UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Procedural generation of dungeon maps, missions and rooms

Leonardo Tórtoro Pereira

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Leonardo Tórtoro Pereira

Procedural generation of dungeon maps, missions and rooms

Master dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Claudio Fabiano Motta Toledo

USP – São Carlos
January 2019

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

T436p   Tórtoro Pereira, Leonardo
             Procedural generation of dungeon maps,
missions and rooms / Leonardo Tórtoro Pereira;
orientador Claudio Fabiano Motta Toledo. -- São
Carlos, 2019.
             124 p.

             Dissertação (Mestrado - Programa de Pós-Graduação
em Ciências de Computação e Matemática
Computacional) -- Instituto de Ciências Matemáticas
e de Computação, Universidade de São Paulo, 2019.

             1. Evolutionary algorithm. 2. Level generation.
3. Procedural content generation. 4. Mission
generation. 5. Adaptive game content. I. Fabiano
Motta Toledo, Claudio, orient. II. Título.

Leonardo Tórtoro Pereira

Geração procedural de mapas de calabouço, missões e salas

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Claudio Fabiano Motta Toledo

USP – São Carlos
Janeiro de 2019

This work is dedicated to my parents and friends for always helping me to grow as a human being.

I also dedicate it to all the teachers who I had the pleasure to learn something from.

ACKNOWLEDGEMENTS

I would like to thank my parents and friends for all the support given through all these years. I would also like to thank my advisor, professor Claudio Fabiano Motta Toledo, for guiding me and encouraging me to enter the world of science and research since the beginning of my undergraduate courses and now in my graduate program.

I would like to thank my fellow graduate student Paulo Victor de Souza Prado for helping me to develop this project, specially the evolutionary algorithm which creates the dungeon maps. His help was imperative for the conclusion of this research. I also would like to thank my friend Mateus Malvessi Pereira for the dungeon visualization algorithm made using GameMaker Studio 2, which allowed us to visualize the generated dungeons in a simple and pleasing way, as in Figure 14, instead of our original visualization in the console. I would also like to thank Giovanna Oliveira Guimarães, the main developer of the room generating algorithm, presented in section 4.2, as part of her scientific initiation. At last, but not least, I would like to thank my fellow graduate student Rafael Miranda Lopes for aiding in the development of the game prototype using the Unity game engine, presented in section 4.3.

I would also like to thank all the researchers, scientists and philosophers, from all times and places, for each contribution, as small as it may have been, that each gave to science and the human knowledge.

I would like to thank every artist I had the pleasure to enjoy and learn from their work. May it be from their books, movies, cartoons, games, musics or comics. Because, as the great philosopher Friedrich Nietzsche said: “we have art in order not to die of the truth”.

Special thanks go to the game journalist and youtuber Mark Brown¹ for his excellent work in analyzing the design of games in his channel and his series of videos *Boss Keys*² being one of the main inspirations for this research. I would also like to thank Shigeru Miyamoto, Takashi Tezuka, Eiji Aonuma, and all the ones who supported *The Legend of Zelda*³ game franchise and kept it alive for all this years with great games, that, aside from being the source from the analysis in the *Boss Keys* series, were a valuable source of entertainment and inspiration for many game developers and game students, including myself.

At last, I would like to thank the great legend from the video game industry who always inspired me, the late Satoru Iwata, a honorable man and incredible programmer.

---

¹ <<https://www.youtube.com/user/McBacon1337/>>

² <https://www.youtube.com/playlist?list=PLc38fcMFcV_ul4D6OChdWhsNsYY3NA5B2>

³ <<http://www.zelda.com/>>

“The only necessity for success is the willingness to suffer 1,000 failures.”
(Kotaro Uchikoshi)

RESUMO

PEREIRA, L. T. **Geração procedural de mapas de calabouço, missões e salas.** 2019. 124 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP – Brazil, 2019.

O projeto propõe duas abordagens de geração de conteúdo para missões e mapas em um jogo com calabouços, e um protótipo de jogo desenvolvido como prova de conceito da factibilidade do algoritmo e também como plataforma de testes para experimentos com usuários. A primeira abordagem define missões ao gerar mapas de calabouço em conjunto com chaves e portas trancadas através do uso de um algoritmo evolutivo. Já a segunda abordagem irá popular o espaço do calabouço criado ao preencher o conteúdo de suas salas usando algoritmos de curvas de preenchimento de espaço e autômatos celulares. O algoritmo evolutivo evolui uma estrutura em árvore que codifica informações sobre o calabouço. O objetivo é convergir os calabouços gerados para se aproximar ao máximo da configuração de entrada fornecida por um *designer* de jogos. O calabouço possui informação sobre as salas, como a quantidade das mesmas, as conexões entre elas e suas posições em um mapa 2D. Também contam com informações semânticas relevantes para a geração de narrativa no calabouço, que são o posicionamento de pares únicos de chaves e fechaduras através do calabouço. Portanto, uma maneira factível para o término do mesmo pode ser gerada proceduralmente. O conteúdo das salas é gerado usando curvas de preenchimento de espaço como as de Hilbert e Moore, além do autômato celular do Jogo da Vida de Conway. Resultados computacionais demonstram que o algoritmo evolutivo é capaz de criar calabouços com propriedades muito próximas às desejadas para uma grande variedade de entradas para calabouços com até 100 salas. Resultados preliminares de validação com humanos mostraram não haver diferença estatística entre os níveis gerados proceduralmente daqueles feitos por humanos. Testes posteriores de validação centrados em humanos com o protótipo de jogo mostram que os níveis gerados pelo algoritmo são percebidos como feitos por humanos de maneira semelhante ou até em maior grau do que suas contrapartidas geradas de fato por humanos, assim como são percebidos como mais divertidos e difíceis do que estas. Portanto, esta pesquisa objetiva gerar elementos de jogabilidade combinando diferentes algoritmos em uma única solução, que pode ser facilmente adaptada para uma variedade de jogos diferentes.

Palavras-chave: Algoritmo Evolutivo, Geração Procedural de Conteúdo, Geração de Nível, Geração de Missão, Conteúdo Adaptativo de Jogo.

ABSTRACT

PEREIRA, L. T. **Procedural generation of dungeon maps, missions and rooms**. 2019. 124 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP – Brazil, 2019.

The present research proposes two procedural content generation approaches for missions and play space in a game with dungeons, and a game prototype developed as a proof of concept for the feasibility of the algorithm and as a test framework for user-based experiments. The first approach will define missions by generating dungeon maps together with locked doors and keys through the use of an evolutionary algorithm. The second approach will populate the generated dungeon space by filling the content of dungeon rooms using space-filling curves and cellular automata algorithms. The evolutionary algorithm evolves tree structures encoding information about the dungeon. The goal is to converge the generated dungeons as close as possible to satisfy the set of parameter inputs provided by a game designer. The dungeon holds information about the amount of rooms, the connections between them and their position in a 2D map. There are also relevant semantic information in the content for the generation of narrative, which presents itself by the placement of unique pairs of keys and locks throughout it. Thus, a feasible way to finish the dungeon can be procedurally generated. The content of rooms are generated using space-filling curves algorithms such as Hilbert and Moore curves as well as Conway's Game of Life Cellular Automata. Computational results report that the evolutionary algorithm provides dungeons with up to a 100 rooms very close to the desired ones for a range of different parameter inputs. The early validation tests with humans show no statistical difference between levels procedurally generated and those made by humans. Further user-centred validation tests with the game prototype show the algorithm-generated levels are perceived as equally or even more human-made than their human-authored counterparts, as well as funnier and more difficult. Thus, the research aims to generate gameplay elements combining different algorithms for a single solution, which could be easily adapted to a range of different games.

Keywords: Evolutionary Algorithm, Procedural Content Generation, Level Generation, Mission Generation, Adaptive Game Content.

LIST OF FIGURES

Figure 1	– An example of a dungeon map generated by our solution. Keys and locked doors are distributed in a way that generates interesting puzzles for the player. The starting room is where the avatar is placed, and the goal is the room with the green triangle.	30
Figure 2	– Example of a dungeon with both its map layout and rooms generated by our algorithms.	31
Figure 3	– <i>The Legend of Zelda: Breath of the Wild</i> (top) and <i>Horizon Zero Dawn</i> (bottom), two award-winning action-adventure games released in 2017 . . .	35
Figure 4	– Graph abstraction of 2 levels from <i>The Legend of Zelda: The Minish Cap</i> game. Only rooms containing items which are essential for the player’s progression are shown. Each color identifies a different type of key-lock pair. Keys are represented as diamonds and locks as squares. The vertical lines show a sequence of rooms in the same path, while the horizontal ones identify a path’s branching. Many rooms in the same horizontal line may be visited by players, if they wish so. Extracted from: < https://www.youtube.com/watch?v=KEVJXqV7XMd >	37
Figure 5	– A possible solution for the puzzle in the upper dungeon from Figure 4. In the upper left image the first key is found and used in the first lock. In the upper right one, two levers are used to open the green door. In the lower left image, the player finds the bow and opens the first door requiring its use. At last, in the lower right image, the player gets the boss key and opens the last lock. Extracted from: < https://www.youtube.com/watch?v=KEVJXqV7XMd > . .	38
Figure 6	– General representation of an Evolutionary Algorithm. Extracted from: (EIBEN; SMITH, 2003) (p. 17).	39
Figure 7	– A* execution example. Extracted from: (LESTER, 2005).	41
Figure 8	– Recursive creation process of an order 3 Hilbert curve starting from an order 2 curve. Adapted from (NØRMAK, 2013).	43
Figure 9	– The 6 first orders from Moore curve.	45
Figure 10	– Creation process of an order 3 Moore curve starting from an order 2 Hilbert curve. Adapted from (NØRMAK, 2013).	45

