG '14)*.

Snook, G. (2000). Simplified 3D Movement and Pathfinding Using Navigation Meshes. In *Game Programming Gems*. Charles River Media.

Sorenson, N., & Pasquier, P. (2010a). The Evolution of Fun: Automatic Level Design through Challenge Modeling. In *Proceedings of the First International Conference on Computational Creativity (ICCCX)* (pp. 258–267). Lisbon.

Sorenson, N., & Pasquier, P. (2010b). Towards a Generic Framework for Automated Video Game Level Creation. *Applications of Evolutionary Computation*. doi:10.1007/978-3-642-12239-2_14

Sorenson, N., Pasquier, P., & DiPaola, S. (2011). A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 229–244. doi:10.1109/TCIAIG.2011.2161310

Spuy, R. (2012). *Foundation Game Design with HTML5 and JavaScript*. Berkeley, CA: Apress. doi:10.1007/978-1-4302-4717-3

Stiny, G., & Gips, J. (1971). Shape Grammars and the Generative Specification of Painting and Sculpture. *IFIP Congress (2)*, *71*, 125–135.

Suits, B. (2005). *The Grasshopper: Games, Life and Utopia*. Broadview Press.

Sullivan, A., Grow, A., Mateas, M., & Wardrip-fruin, N. (2012). The Design of Mismanor: Creating a Playable Quest-based Story Game. In *Proceedings of the International Conference on the Foundations of Digital Games* (pp. 180–187). doi:10.1145/2282338.2282374

Sullivan, A., Mateas, M., & Wardrip-Fruin, N. (2009). QuestBrowser : Making Quests Playable with Computer- Assisted Design.

Sullivan, A., Mateas, M., & Wardrip-Fruin, N. (2010). Rules of Engagement: Moving beyond Combat-based Quests. In *Proceedings of the Intelligent Narrative Technologies III Workshop*. doi:10.1145/1822309.1822320

Sweetser, P., & Wyeth, P. (2005). GameFlow: A Model for Evaluating Player Enjoyment in Games. *Computers in Entertainment (CIE)*, *3*(3), 1–24. doi:10.1145/1077246.1077253

Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., & Stanley, K. O. (2013). Procedural Content Generation: Goals, Challenges and Actionable Steps. In S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, & J. Togelius (Eds.), *Artificial and Computational Intelligence in Games* (Vol. 6, pp. 61–75). Dagstuhl, Germany: Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik. doi:http://dx.doi.org/10.4230/DFU.Vol6.12191.61

Togelius, J., De Nardi, R., & Lucas, S. (2006). Making Racing Fun through Player Modeling and Track Evolution. In *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*.

Togelius, J., De Nardi, R., & Lucas, S. M. (2007). Towards Automatic Personalised Content Creation for Racing Games. *2007 IEEE Symposium on Computational Intelligence and Games*, 252–259. doi:10.1109/CIG.2007.368106

Togelius, J., Justinussen, T., & Hartzen, A. (2012). Compositional Procedural Content Generation. *FDG Workshop on Procedural Content Generation (PCG)*. doi:10.1145/2538528.2538541

Togelius, J., Karakovskiy, S., & Baumgarten, R. (2010). The 2009 Mario AI Competition. *IEEE Congress on Evolutionary Computation*, 1–8. doi:10.1109/CEC.2010.5586133

Togelius, J., Karakovskiy, S., Koutník, J., & Schmidhuber, J. (2009). Super Mario Evolution. In *Proceedings of the 5th international conference on Computational Intelligence and Games* (pp. 156–161). Piscataway, NJ, USA: IEEE Press.

Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. N. (2011). What is Procedural Content Generation?: Mario on the Borderline. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games* (pp. 3:1–3:6). New York, NY, USA: ACM. doi:10.1145/2000919.2000922

Togelius, J., & Schmidhuber, J. (2008). An Experiment in Automatic Game Design. *2008 IEEE Symposium On Computational Intelligence and Games*, 111–118. doi:10.1109/CIG.2008.5035629

Togelius, J., Yannakakis, G., Stanley, K. O., & Browne, C. (2010). Search-based Procedural Content Generation. In *Proceedings of the European Conference on Applications of Evolutionary Computation (EvoApplications)*. doi:10.1007/978-3-642-12239-2_15

Tosca, S. (2003). The Quest Problem in Computer Games. *Technologies for Interactive Digital Storytelling and Entertainment (TIDSE)*.

Tozour, P. (2002). Building a Near-Optimal Navigation Mesh. In *AI Game Programming Wisdom*. Charles River Media.

Treanor, M., Blackford, B., Mateas, M., & Bogost, I. (2012). Game-o-matic: Generating Videogames that Represent Ideas. *Procedural Content Generation Workshop at the Foundations of Digital Games Conference*.

Tsang, E. (1993). *Foundations of Constraint Satisfaction (Computation in Cognitive Science)*. Academic Press, London and San Diego.