Figure 11 – The Tree Structure (Genotype) of a generated dungeon. Parameters: 20 Rooms, 4 Keys, 4 Locks and 1.5 Linearity Coefficient. Each node is a room, each edge a door connecting rooms. A dashed edge represents a locked door and a thick line surrounding a node indicates there is a key in said room. Both keys and locks ids are shown beside the corresponding edge or node. The root node, with the thin dashed line, is the starting room of the dungeon, while the node with the thick dashed line is the final one.	60
Figure 12 – Example of the directions used when creating the children rooms: the room orientation is represented by the compass rose. The first room is always created aligned with the room, and every children’s parent is always considered to be north of it.	60
Figure 13 – The resulting dungeon (Phenotype) from translating the genotype in Figure 11. Keys are represented by yellow squares, locks by red rectangles, each with their respective ids. Rooms are presented in white squares, and doors as green rectangles. The starting room is marked with a blue ‘S’ and the final room a black ‘B’.	61
Figure 14 – Left: The Grid Representation (Phenotype) of the dungeon in Figure 11. Right: The same dungeon, but with each key opening only its corresponding lock (color-matching)	63
Figure 15 – First step of genotype-phenotype translation.	64
Figure 16 – Second step of genotype-phenotype translation	64
Figure 17 – Third step of genotype-phenotype translation	64
Figure 18 – Fourth step of genotype-phenotype translation	65
Figure 19 – Fifth and last step of genotype-phenotype translation	65
Figure 20 – Visual representation of the discretization process of two Hilbert curves of different orders (2, on the left, and 3, on the right). The green cells are walkable tiles, and black ones are blocks.	73
Figure 21 – Creation of two 7x7 rooms using order 2 (left) and 3 (right) Hilbert curves. The section is extracted starting at point (4,2) in both corresponding Hilbert matrices. In the second to last row, the doors are placed in previously defined exits. Then, in the last row, paths to connect the doors in the right room are created.	75
Figure 22 – Screenshot from the prototype game in the Unity engine. The screen shows a dungeon generated using the outputs of both the dungeon and rooms PCG algorithms proposed in this research	76

Figure 23 – Left, from top to bottom: <i>The Legend of Zelda’s</i> The Demon Dungeon adaptation (Dungeon 5), <i>The Legend of Zelda’s</i> The Lizard Dungeon adaptation (Dungeon 2), <i>The Legend of Zelda: Oracle of Ages’</i> Spirit’s Grave Dungeon adaptation (Dungeon 4) . Right: EA-generated dungeon and its parameters. From top to bottom: 32 rooms, 4 keys, 4 locks and linear coefficient of 1,22 (obtained from LC calculations over the <i>Zelda’s</i> dungeon) (Dungeon 1). 23 rooms, 7 keys, 6 locks, and LC of 1,09 (Dungeon 6). 21 rooms, 4 keys, 4 locks and LC of 1,05 (Dungeon 3).	84
Figure 24 – Average of the sum of votes in each group of levels for the Turing test (left) and the evaluation of fun (right). 5 means a strong positive opinion evaluation of fun (strong agreement that the map was made by a human/was going to be fun to play), and 1 means a strong negative opinion (strong disagreement that the map was made by a human/was going to be fun to play).	86
Figure 25 – Examples of rooms created applying SFC and CA. Both top levels were created using Hilbert curve, order 4, <i>width</i> = 2. The left one has 2 doors and the right one has 4. Second row consists of two levels created with Moore curve with the same parameters, but left one has 4 doors and right one has 3. Third row, on the left, contains Hilbert-generated room with same rules, 4 doors, but aided with the CA rule B3/S23. On the right the level is generated with Hilbert, order 4, width 1 and 3 doors. Then, the last row contains another pair of Hilbert-created rooms with 3 doors. On the left, width 3 and order 4, and on the right, width 2 and order 3.	87
Figure 26 – Box & Whisker visualization of rooms generated with different parameters. Orange box is the density, blue one is the exploration value and green one the symmetry. Each 3 sequential metrics correspond to a combination of parameters, identified according to our previously established nomenclature.	89
Figure 27 – Box & Whisker visualization of rooms generated with different parameters, all of them with 2 doors. Blue box measures the density, the green one is the exploration value and the red one is the symmetry. Each 3 sequential metrics correspond to a combination of parameters, which are identified at the bottom of the graph.	90
Figure 28 – Screen-shots from the player’s view in-game. Top-left: the starting room. Top-right: the final room. Bottom-left: a room with an key; Bottom-right: the room with a locked door.	92
Figure 29 – Screenshots from the in-game survey. The previously presented questions were translated into Brazilian Portuguese as the survey was published in medias mainly accessible to Brazilian users.	94
Figure 30 – Average Survey Answers for the First Part of the Experiment	95
Figure 31 – Average Rooms, Locks and Keys Usage for the First Part of the Experiment	96

Figure 32 – Average Ratio of Room Revisiting for the First Part of the Experiment . . .	97
Figure 33 – Bubble Chart Representing the Amount of Votes for Each Fun and Difficulty Combination in the In-game Survey for the LizardZelda (left) and LizardCEA (right) Maps	97
Figure 34 – Bubble Chart Representing the Amount of Votes for Each Fun and Difficulty Combination in the In-game Survey for the MoonZelda (left) and MoonCEA (right) Maps	98
Figure 35 – Bubble Chart Representing the Amount of Votes for Each Fun and Diffi- culty Combination in the In-game Survey for the DragonZelda (left) and DragonCEA (right) Maps	98
Figure 36 – Average Survey Answers for the Second Part of the Experiment	99
Figure 37 – Average Rooms, Locks and Keys Usage for the Second Part of the Experiment	101
Figure 38 – Average Ratio of Room Revisiting for the Second Part of the Experiment . .	101
Figure 39 – The Demon - Dungeon 5	115
Figure 40 – The Demon EA Generated Counterpart- Dungeon 1	115
Figure 41 – The Lizard - Dungeon 5	116
Figure 42 – The Demon EA Generated Counterpart- Dungeon 6	116
Figure 43 – Spirit’s Grave - Dungeon 4	117
Figure 44 – Spirit’s Grave EA Generated Counterpart- Dungeon 3	117
Figure 45 – Lizard Map	119
Figure 46 – Lizard Map Counterpart Created by the CEA	120
Figure 47 – Moon Map	121
Figure 48 – Moon Map Counterpart Created by the CEA	122
Figure 49 – Dragon Map	122
Figure 50 – Dragon Map Counterpart Created by the CEA	123
Figure 51 – Eagle Map	123
Figure 52 – Eagle Map Counterpart Created by the CEA	123
Figure 53 – Lion Map	123
Figure 54 – Lion Map Counterpart Created by the CEA	124
Figure 55 – Snake Map	124
Figure 56 – Snake Map Counterpart Created by the CEA	124

LIST OF ALGORITHMS

Algorithm 1 – Basic Evolutionary Algorithm. Adapted from: (EIBEN; SMITH, 2003) (p. 16)	39
Algorithm 2 – Recursive DFS. Adapted from: (CORMEN, 2008)	42
Algorithm 3 – Recursive Hilbert Curve. Algorithm adapted from (STEPHEN, 2014)	44
Algorithm 4 – Generic Cellular Automaton	47
Algorithm 5 – Generates Random Tree Structures	62
Algorithm 6 – EA	66
Algorithm 7 – Crossover	66
Algorithm 8 – Mutation	67
Algorithm 9 – Crossover with Constraints	71
Algorithm 10 – Mutation with constraints	77

LIST OF TABLES

Table 1 – Number of rooms (d_r) and keys (d_k) obtained after 100 executions of the EA for each combination of features	80
Table 2 – Number of locks (d_l) and Linear Coefficient (d_{lin}) obtained after 100 executions of the EA for each combination of features	80
Table 3 – Number of rooms (d_r) and keys (d_k) obtained after 100 executions of the CEA for each combination of features	81
Table 4 – Number of locks (d_l) and Linear Coefficient (d_{lin}) obtained after 100 executions of the CEA for each combination of features	81
Table 5 – Difference between number of locks in the level and opened by the A* (Δ_l) and difference between number of rooms visited by the DFS (Δ_r) obtained after 100 executions of the CEA for each combination of features	82
Table 6 – Resulting fitness (f) and execution time (Δt) in seconds obtained after 100 executions of the CEA for each combination of features	82
Table 7 – F-Tests followed by t-Tests for all 3 pairs of dungeons for the 3 questions in the survey. All F-tests had a Degree of Freedom of 96 and t-Tests, 192. The t-Tests considered a Hypothesized Mean Difference of 0.0.	86
Table 8 – Exploration (x-axis) vs Linearity (y-axis) Heatmap Visualizations for SFCs with different Order and Width, for SFCs aided with CA, and for levels generated only using the CA. All heatmaps were generated using 10,000 randomly generated levels for each set of parameters. The closer to the white color, more levels have been created that satisfy the corresponding combination of metrics	91
Table 9 – Input parameters and fitness for each CEA-generated dungeon presented in ProCoG.	93
Table 10 – F-tests followed by one-tailed t-tests for all 3 pairs of dungeons for the 3 questions in the in-game survey in the first-half of the experiment with the game prototype. All tests considered the answers from 70 different users and an α value of 5%.	95
Table 11 – F-tests followed by one-tailed t-tests for all 3 pairs of dungeons for the 3 questions in the in-game survey in the second-half of the experiment with the game prototype. All tests considered the answers from 44 different users and an α value of 5%.	100

LIST OF ABBREVIATIONS AND ACRONYMS

ASP	Answer Set Programming
BN	Bayesian Network
CA	Celullar Automaton
CEA	Constrained Evolutionary Algorithm
CR	Crossover Rate
D	Down
DFS	Depth-First Search
E	East
EA	Evolutionary Algorithm
GP	Genetic Programming
KR	Key Room
L	Left
LC	Linearity Coefficient
LR	Locked Room
ML	Machine Learning
MR	Mutation Rate
N	North
NPC	Non-Player Character
NR	Normal Room
PCA	Principal Component Analysis
PCG	Procedural Content Generation
R	Right
RPG	Role-Playing Game
S	South
SFC	Space-Filling curve
Std. Dev.	Standard Deviation
W	West

CONTENTS

1	INTRODUCTION	27
1.1	Context	27
1.2	Gaps and Goals	28
1.3	Hypotheses	29
2	BACKGROUND	33
2.1	Action-Adventure Games	33
2.2	Dungeons	34
2.3	Evolutionary Algorithm	36
2.4	Path-Finding Algorithms	40
2.4.1	<i>A* Algorithm</i>	40
2.4.2	<i>Depth-First Search</i>	42
2.5	Space-Filling Curves	42
2.5.1	<i>Hilbert Curve</i>	43
2.5.2	<i>Moore Curve</i>	44
2.6	Cellular Automata	44
2.7	Procedural Content Generation	46
2.8	Game Engine	49
3	LITERATURE REVIEW	53
4	METHODOLOGY	59
4.1	Dungeon Map Generation	59
4.1.1	<i>Tree Structure</i>	59
4.1.2	<i>Keys and Locks</i>	61
4.1.3	<i>The Spatial Placement: Grid</i>	63
4.1.4	<i>Evolutionary Algorithm</i>	65
4.1.4.1	<i>Crossover</i>	65
4.1.4.2	<i>Mutation</i>	67
4.1.4.3	<i>Fitness</i>	67
4.1.4.4	<i>Evolution and Fix Algorithm</i>	68
4.1.5	<i>Constrained Evolutionary Algorithm</i>	69
4.1.5.1	<i>Constrained Crossover and Mutation</i>	69
4.1.5.2	<i>Path-Finding</i>	71

4.2	Room Generation Techniques	72
4.2.1	<i>Sectioning Space-Filling Curves</i>	73
4.2.2	<i>Adding the Game of Life</i>	74
4.3	Game Prototype	74
5	RESULTS	79
5.1	Dungeon Generation Results	79
5.1.1	<i>EA Results</i>	79
5.1.2	<i>Constrained EA Results</i>	80
5.2	Experiments with Users	83
5.3	Room Generation Results	87
5.4	Experiments with the Game Prototype	91
5.4.1	<i>Part 1 - CEA-Only Maps</i>	94
5.4.2	<i>Part 2 - CEA + SFC Maps</i>	99
6	CONCLUSIONS AND FUTURE WORK	103
6.1	Conclusions	103
6.2	Future Work	104
	BIBLIOGRAPHY	107
ANNEX A	DUNGEONS AVAILABLE IN THE SURVEY	115
ANNEX B	DUNGEONS AVAILABLE IN PROCOG	119

INTRODUCTION

1.1 Context

Video games represent one of the biggest international markets, worth more than US\$100 billion, with a stable growth projection for the next several years. The video game industry is also big in Brazil, the world's 13th largest market, with expected revenue of US\$1.5 billion in 2018. There are more than 2.2 billion players throughout the globe, being 75.7 million Brazilians (MCDONALD, 2017) ¹. Games are usually built around two principles: missions and play space. The former is compounded of an ordered set of goals that a player must accomplish to complete the level. The latter is the physical layout of the level (DORMANS; BAKKES, 2011). Many games have as space a dungeon: sets of interconnected rooms, each one containing different challenges and, generally, presenting a boss monster at the end. The genres of action, adventure, platform and role-playing are examples of games with dungeons as play space.

Aside from the challenges in each room, dungeons can also offer mission elements to the player based on their layout. *The Legend of Zelda* series of games are famous for exploring these elements, specially with locked doors (MIYAMOTO; TEZUKA, 1986)². There are many locked rooms and the players must find and use specific items in order to open them, aiming to proceed with their adventures. Thus, they must remember the location of locked paths as well as to explore the map, adding more play time to the game. This type of mission is categorized as spatial awareness puzzles³ by some designers (ADAMS, 2009, p. 267-268), and also known as spatially situated quests by others (CALVIN; MICHAEL, 2007). These challenges are an important discussion topic for game designers and enthusiasts as shown by the series of videos

---

¹ <<https://newzoo.com/insights/infographics/brazil-games-market-2018/>>

² Locked doors is a term given to any obstacle that prevents the advance of a player through the game until a workaround is found. it can be a set of locked doors and keys or switches that open them (ADAMS, 2009, p. 268)

³ Puzzles which require players to memorize the location of objects related to their avatar

entitled *Boss Keys*, which have more than 700,000 views and focus mainly on dungeon design⁴. The series also present a graph-based dungeon representation containing the missions of the dungeons that inspired this project. We will talk more about such representation in Chapter 2.

The creation of dungeons and the decision where to place locks and keys are commonly a job for professional designers, turning each new dungeon a demanding task for the game's creators. As the production costs can reach the hundreds of millions (FRITZ; PHAM, 2012) and there is a large competition in the game industry, it is strongly advisable that game development studios make use of ways to generate automatically such content without losing quality. The strong competition, as previously stated, can be better perceived following some numbers: more than 7.5 thousand games were launched in 2017 through the *Steam* platform (KUCHERA, 2018), one of the most famous digital distribution systems for personal computers. Also, at least one thousand games are released each month since 2009 for mobile platforms, as can be seen in the *iTunes App Store*, for example. In September 2016 alone, 46 thousand games were released in the platform (POCKETGAMER.BIZ, 2017).

The generation of content for games in an automated way is known as Procedural Content Generation (PCG). Its history, applications and some research studies about the technique are better introduced in Chapter 2, while the most relevant researches related to our work are presented in Chapter 3.

1.2 Gaps and Goals

The field of PCG has many publications, with the interest on the area growing significantly in the last decade. A great number of those researches focuses in the proposition of algorithms to procedurally generate specific types of content. They usually do not integrate algorithms in order to simultaneously and automatically generate different contents. The present master's research aims to contribute by filling such gap. In order to do so, the creation of a set of PCG algorithms is proposed, where each algorithm generates a different content. These algorithms, when put together, are able to automatically create feasible levels with a large amount of content. This approach will be evaluated through the combination of PCG algorithms aimed to provide content for both mission and space of games set in dungeons. Therefore, we integrate two different PCG algorithms. The first one uses a set of input given by a designer to create the dungeon's layout together with the placement of locks and keys. The goal is to generate levels as close as possible to the desired input parameters, by allowing the generation of missions in sync with the play-space. The second algorithm is in charge of rooms creation, making each of them unique for the player to go through. By using both methods, it is possible to create the whole play-space of dungeons and locked-door missions.

The first algorithm aims to generate the dungeon layout along with its missions in an

⁴ <https://www.youtube.com/playlist?list=PLc38fcMFeV_ul4D6OChdWhsNsYY3NA5B2>

efficient and adaptive way. That is, without using predefined grammar rules and without assuming a large amount of complex restrictions. The second algorithm, which creates the contents of rooms, aims to apply Space-Filling curve (SFC) algorithms in the PCG context. This will be done by evaluating the benefits of SFC together with Celullar Automaton (CA). As far as we known, the SFC with CA proposal for PCG was not found in the literature except by the recent work of Macedo and Chaimowicz ([MACEDO; CHAIMOWICZ, 2017](#)). The authors use space-filling curves as an auxiliary method to create levels with cellular automata.

As a proof of concept for our approach, we developed an adventure game prototype using the Unity game engine, the most popular among game developers and one of the market's leaders ([UNITY, 2018](#)). This game was then coded to gather gameplay data for each dungeon and hosted in a server, leading to our final experiment with players to validate our algorithm.

1.3 Hypotheses

The main hypothesis about the first algorithm described in the previous section is that an Evolutionary Algorithm (EA) can create both the play-space and missions of a dungeon without the need of a previously defined ruleset or a large amount of restrictions. Such hypothesis is supported by positive results found in the literature in works that apply EAs for level generation ([MCGUINNESS; ASHLOCK, 2011](#); [KHALIFA *et al.*, 2016](#); [LIAPIS; YANNAKAKIS; TOGELIUS, 2013](#)). We highlight the work of Valtchanov and Brown introducing an EA able to evolve a level encoded in a tree structure. The encoding is translated in a 2D grid of rooms, similar to our representation, as detailed in Chapter 4. Although both researches use similar structures to represent a level, the major difference is that ours is focused in creating locked-door missions, turning the whole dungeon into a challenge, while Valtchanov and Brown's work aims the creation of the level's layout and the setting of some rooms as special ones, where some desired missions may be placed inside. Therefore, in their work, the authors do not take the level itself as a possible challenge for the player ([VALTCHANOV; BROWN, 2012](#)).

The recent work of Liapis presented an EA able to create missions and levels, but only through the use of many constraints. However, according to the authors, these constraints can be difficult to satisfy. Our goal is to create both contents with as few constraints as possible, but still managing to suit the designer's needs. Moreover, Liapis' work also handles the positioning of enemies and items, while ours handles the locked-doors missions ([LIAPIS, 2017](#)).

Thus, we hypothesize the algorithm is able to create a supporting tool for the game designer, providing a large variety of levels based on a set of predefined features. These levels already have locked-doors missions in their structure, a novelty in literature by the time we published the early results of our work ([PEREIRA; PRADO; TOLEDO, 2018](#)), and it can help to reduce costs and time during level design for dungeon and puzzle-based games. The recent work of Ruela and Delgado also generates locked-door missions together with the play-space of

grammars, statistics, hand-made patterns or fractals. Aiming to create something attractive to players, we decided to use space-filling curves, supported by cellular automata, to add diversity in the generated rooms. This approach is already known in the industry as in the Galak-z game discussed in Chapter 3. In Galak-z, the developers tested different approaches to procedurally generate levels, some of which are cellular automata and Hilbert curves (UNITY3D, 2014). Also, space-filling curves were recently introduced in scientific papers by Chaimowicz and Macedo (MACEDO; CHAIMOWICZ, 2017). More details on this approach can be found in Chapter 4, and a dungeon generated using the combination of both map and room generation techniques is presented in Figure 2.

Figure 2 – Example of a dungeon with both its map layout and rooms generated by our algorithms.

The present thesis is divided as follows: Chapter 2 defines the main concepts related to PCG and the game genres used in the research, allowing a better understanding of the work. The following chapter, 3, holds a literature review of the main works related to the thesis, and Chapter 4 presents the research's methodology for the proposed algorithms. Next, Chapter 5 shows the results obtained during the experiments. The last chapter, 6, reports the final thoughts about the project and conclusions from the results achieved by this work.