Tyson, P. (2012). *Getting Started with Dwarf Fortress: Learn to Play the Most Complex Video Game Ever Made*. O'Reilly Media.

van Dongen, S. (2000). *Graph Clustering by Flow Simulation*. University of Utrecht.

Viglietta, G. (2012). Gaming is a Hard Job, but Someone has to do it! *Fun with Algorithms*, 1–32. doi:10.1007/978-3-642-30347-0_35

Wallner, G. (2013). Play-Graph: A Methodology and Visualization Approach for the Analysis of Gameplay Data. *Foundations of Digital Games 2013 (FDG '13)*.

Wallner, G., & Kriglstein, S. (2012). A Spatiotemporal Visualization Approach for the Analysis of Gameplay Data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1115–1124). New York, NY, USA: ACM. doi:10.1145/2207676.2208558

Weber, J., & Penn, J. (1995). Creation and Rendering of Realistic Trees. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques* (pp. 119–128). New York, NY, USA: ACM. doi:10.1145/218380.218427

West, D. (2000). *Introduction to Graph Theory*. Prentice Hall.

Williams-King, D., Denzinger, J., Aycock, J., & Stephenson, B. (2012). The Gold Standard: Automatically Generating Puzzle Game Levels. *AIIDE*, 191–196.

Yannakakis, G. N., & Togelius, J. (2011). Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing*, *2*(3), 147–161. doi:10.1109/T-AFFC.2011.6

Zagal, J., & Bruckman, A. (2008). The Game Ontology Project: Supporting Learning While Contributing Authentically to Game Studies. In *Proceedings of the 8th International Conference on International Conference for the Learning Sciences - Volume 2* (pp. 499–506). International Society of the Learning Sciences.

Zagal, J. P., Mateas, M., Fernández-Vara, C., Hochhalter, B., & Lichti, N. (2005). Towards an Ontological Language for Game Analysis.

Zagalo, N., Prada, R., Alexandre, I. M., & Torres, A. (2008). Authoring Emotion. In *Affective Computing*. I-Tech Education and Publishing.

Zhu, R., & Takaoka, T. (1989). A Technique for Two-dimensional Pattern Matching. *Communications of the ACM*, *32*(9), 1110–1120. doi:10.1145/66451.66459

***Note**: All provided links were checked for availability by the 1st of December 2015.*

Cited Videogames

.kkrieger. Farbrausch. 2004.

And Yet it Moves. Broken Rules. 2009.

Angry Birds. Rovio Entertainment. 2009.

Another World. Delphine Software; Interplay Entertainment. 1991.

Bionic Commando. Capcom. 1987.

Borderlands. Gearbox Software. 2009.

Braid. Number None Inc. 2009.

Bubble Bobble. Taito. 1986.

Canabalt. Semi-secret Software. 2009.

Civilization. Microprose. 1991.

Cloudberry Kingdom. Ubisoft. 2013.

Contra. Konami. 1987.

Doodle Jump. Lima Sky. 2009.

Dwarf Fortress. Bay 12 Games. 2006.

Elite. Braben D., Bell, I. 1984.

Façade. Mateas, M. Stern, A. 2005.

Farmville. Zynga. 2009.

Golden Axe. Sega. 1989.

I Must Run. Gamelion Studios. 2012.

Infinite Mario Bros.

Infinite Tux.

KGoldrunner. https://games.kde.org/game.php?game=kgoldrunner.

Left 4 Dead. Turtle Rock Studios. 2008.

Lode Runner. Smith, D. 1983.

Little Big Planet. Media Molecule. 2008.

Metal Slug. Nazca Corporation. 1996.

New Super Mario Bros. U. Nintendo. 2012.

New Super Mario Bros. Wii. Nintendo. 2009

Pong. Atari. 1972.

Prince of Persia. Brøderbund. 1989.

Rick Dangerous. Core Design. 1989.

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

Run Like Hell. Mass Creation. 2013.

Sim City. Maxis. 1989.

Sonic the Hedgehog. Sega. 1991.

Sonic 4. Sega. 2010.

Spelunky. Microsoft Studios. 2009.

Spore. Maxis. 2008.

Spore: Creature Creator. Maxis. 2008.

Streets of Rage. Sega. 1991.

Super Mario Bros. Miyamoto, S., & Tezuka, T., Nintendo. 1985.

Super Mario 64. Nintendo. 1996.

Temple Run. Imangi Studios. 2011.

Tetris. Pajitnov A. 1984.

The Adventures of Tintin. Ubisoft. 2011.

The Cave. Double Fine Productions. 2013.

The Lost Vikings. Silicon & Synapse (Blizzard entertainment). 1992.

Thomas Was Alone. Bithell, M. 2012.

Trine. Frozenbyte. 2011.

World of Warcraft. Blizzard Entertainment. 2004.

XRick. http://www.bigorno.net/xrick/. 2002.