BACKGROUND

In this section we present the concepts behind the main themes guiding the present research. First we introduce the action-adventure game genre, the one which the game prototype developed for this research belongs to. The concepts of game's dungeons follow, as well as the main challenges the player must overcome inside them, and a model to represent said dungeons. Next, we present Evolutionary Algorithms that will be used in the generation of dungeon maps. We also describe some search algorithms, specially the A* and Depth-First Search, both applied to evaluate the levels in this work. Also, the principles of Space-Filling Curves and Cellular Automata are presented, since they are used to generate the contents of the rooms by our algorithm. Finally, the main ideas about Procedural Content Generation are shown, and a brief introduction of game engines is provided.

2.1 Action-Adventure Games

The action-adventure game genre demands great focus in physical challenges, including reflex and motor coordination similar to action games, but also includes many elements from adventure games. Some of these elements are: item gathering, exploration and interaction with the environment (and the people inhabiting it). The story usually takes place in a large world connecting different important areas, and the game requires puzzle solving of different types (ROLLINGS, 2003; RYAN, 2002; LUBAN, 2002). For most games, the player controls a single avatar that is the protagonist of the story. This mix of elements brings the genre close to the Role-Playing Game (RPG), although the latter generally focuses more in story-telling, level and abilities system, which are not present in the majority of action-adventure games (ADAMS; ROLLINGS, 2006).

It is important to highlight that the genre definition for games are not exact and vary between authors and players. Also, many definitions valid for overall genre classification may conflict with the gameplay of specific games, demanding a case-by-case study.

The most referred game as the progenitor of the action-adventure genre is the *Adventure* game (ROBINETT, 1979), an adaptation of a text adventure game which allowed the player to fight battles as in action games. Another landmark for the genre was the game *The Legend of Zelda* from 1986. The game was a huge commercial success and one of the first to skillfully combine the main elements of the genre. *The Legend of Zelda* brought exploration of a world much larger than usual at that time with transport and inventory puzzles, combats based on action mechanics, currency system and a level system close to the ones in RPGs but without experience points. It was the first title of one of the most famous game franchises, currently consisting of 19 main titles, all of the action-adventure genre.

Nowadays the genre has a roster of many successful games with many nominated for The Game Awards in 2017. Some of them were: *The Legend of Zelda: Breath of the Wild* (Figure 3, top) (FUJIBAYASHI, 2017), *Horizon Zero Dawn* (Figure 3, bottom)(JONGE, 2017) and *Super Mario Odyssey* (MOTOKURA, 2017), all in the Game of the Year category (along others), with *The Legend of Zelda: Breath of the Wild* being the winner. Also, *Hellblade: Senua's Sacrifice* (ANTONIADES, 2017), *Uncharted: The Lost Legacy* (ESCAYG, 2017), *Metroid: Samus Returns* (HOSOKAWA, 2017) and *Nier: Automata* (TARO, 2017), all action-adventure games, were nominated for other categories.

Aside from their great commercial success, the games from the genre also hold an important place in the research area, being a test platform for many PCG algorithms as it will be better reported in Chapter 3. The many types of puzzles coming from action-adventure games bring varied and interesting challenges for researchers, which use the most different approaches ranging from evolutionary algorithms to statistical learning and grammars to solve them (SUMMERVILLE; MATEAS, 2015; SUMMERVILLE *et al.*, 2015; LIAPIS, 2017; VALTCHANOV; BROWN, 2012; DORMANS; BAKKES, 2011).

One of the most common challenges of the genre is to eliminate all enemies in a given area. This task turns out being a logical puzzle in many situations, as the majority of enemies in the games are weak to specific weapons or items, or need to be attacked in a specific time window. Other more straight logical challenges are: solving spatial puzzles by pushing objects around the level to proper locations, some times with movement restriction; solving logic puzzles varying from common logical and mathematical problems recreated in the play-space, as using a specific item's attribute to access a new area; and the spatial awareness puzzle, or locked-doors mission, already defined in Chapter 1.

2.2 Dungeons

Dungeons are the main play-space for most adventure, action-adventure and RPGs. They are build with an entrance, many intermediate rooms and a final room with a challenging boss monster, which may protect a rare item or key used for game progression and/or to reach new

Figure 3 – *The Legend of Zelda: Breath of the Wild* (top) and *Horizon Zero Dawn* (bottom), two award-winning action-adventure games released in 2017

areas. The intermediate rooms commonly hold different challenges, varying with the individual mechanics of each game. In general, the challenges are random encounters with monsters (which is the case for RPGs); puzzle challenges of many types (for example, move objects to open a passage or use specific items in certain objects); and fights with enemies present in the level (for adventure and action-adventure games). Also, it is commonplace to find items scattered throughout the rooms and, in some cases, switches that allow access to other parts of the dungeon. The items spread across rooms can be keys, consumable items or equipable items and they can be used to unlock passages to other rooms in some games.

Our project focuses on procedurally generating content to this last set of challenges: finding items scattered throughout the dungeon's rooms. *The Legend of Zelda* game franchise

is based on this kind of mission, along with puzzles and enemies spread inside some rooms. However, the puzzles and enemies placement are beyond the scope of the current research. As already stated, we intend to create dungeons with different features related to the positioning of rooms, and the distribution of items able to open locked paths and their locks. The *Boss Keys* series of videos makes a deep analysis of this type of play-space and missions, including the creation of appealing visual representation through the use of graphs to represent the different shapes of dungeons, their keys and locks. Figure 4 shows two dungeons, from *The Legend of Zelda: The Minish Cap* game (FUJIBAYASHI, 2004), converted into the graphs suggested by the creator of *Boss Keys* series.

In the upper graph of Figure 4, we can see the dungeon with many branches and wider path choices for the player. This demands the player to backtrack its way in the dungeon, that is, going back to an already visited room in order to reach the goal. The lower graph in Figure 4, however, has a linear structure that becomes unnecessary for the player to backtrack for most parts of the dungeon. The dungeons from the first kind usually have more puzzle-based challenges, while the second one holds more action-based challenges by placing more and stronger enemies in its rooms ¹.

The ability to allow diversity in the content of rooms and their placement in the dungeon is very important to allow a PCG method to be general enough when creating both types of dungeon. This characteristic is taken into account by the algorithm presented in our research, as described in Chapter 4. The graph representation also allows to easily observe another goal of our algorithm being used in a commercial game: the existence of unique pairs for lock-key combination. Figure 5 shows different pairs with different colors (diamonds are keys and squares are locks), and the order for the player to solve the challenges until the completion of the dungeon, which is the upper one in Figure 4.

2.3 Evolutionary Algorithm

Evolutionary Algorithms (EAs) are stochastic meta-heuristics for search and optimization which are based upon Darwin's theory of evolution. Although there are many subgroups and branches of the main concept of the algorithm, all follow the same *modus operandi*: a population consisting of different individuals is created by a set of rules (usually, using an uniform random distribution). Then, the selection for breeding and survival of individuals is simulated for many generations, until a specific stop criterion has been satisfied. The individuals are representations of possible solutions for the problem, coded in a myriad of different ways. The individuals are evaluated through some kind of metric, mostly known as a fitness function. The theory-of-evolution-based features of the EA are mostly found in the process of survival and breeding selection, in which the best individuals have greater chance to breed and/or be kept for the next

¹ <<https://www.youtube.com/watch?v=KEVJXqV7XMc>>

Figure 4 – Graph abstraction of 2 levels from *The Legend of Zelda: The Minish Cap* game. Only rooms containing items which are essential for the player's progression are shown. Each color identifies a different type of key-lock pair. Keys are represented as diamonds and locks as squares. The vertical lines show a sequence of rooms in the same path, while the horizontal ones identify a path's branching. Many rooms in the same horizontal line may be visited by players, if they wish so. Extracted from: <https://www.youtube.com/watch?v=KEVJXqV7XMc>

generation. Figure 6 shows a diagram representing the basic procedures of an EA.

For the general case, the parent selection for breeding happens by comparing the fitness between individuals from the population, with the better ones having more chances to take part in the recombination process. This fitness is calculated by a function defined specifically for the problem which needs to be solved. After an amount of parents has been chosen (in most cases, a couple), the breeding procedures of crossover and mutation are executed since a probability condition is met. The crossover of individuals will shuffle pieces of the parent individuals to

Figure 5 – A possible solution for the puzzle in the upper dungeon from Figure 4. In the upper left image the first key is found and used in the first lock. In the upper right one, two levers are used to open the green door. In the lower left image, the player finds the bow and opens the first door requiring its use. At last, in the lower right image, the player gets the boss key and opens the last lock. Extracted from: <https://www.youtube.com/watch?v=KEVJXqV7XMc>

produce one or more new individuals. Next, the novel individuals may be mutated, which usually means a random change adding further variability to the representation of the individual.

After the offspring individuals are created, the selection of the survivors begins by selecting a proportional mix of old and new individuals, where the survival of the fittest individual is prioritized. The algorithm ends after a stop criterion that can be: the execution of a fixed number of generations, the best possible solution has been found, there was no improvement in the best solution for a fixed number of generations, and others. The Algorithm 1 exemplifies the execution of an EA with the most common procedures in literature (EIBEN; SMITH, 2003).

Different representation (codification) for individuals have appeared according to the types of problems to be solved. The main codifications are: strings of characters from a finite alphabet, belonging to the research field of Genetic Algorithms; real valued vectors, known as Evolution Strategies; finite state machines, in Evolutionary Programming; and trees in Genetic Programming (GP).

Figure 6 – General representation of an Evolutionary Algorithm. Extracted from: (EIBEN; SMITH, 2003) (p. 17).

Algorithm 1 – Basic Evolutionary Algorithm. Adapted from: (EIBEN; SMITH, 2003) (p. 16)

```

1: INITIALIZE population with random solutions
2: EVALUATE each individual
3: while STOP CRITERION not satisfied do
4:   SELECT parents
5:   CROSSOVER pairs of parents
6:   MUTATE resulting children solution
7:   EVALUATE new individuals
8:   SELECT individuals for the new generation
9: end while
  
```

As the codification of individuals presented in our EA is based on trees, the algorithms used in GP can be applied in our problem. Therefore, they are briefly discussed next and the ones used in our research are highlighted. It is common for populations in GP to have thousands of individuals, leading to a slower convergence. As will be discussed in Chapter 5, that is not the case for our algorithm whose population is relatively small.

The most common crossover algorithm for GP is the swap of branches between trees, and the most common mutation method is to make random changes in the tree's structure. Both are used in our algorithm, as it will be shown in Chapter 4. The parent selection generally is relative to the fitness using ranks. However, we use tournament selection in our approach. The survivors' selection is given by replacing the whole previous generation for the new one, which is the one we use. Another important feature of GP kept in our work is the variability between individuals length and depth, essential for dungeons representation.

2.4 Path-Finding Algorithms

Path-finding algorithms find the shortest path between two given points, usually in a graph or tree structure. Its most basic routine consists of starting to explore a graph from an initial vertex until the destiny is found, generally searching for the path with the lowest cost. The path-finding problem can be solved using simpler algorithms as breadth-first search and Depth-First Search (DFS) within complexity $O(|V| + |E|)$, being V the number of vertexes and E the number of edges. The related shortest path demands more computational complexity as the *Bellman-Ford* algorithm with $O(|V||E|)$ complexity, or *Dijkstra's* algorithm that takes $O(|E| + |V|\log|V|)$ by using dynamic programming to eliminate paths and Fibonacci stack in its implementation (FREDMAN; TARJAN, 1984).

We now present the A^* and the DFS algorithms in details, as both were used to navigate the dungeons generated in our research.

2.4.1 A^* Algorithm

The A^* is a variation of *Dijkstra's* algorithm and it is widely used in video games. The algorithm gives a weight to each open node equals to the weight of the edge leading to that node plus the approximated distance of the current node to the goal. This approximation is given by an heuristic and allows the algorithm to discard the longest paths after finding an initial route, making the A^* more efficient than *Dijkstra's* algorithm. The choice of the right heuristic function is very important once the closer it is to the real distance, the algorithm will evaluate less not-optimal nodes. Thus, the most efficient heuristic for an A^* algorithm is one which can give the real distance between nodes. Unfortunately, this demands a lot of computation time (MATHEW, 2015), so an admissible heuristic is chosen. That is, a function that never overestimates the cost of reaching the goal. The A^* is widely used in games for its relatively easy implementation and great efficiency. However, the game environments are usually complex and hold a large number of characters that need to execute the algorithm simultaneously. Thus, many optimizations for the algorithm, the search space, the heuristic function and the use of memory are needed. The work of Cui and Shi, “ A^* -based Path-finding in Modern Computer Games” shows some of those optimizations and many uses of A^* in games (CUI; SHI, 2011). The time complexity of the algorithm will depend on the heuristic used to estimate the remaining distance. In the worst case, it is $O(b^d)$, where d is the depth of the solution and b the average number of successors per state, that is, the branching factor.

We now present an example of the A^* algorithm to clarify it for the reader. An implementation of the algorithm that can be executed from the browser and accepts different configuration inputs can be found in Github². In Figure 7 it is possible to observe 4 steps of the A^* 's execution in the same map. This map consists of a green tile representing the starting

² <<https://qiao.github.io/PathFinding.js/visual/>>

point, a red one which is the goal, and blue ones standing for walkable tiles. It can be seen, in the upper left image, that all neighboring cells from the starting nodes are highlighted in green, with arrows pointing to the direction of the movement the object must do in order to go from the starting cell to the others. The set of tiles highlighted in green is the one called “open” tiles and it is the one evaluated at every iteration, while searching for the best route. The 3 values shown in each tile are the values usually referred as F , G and H . The value G , shown in the bottom left corner of each square, is the cost needed to move from the starting node (green) to the actual tile, following the route generated by the algorithm to reach the tile. The value in the bottom right corner, labeled as H , is the estimated cost given by the heuristic of moving from the actual tile to the objective (red). This is a prediction of the remaining distance to reach the goal. At last, the upper left corner of the tiles contains the value for F , which is the sum of G and H . The green-highlighted tile, with the lowest value for F in each iteration, is the one chosen to be part of the actual path.

Figure 7 – A* execution example. Extracted from: (LESTER, 2005).

In the upper left image from Figure 7, we can observe the tile standing at the right side of the starting node highlighted in cyan. Thus, it was chosen the tile with the least distance to the goal (lowest value of F), in this case, 40. This tile is added to the list of “closed” nodes (hence the cyan highlight). In the upper right image we can observe the path has been altered: the node which was closed in the previous step did not have a connection to any not-opened node. This can be seen by looking at the arrows, where none originated from it. These arrows show the movement direction leading to the shortest path from the starting point, the path which gives

the value of G . Therefore, if a visited node holds a lower value of G , when moving to one of its neighbors than the actual G value of that neighbor, the neighbor's G value is updated as well as the direction of the arrow in this neighbor node. Following the algorithm's execution, we can see that the goal has been reached in the bottom left image. The path is shown in the bottom right image, following the direction of the arrows starting from the goal tile (in red) until the starting tile is reached. By doing this, we find the path is composed of the 5 tiles with a red dot in its center, and the shortest distance is of 68 units of measurement.

2.4.2 Depth-First Search

The Depth-first Search is an algorithm used to traverse or search a tree or graph data structure. It starts in the root node for trees, or in a selected starting node in graphs, and explores as far as possible along each branch before it backtracks. The algorithm takes linear time in the size of the graph $O(|V| + |E|)$. If there are multiple nodes to be chosen in the actual branch, the algorithm may follow a convention or visit a node randomly (CORMEN, 2008). Algorithm 2 describes a recursive implementation of DFS. The algorithm has some interesting applications in the creation and solution of mazes, as it guarantees all nodes are visited (SYEED *et al.*, 2010; KOZLOVA; BROWN; READING, 2015). For very large or infinite trees and graphs, the original DFS may not work properly, but some adaptations such as the Depth-First Iterative-Deepening can be used (KORF, 1985).

Algorithm 2 – Recursive DFS. Adapted from: (CORMEN, 2008)

Require: Graph G and vertex v of G

```

1: function DFS( $G, v$ )
2:   label  $v$  as discovered
3:   for edges from  $v$  to  $w$  in  $G.\text{adjacentEdges}(v)$  do
4:     if vertex  $w$  is not labeled as discovered then
5:       DFS( $G, w$ )
6:     end if
7:   end for
8: end function

```

2.5 Space-Filling Curves

Space-filling curves are a special case of fractals able to completely fill a n -dimensional space. In the specific case of 2D-filling curves, they are generically called Peano curves, homage to the mathematician who first discovered these curves. These curves have the property to map information from one dimension to another while preserving locality. Many different curves were discovered as Hilbert's, Sirpińskis, Lebesgue, among others. They are applied to solve problems in a wide range of areas (SAGAN, 2012). The curves have been recently used to model crystals in a machine learning algorithm (JASRASARIA *et al.*, 2016), to solve global optimization problems

by reducing the problem's dimension (LERA; SERGEYEV, 2015), to optimize the running time of Delaunay triangulation algorithms (SU *et al.*, 2016) and also to compress images (OUNI; LASSOUED; ABID, 2015).

Its practical application in PCG is also known thanks to the game Galak-Z, mentioned in Chapter 1. The game's developers presented PCG tests both using Hilbert and Lebesgue curves. Even so, until the present date, only the work of Macedo and Chaimowicz applied the curves in scientific literature (MACEDO; CHAIMOWICZ, 2017). Their work is better discussed in the next section. Next, we present the steps needed to create Hilbert and Moore curves since the current research will employ both.

2.5.1 Hilbert Curve

A Hilbert curve, first described by David Hilbert, is a continuous fractal space-filling curve, variant of the Peano curves. One way to implement the Hilbert curve is starting at order 0, which is empty and represented by points in space. An order 1 curve is just a single cup given by lines connecting the previous points. Next, each increasing order of the curve is made as follow: a Hilbert Curve of order m is defined by 4 other Hilbert curves of order $m - 1$ being connected by 3 lines. Each of the curves occupies a different corner in the unit square and the 2 curves in the top are rotated in relation to their original orientation. The top left one is rotated 270° and the top right one rotates 90° . A total of 3 connecting lines are created: 2 connecting each top curve with the one below it, and the last one between the bottom curves. After the connection, the higher order fractal is created. Figure 8 shows the process.

Figure 8 – Recursive creation process of an order 3 Hilbert curve starting from an order 2 curve. Adapted from (NØRMAK, 2013).

Algorithm 3 describes a recursive way to create Hilbert curves with any order and desired distance between points. For every iteration of the recursive function *Hilbert*, it executes four times the operation of draw lines as illustrated in black at the second step of Figure 8. Each call of the function triggers four new calls of itself with lower order curves, where each level of

recursion reduces 1 unity of the current order until it holds the value 0. The order-0 curve is the basic component with 4 points.

Algorithm 3 – Recursive Hilbert Curve. Algorithm adapted from (STEPHEN, 2014)

Require: Curve's order *ord*, and direction vector to draw the line *dx* and *dy*.

```

1: function HILBERT(int ord, int dx, int dy)
2:   if ord  $\geq$  1 then
3:     Hilbert(ord - 1, dy, dx)
4:     DrawLine(dx, dy)
5:   end if
6:   if ord  $\geq$  1 then
7:     Hilbert(ord - 1, dx, dy)
8:     DrawLine(dy, dx)
9:   end if
10:  if ord  $\geq$  1 then
11:    Hilbert(ord - 1, dx, dy)
12:    DrawLine(-dx, -dy)
13:  end if
14:  if ord  $\geq$  1 then
15:    Hilbert(ord - 1, -dy, -dx)
16:  end if
17:  if ord  $\leq$  0 then
18:    DrawPoints()
19:  end if
20: end function

```

2.5.2 Moore Curve

Another well-known SFC is the Moore curve. It is a variation of the Hilbert one by creating a procedure that joins, after some processing, four Hilbert curves of a lower order. This is done in such a way that the endpoints of the curves will coincide. Figure 9 shows the 6 first orders of the curve. The curves are placed in each corner of the unit square, rotating both on the left by 270° and both on the right by 90° . Next, the bottom ones are linked to the ones above them, and the top ones are linked between each other. The pseudo-code for this algorithm is a slight modified version of Algorithm 3 and it was omitted of this document.

2.6 Cellular Automata

Cellular Automata are grids of regular cells, each within a finite set of states. An initial state is defined by setting the status of each cell and referred by $t = 0$. A neighborhood is defined as the closest cells according to some arbitrary rule. A mathematical function can be used as a rule to change the current state of one cell and its neighbors from one generation to another. The same rule is usually applied to all cells and does not change over time (WOLFRAM, 1983; SCHIFF, 2007; ILACHINSKI, 2001).

Figure 9 – The 6 first orders from Moore curve.

Figure 10 – Creation process of an order 3 Moore curve starting from an order 2 Hilbert curve. Adapted from (NØRMAK, 2013).

The Automata can be divided in 4 different classes. In class 1 almost all starting patterns quickly evolve to a stable and homogeneous state, making any randomness from the initial state disappear. Some randomness is kept in class 2, where all starting patterns evolve quickly in stable or oscillatory structures³. This is likely to preserve local changes in the starting pattern of the local region. Class 3 is compounded by automata with all starting patterns evolving in a pseudo-random or chaotic way. In this class, any emerging stable structure is quickly destroyed by the surrounding noise. Also, local changes tend to spread indefinitely. At last, almost all starting patterns evolve in structures in class 4 that interact in a complex and interesting way, creating local structures capable of surviving a long period of time. Local changes also tend to spread indefinitely (ILACHINSKI, 2001). It is important to note that this classification is not exact, and some rules can fulfill criteria from more than one class (WOLFRAM, 2002).

The number of possible configurations from a certain neighborhood in an automaton is equal to the number of states in a cell to the power of the number of cells in the neighborhood

³ In a cellular automaton, an oscillatory structure, or oscillator, is a pattern that returns to its original state (both in position and orientation) after a finite number of generations. Therefore, the evolution of the pattern is repeated indefinitely.

(including the current cell). For instance, an automaton with 2 states and *Moore's* neighborhood (all 8 adjacent cells, including the diagonals, are considered neighbors) has $2^9 = 512$ possibilities of configurations. This number can increase drastically by considering a wider neighborhood. For a size 2 *Moore's*, we have $2^{25} = 33.554.432$ possible configurations. This large number of possibilities, added to its wide use in many different areas of research such as image processing (QIN *et al.*, 2015), climate dynamics (PARDO-IGÚZQUIZA; COLLADOS-LARA; PULIDO-VELAZQUEZ, 2018), cryptography (MAITI; GHOSH; CHOWDHURY, 2017), and many others, have caught the attention of PCG researchers (SHAKER; TOGELIUS; NELSON, 2016).

There are several works in the literature that applied CA to generate game contents such as (JOHNSON; YANNAKAKIS; TOGELIUS, 2010; ADAMS; PAREKH; LOUIS, 2017). However, as already mentioned, only the work in (MACEDO; CHAIMOWICZ, 2017) combines both CA and SFC. These related works will be discussed later in Chapter 3.

In our research we use the Conway's Game of Life, a class 4 CA, in order to create rooms for dungeons. Conway's Game of Life is one of the most widely known CA and it present two states (alive or dead), following 4 rules (GARDNER, 1970):

1. Live cells with fewer than two live neighbors die.
2. Live cells with 2 or 3 live neighbors live on.
3. Live cells with more than 3 neighbors die.
4. Dead cells with exactly 3 neighbors become live cells.

Since the creation of Conway's Game of Life, many variations have been presented by changing the number of cells needed to apply the rules. A standard game of life is symbolized as *B3/S23*, since a cell is Born (*B*) if it has 3 living neighbors, Survives (*S*) if it has 2 or 3 living neighbors, and dies otherwise. The Game of Life uses a Moore neighborhood, i.e., the eight adjacent cells (including the diagonals) are considered neighbors. Algorithm 4 describes the general process of a CA.

2.7 Procedural Content Generation

The technique know as Procedural Content Generation was originally developed in order to reduce the memory usage by game content, allowing more content to be inserted, since many of them were limited by the capacity of cartridges and floppy disks. By the time of its creation, in the 80s, computers had little memory and processing power, so this was a must for improving game content. Trying to break the limits from the other games, the developers from *Rogue* and *Elite* were one of the pioneers using PCG. *Rogue*, a dungeon crawler game, generated levels only when the player was about to enter them. They were created using algorithms which defined

Algorithm 4 – Generic Cellular Automaton

Require: Grid g with dimensions gX and gY , and with each cell being either alive (1) or dead (0). Number of generations n to execute. Actual generation a initialized with 0. Number of neighbors to satisfy born condition b , to satisfy survive condition $s1$ and $s2$.

```

1: g.Initialize()                                ▷ Random initialization of each cell
2: while  $a < n$  do
3:   for Each cell  $c$  in  $g$  do
4:      $l \leftarrow c.GetLivingNeighbors()$ 
5:     if  $c = 0$  AND  $l = b$  then
6:        $c \leftarrow 1$                                 ▷ Cell is born
7:     else if  $c = 1$  then
8:       if  $l \neq s1$  AND  $l \neq s2$  then
9:          $c \leftarrow 0$                                 ▷ Cell dies
10:      end if
11:    end if
12:  end for
13: end while

```

features such as: size of the current room, the connections between rooms, location of items and enemies, placement of the level's entry and exit (DESIGN, 1984). *Elite* is a game about space combat and commercial trade between planets. Its 256 planets for each of the 8 galaxies were procedurally generated from a random seed (BELL; BRABEN, 1984).

The original motivations to use PCG turned obsolete with the improvement of memory, processing speed and capacity of computers. However, the advancements of technology came together with a massive expansion of the number of players, the complexity of games, their contents and, accordingly, the increase in cost and production time of games. This made the game developers look back to PCG, but with novel goals: create large amounts of diverse contents as fast as possible and, in some cases, adapting them to different types of players. This aims to add more replay value for games by creating different missions and levels in each new playthrough (IOSUP, 2011; HENDRIKX *et al.*, 2013). Some recent games that use PCG are *No Man's Sky* (GAMES, 2016b), *The Binding of Isaac* (MCMILLEN; HIMSL, 2011), *Elite: Dangerous* (DEVELOPMENTS, 2014) and *Civilization VI* (GAMES, 2016a). We highlight that the use of PCG in the game industry is usually made possible by applying random algorithms, without adapting directly to the user's profile, or by randomly selecting elements from a database of assets (levels, models, textures or any other type) previously created by human developers. They can also use available algorithms for procedural art and animation generation, which differ from the other types of PCG approaches. The developers generally create algorithms tailored for their specific game, making it quite difficult to reuse in other projects.

The game *Civilization VI* applies procedural generation of models to create different contents, including farms. Also, the sea coasts from the maps are created automatically using spline curves, but with its randomness limited by artists. The visual effects from cracking ice

next to the coast were also created procedurally ⁴. *Elite: Dangerous*, on the other hand, uses PCG in two different ways. The first is generating the planets' ecosystems. Artists created a set of rules, together with programmers, that when applied to planets was able to provide a visually pleasing and realistic ecosystem ⁵. The second one takes care of creating star systems, called *Stellar Forge*. The developers implemented many rules based on astronomy to simulate a wide range of realistic physical and chemical reactions, in order to create unique systems for each combination of parameters, extracted from random seeds ⁶.

The game *No Man's Sky* was acclaimed for an ambitious PCG approach: it was able to procedurally generate an entire universe, with almost 18 quintillion planets, each with a flora and fauna also created through algorithms. Its music was also procedurally generated. In order for each creature to be unique and believable, the developers created a set of feasible resources for each part of the body (e.g., different heads, legs, bodies, etc.). Taking into account what elements were already part of the body, some elements of the set of yet-uncreated parts were marked as non selectable. After the selection of an element, its child components were then selected. For example, after the selection of a type of head, a mouth must be chosen for the creature. However, only those compatible with the actual head remain in the selection pool for the algorithm ⁷.

Finally, the game *The Binding of Isaac* applies PCG to create its levels in the shape of dungeons, as the ones in this research. Although relatively older than previous examples, the game was remade and launched in 2014 as *The Binding of Isaac: Rebirth* (MCMILLEN, 2014) and it had two expansions as downloadable content: one at 2015 and another at 2017. It was also ported for consoles of the current generation, having a life cycle that lasted until 2017. Part of this longevity and success is thanks to the high challenge and replayability factors provided by PCG. Its algorithm depends a lot on human work: more than a thousand rooms were created by the game designer, ordered according to difficulty and chapter which they belong to (an aesthetic factor), and randomly chosen to be placed in the dungeon. The content of each room is drawn between more than 50 enemies in the game and items, that have different probabilities to be placed in each room according to their relevance. Enemies also vary according to luck: the same enemy can be created, with lower probability, as a stronger specimen that can have different colors (also drawn randomly) each with a unique ability ⁸. The game development still demands a lot of effort from the creators by generating the initial content, but the PCG algorithm of the game, even though being simple, is able to return great variability and commercial success: the original title sold more than 3 million copies in the *Steam* platform, and its remake more than 2 million copies ⁹.

⁴ <<http://firaxis.com/?/blog/single/procedural-terrain-generation-in-sid-meiers-civilization-v>>

⁵ <<https://www.youtube.com/watch?v=GEVutbSqBI0&t=466>>

⁶ <<https://us2.campaign-archive.com/?u=dcbf6b86b4b0c7d1c21b73b1e&id=76df98203b#StellarForge>>

⁷ <<http://3dgameblog.com/wordpress/?p=836>>

⁸ <<http://edmundmcmillen.blogspot.com.br/2011/09/binding-of-isaac-gameplay-explained.html>>

⁹ <<http://steampy.com/search.php?s=binding+of+isaac>>

The academic research in the area is also growing in recent years, with a large number of papers published since 2010. The main focus is on play-space generation (the dungeon and levels), and a lesser focus on missions. Some techniques try to create both contents simultaneously, most through the use of grammars (DORMANS, 2010; MERRICK *et al.*, 2013; LINDEN; LOPES; BIDARRA, 2013). These results have good quality as game's content, but the approach demands that a designer build the grammar and its rules. Moreover, the generated levels will always belong to the same subset allowed by the grammar. We try to deal with play-space and mission generation in our PCG approach. The related works from literature will be discussed in Chapter 3.

2.8 Game Engine

A game engine is a software framework designed to aid programmers, artists and game designers to create, implement and build video games. It is mainly composed of a render engine for 2D and/or 3D graphics, a physics engine or a collision detection and response engine, sound engine, script language, animation, artificial intelligence, network connection, data streaming, memory manipulation, thread generation, localization support and scenes graph. However, the complexity of engines has increased over the years, giving even more functionality to them (GREGORY, 2014, Chapter 1). The ability to use a single game engine to create many different games saved a lot of resources in time and money for game development companies, instead of the old need to develop an engine (or heavily alter a previous one) for each new game. Most engines nowadays allow the user to build games for many different platforms with just a few (or even without) alterations in the project, saving a lot of time (PEDERSEN, 2009).

Game engine's originated in the middle of the 90's, thanks to the software architecture from the game *Doom* (HALL, 1993). It was able to separate successfully its main software components, its art assets, game worlds and rules which affected the player's experience. Since then, the value of this separation of components was clear, once the developers started to license new games and rebuild them in new products by creating new art assets, world layouts, weapons, characters, vehicles and game rules, but only making minor changes in the engine's software. At the end of the 90's, some games like *Quake III Arena* (JAQUAYS, 1999) and *Unreal* (BLESZINSKI, 1998) were made with re-usability in mind. Their engines were highly customizable through script languages, and the licensing of such motors turned out being a viable secondary means of profit for their creators (GREGORY, 2014, Page 11).

Nowadays there are many game engines available in the market, each with its own business model, solutions, architectures, script languages, functionality and approaches to enable the usage by non-programmers. Three of the main engines will be presented briefly, together with the reasons why the *Unity* engine was chosen by our project.

The *Unreal Engine*, now on version 4, is one of the most powerful engines and one

of the oldest (since 1998, when the game *Unreal* was released). As its most relevant business competitor, *Unity*, both are used for applications beyond gaming as apps for enterprises, simulations, cinematic experiences and movies. However, their focus is creating high quality games for a plethora of platforms, including virtual and augmented reality. Some of the unique selling points for *Unreal* are the realtime photoreal rendering; the full C++ source code being available, allowing for customization; a *Blueprint* system for visual scripting, allowing designers to make many contents without knowing how to code; robust multiplayer framework; and many others. Its business model allows free use of the software, with all its functionality. However, 5% of all revenue from the games made with the engine goes to the company. The engine is aimed for both small and large development teams, therefore, the engine is used in many independent games developed by smaller studios, with a handful of developers and usually lower development costs. But it is also used to develop *Triple A* games, made by large studios, which spend a large amount of investment and employ many developers (GAMES, 2018).

The *Unity* engine is also destined for both 2D and 3D games, teams and projects of different sizes and it has many state-of-the-art functionality to help create *triple A* games. There is no native visual script in *Unity*, but there are third-party alternatives in its asset store. *Unity* is well-known for trying to bond deeply with its users and attract new users with the *Unity User Groups*¹⁰, the *Unite* event¹¹ and the *Unity Developer Day*¹². Those heavy marketing strategy made it, currently, the most used game engine in the world and with the largest number of players for their games. The games made with *unity* had 5 billion downloads together in the 3rd quarter of 2016; 2.4 billion different mobile devices are used to play those games; 34% from the 1000 biggest free mobile games are made with *Unity*; and there are 770 million different people playing *Unity* games. Roughly 90% of *Samsung VR* and 53% of *Oculus Rift* games were made with *Unity*.

Another important feature that makes it so attractive is the direct integration of many secondary tools, essential for game development in the engine. For instance, the *Unity Analytics* allow developers to easily obtain gameplay and behavior data from player, without the need of third-party solutions. There is also the *Anima2D* tool which allows the in-engine creation of 2D animation and the level design tool *ProBuilder*. The current programming language in *Unity* is C#. Its business model is based in selling licenses based on the user's revenue: the free license gives all the core features of the engine and allows companies with less than U\$ 100,000 year revenue publish their games for free. For yearly revenues ranging from 100,000 to 200,000 dollars, it is necessary to buy the *Unity Plus* license, which costs U\$35 monthly and offers some advantages as additional features for the *Analytics*. The *Unity Pro* license is a need for companies with revenue greater than \$200,000 a year and it gives support from *Unity's* employees. Its actual

---

¹⁰ <<https://unity3d.com/pt/community/user-groups>>

¹¹ <<https://unite.unity.com/>>

¹² <<https://unity3d.com/pt/events/unity-developer-day-brasil-2017>>

pricing is a monthly fee of U\$125 ¹³. Another advantage of *Unity* is its learning process that is considered easier when compared to other market leaders, according to developers and game journalists (RASHID, 2016; DEALS, 2016; PLURALSIGHT, 2015).

The *CryEngine* engine is another powerful tool, granting the access to many tools with a heavy focus in graphics rendering and 3D games. This is done specially through the use of its real-time global lightning systems and physics-based rendering ¹⁴. *CryEngine* allows the use of both C# and C++ programming languages, common in the game development industry. Another advantage the engine offers is the royalty free policy, making its business model a "pay what you want" one. It also offers additional support for a monthly fee: U\$50,00 the basic and U\$150,00 the premium. Both include technical support and training programs (CRYTEK, 2018). However, a negative point is the lack of different platforms to release the games: only four platforms ¹⁵. In comparison, *Unreal Engine* build for 15 different platforms ¹⁶ and *Unity*, 28¹⁷, making them a better choice for developers who want to reach the largest consumer market possible.

Therefore, we chose *Unity* as the engine to develop our game prototype based on its faster learning rate, possibility to publish in many platforms, being the most used engine in the industry, C# language (preferred by the author), the many integrated tools, which make development easier in small teams, and the possibility to collect gameplay data from users, if needed for experiments with human players using *Unity Analytics*.

---

¹³ <<https://store.unity.com/>>

¹⁴ <<https://www.cryengine.com/features>>

¹⁵ <<https://www.cryengine.com/features/platforms>>

¹⁶ <<https://www.unrealengine.com/en-US/what-is-unreal-engine-4>>

¹⁷ <<https://unity3d.com/pt/unity/features/multiplatform>>

LITERATURE REVIEW

In this chapter, we review some works applying PCG to generate play-space and missions. The authors in (SUMMERVILLE; MATEAS, 2015) and (SUMMERVILLE *et al.*, 2015) applied PCG to action-adventure games as proposed by our work. First, they use Machine Learning (ML) techniques to learn distribution metrics in the construction of levels, and use the trained model to evaluate if new levels are similar in quality to the ones created by savvy designers. The learning algorithm is a Bayesian Network (BN) trained with 34 levels, translated from three games of *The Legend of Zelda* franchise. The network is trained with this data to create a graph representation of the level, containing the connections between rooms. Four different levels from these three games were taken apart to be used as a test set. Different topologies for the BN were tested, all hand-made by the researchers. The network was able to validate, through high-level constraints given by a designer, the topology of levels which respect conditions similar to the ones that were found in training (SUMMERVILLE *et al.*, 2015).

The next step of the work in (SUMMERVILLE *et al.*, 2015) aims to create an algorithm that allowed the generation of levels using the validated topologies and the creation of rooms for that level, and it was reported in (SUMMERVILLE; MATEAS, 2015). The generation of levels was also done with ML, however, using Principal Component Analysis (PCA) to reduce the dimensionality of the features. To feed the PCA, the authors used 488 rooms from the same games in the previous research. The generated rooms were interpolated with levels generated by humans in order to contain all the desired properties. Many constraints related to the game dynamics needed to be checked and fixed separately. The constraints were: existence of a final room; a boss room (final enemy of the dungeon); a room with a special item; access to all rooms; number of locked doors match the number of keys; all keys had to be used in order to complete the level, and all keys had to be found before the related locked room. Algorithms for each situation were defined by the authors (SUMMERVILLE; MATEAS, 2015) to validate these constraints. Based on such problems, in our research, one goal is to avoid the use of fix algorithms which drastically alter the level's or the missions' structure generated.

The work of Snodgrass and Ontañón uses Multidimensional Markov chains to create maps from two classic games *Super Mario Bros.* and *Kid Icarus*. The authors create constraints for the content generated by the Markov chains since the approach does not guarantee feasible results. These chains are matrices with size $M \times N$, representing the height and width of the level, respectively. Each element from the matrix is a type of tile present in the represented game's level. The results show that, even after applying constraints, some levels are still unfeasible, but they appear less frequently than only applying the Markov chain (SNODGRASS; ONTAÑÓN, 2016).

Another approach using statistical learning is present in Xu et al., where Monte Carlo method creates enemies with random movement and placement for a stealth game. The method is combined with an analysis and planning framework for stealth paths, enabling the researchers to guarantee the feasibility of the level and measure its difficulty. The method also requires the use of constraints to ensure feasibility (XU; TREMBLAY; VERBRUGGE, 2014).

In Ibirika et. al.'s work, PCG is employed to create caves, both in 2D and 3D, through the use of Voronoi diagrams and Delaunay triangulation. The caves have some special gameplay interest placed by a designer like treasure chambers, encounters with enemies, and others (SANTAMARÍA-IBIRIKA *et al.*, 2014). Johnson et al. also create caves, but applying CA to create infinite cave-like dungeons. By using the automaton, it is possible to provide a new cavern for the player during loading screen time. It becomes not necessary to save large amounts of maps in memory in order to give the players the feeling of an infinite dungeon (JOHNSON; YANNAKAKIS; TOGELIUS, 2010). Ashlock uses an EA to evolve fashion-based automata (which leave homogeneous region homogeneous) with different rules. The EA benefits half-unobstructed cell spaces and, together with the CA rules, create varied cavern-like level maps with large open areas for players to explore while creating interesting visuals (ASHLOCK, 2015). Adams et al. use an hybrid approach of CA and genetic algorithm where the automata creates the labyrinthic levels of the game, while the genetic algorithm evolve the automata's rules. The players were asked to rate dungeons created during 10 generations, to play the best level and to evaluate its playability (ADAMS; PAREKH; LOUIS, 2017).

An interesting generation of locked-doors missions able to change the play-space is found in Ashmore's and Nitsche's work. In their approach, there is a one-to-one mapping for locks and keys, where each key can open only one predetermined lock. It is the same approach proposed in our research, as it will be later described. Nevertheless, the rules that generate them and the play-space are hand-made by a designer, thus making every new combination of challenges and metrics a new task for a human, which goes against the goal in our present research (CALVIN; MICHAEL, 2007). A work with a different approach for mission generation is the one from Horswill and Foged, in which they use constraint propagation to populate a level with enemies and other objects. They distribute predetermined items and enemies through a level following some constraints, like placing keys only in reachable rooms and items in useful

locations. The distribution changes for each difficulty level chosen, placing more enemies and less items for harder difficulties. The algorithm was able to successfully distribute the assets in real time for many applications (HORSWILL; FOGED, 2012).

The educational puzzle game *Refraction* asks the players to arrange devices in a grid, aiming to build nets with lasers and learn different mathematical concepts. Smith et al. use this game as a test framework for PCG algorithms. Three different level generation tools are presented, all comparing their results using Answer Set Programming (ASP) to a second and alternative implementation. The first one generates missions, while the alternative implementation uses techniques from compilers. The second inserts the previously generated missions in a grid, and its alternative implementation uses depth-first search. The last one solves the created puzzles after their insertion in the grid and its alternative algorithm uses DFS. The results support the use of ASP with the appropriate constraints for generating levels for desired features, where competitive performance are reported against other methods (SMITH *et al.*, 2012).

About the specific application of EA for level generation, we find many works in the literature and present the most related ones to our research. Dahlskog and Togelius use EAs to evolve levels from a platform game to replicate a human designer's style. Their algorithm analyzes the levels in three different ways: micro-, meso- and macro-patterns. Micro-patterns are thin slices of levels used as pieces of an EA-based PCG algorithm, which searches for macro-patterns. The macro-patterns are combinations of meso-patterns, larger designed structures from a level. Computational and statistical results show the algorithm can replicate the designer's style. However, no user tests were made which would help to reinforce their findings (DAHLKOG; TOGELIUS, 2014).

Next, Liapis et al. show a tool to aid designers when creating levels using EA. The designer needs to draw a small-scale sketch of the map, identifying different resources and their locations. The EA offers suggestions of possible maps, following the restrictions imposed by the sketch. The designer proceeds to choose the best suggestion. The tool can work for many game genres and a real-time strategy game is used as example. Experiments with professional designers returned an overall positive evaluation. However, the authors report that the approach may be better used as a tool to assist game design learning, instead of an industrial tool (LIAPIS; YANNAKAKIS; TOGELIUS, 2013). McGuinness and Ashlock propose an EA based system to create tiles and plans in order to decompose the problem of creating a level for big games. An evolutionary computing algorithm is used to create a library of tiles and assembly plans, which are then used to put together many smaller maps in a bigger feasible map. The algorithm was able to correctly generate feasible levels (MCGUINNESS; ASHLOCK, 2011).

A genetic algorithm was developed from a framework that could generate content for different genres of games in the work of Khalifa et al. The algorithm uses two different populations: the first tries to improve the chromosome with feasible rules' fitness, while the second reduces its value when an unfeasible rule is found. Three classical versions of games

were used for tests: an adaptation of *Frogger*, another from *Pacman* and the last one from *The Legend of Zelda*. For each one, the levels created by the algorithm were compared against those generated from a random algorithm and another algorithm using the General Video Game AI Framework. All created levels were tested with users and the results indicated a clear preference for levels generated with the genetic algorithm over the other approaches (KHALIFA *et al.*, 2016).

EAs were also used to create levels for a game genre with more complex rules: physics-based puzzle games. In the works of Ferreira and Toledo, the EA was used to create levels for a clone of the Angry Birds game. This genre needs special care, since a set of physical constraints must be satisfied when aiming for level stability. Also, the simulation of the gameplay is necessary not only to check the physical constraints, but also to assure if it is possible for a player to finish the level with the initial number of birds. This drastically increases the EA's execution time, compared to applications in other games.

The authors made experiments with human players, who were asked to play the game both with human-generated levels, cloned from the original game, and EA-generated ones. The users were not able to differentiate between both sets of levels, showing another successful case of EA applied to PCG (FERREIRA, 2015; FERREIRA; TOLEDO, 2017). A follow-up research made by the present author applied ML techniques in order to reduce the EA's simulation time. The ML model learned if a level is feasible or not by analyzing the level's structure, number of birds and enemies. If a level was evaluated as unfeasible by the ML model, the simulation was skipped and the level's fitness penalized. Computational results confirmed the EA's convergence time was reduced, while the quality of solutions was kept (PEREIRA; TOLEDO, 2017).

Valtchanov's and Brown's research was one of the main inspirations for this project. They used genetic programming to evolve tree structures modeling a dungeon. The tree is translated into a 2D grid, discarding unfeasible nodes during the process. The work focuses in the creation of big dungeons with different size and shapes of rooms. Nonetheless, there is no major concern about the dungeon's missions, as the algorithm only selects some sparse rooms and label them as special. Then, mission events can be added to these rooms (VALTCHANOV; BROWN, 2012). Our research extends this one in the way we model the level and the problem. We also focus on adding mission elements to the dungeon. Locked-door missions are added directly in the tree structure, and the genetic programming operators are adapted to evolve both space and missions elements. The details will be presented in Chapter 4.

Other researches procedurally built together mission and play-space, but most of them use grammar based techniques for doing so. For example, Dormans, in a solo work, and later together with Bakkes, wrote about two projects where distinct sets of grammar rules and independent processes are used to create the play-space and locked-door missions for an action-adventure game level. These contents are reunited at the end of the generation. The results are visually interesting, but no tests with users are reported and no computational metric was used to measure

diversity and/or quality of the solutions (DORMANS, 2010; DORMANS; BAKKES, 2011). Dormans later proceeds with mission-generating grammars by creating diverse combinations of key-lock pairs and adding more complexity to the level. Again, no human or computational tests were made, becoming not possible to assess the quality of the levels generated in (DORMANS, 2011).

Another work using grammars was done by Linden et al., where the authors create a grammar with a linear set of diverse rules containing enemies and items, besides locks and keys. The many grammars created are evaluated through complexity histograms (number of existing subgraphs in the final level layout) and danger levels (total amount of damage a level can do to a player) of a graph. Results show the grammars can create levels that are different among themselves, but similar when created by the same grammar (LINDEN; LOPES; BIDARRA, 2013). Merrick et al., on the other hand, builds a grammar focused in the generation of levels together with the placement of Non-Player Character (NPC). Computational experiments show the generated levels hold similar values for interest and novelty from all tested grammar-related parameters (MERRICK *et al.*, 2013). Both previous researchers did not experiment with human players.

Ashlock and McGuinness created both play-space and mission for RPGs by first evolving an user-input map file with an EA that uses the proximity of placement of key-elements as its fitness. Then, rooms are identified using a flood-fill algorithm and are represented in an adjacency graph. Finally, each room will receive a randomly selected description that specify monsters, traps, treasures and notes to the referee of the game appropriate to each room's conditions. All these descriptions must also be input by a human designer. Although heavily dependant on human input, the research provides interesting results and can be used as an aiding tool for designers (ASHLOCK; MCGUINNESS, 2014).

Liapis recently introduced an EA approach to create play-space and missions simultaneously. First, the algorithm creates a high-level overview of the map used to specify constraints and goals in multiple constrained optimization algorithms. These algorithms create high resolution segments of the map. The initial draft is evolved using a fitness function with parameters given by humans as in our approach. Each tile created in the overview, together with its connections, guides the evolution in each different segment of the high-resolution map. After all pieces have been evolved, they are combined to create a whole dungeon. This dungeon contains treasures and enemies and, according to the presented results, can adapt successfully to the designer's input. However, one of the main hurdles pointed by the author himself is the need of many constraints in the evolutionary process to ensure a feasible solution (LIAPIS, 2017). Our approach aims to create mission and play-space together with few human inputs, but trying to overcome the need of a large amount of constraints.

The main inspiration for the room generation in our work are the algorithms presented by the Galak-Z game's developers. The advantages and disadvantages of the many procedural

dungeon generation approaches like CA, Hilbert curves and Z-order curves are explained by them. The Hilbert-curve-based one translated the curve into a grid, chose a random point in which to start creating a path following the curve, and built the room using the resulting path between starting and ending points (UNITY3D, 2014). Although they used the curves to create the path between rooms, we use them to create the paths inside the rooms in our project.

SFCs have been successfully employed in many computational applications, e.g., to represent crystals in ML models, for mesh refinement and partitioning, as well as compression of color data (JASRASARIA *et al.*, 2016; BURSTEDDE; HOLKE, 2016; BORRELL *et al.*, 2018; DOLONIUS *et al.*, 2018). Thus, it is a reasonable assumption that the PCG field may also benefit from SFCs, and this research intends to be a starting point to investigate such hypothesis.

The most related work to this subject is presented by (MACEDO; CHAIMOWICZ, 2017). Similar to the algorithm used in Galak-Z and the one used in this research, those authors use sections from Hilbert curves, randomly selected, to create the map of the game. However, they apply those sections as initial configuration of the CA, setting every cell in which the curve is present as *false*. Next, the authors run an algorithm to guarantee a minimum width for the curve, expanding it when too narrow, so the player can traverse it. Also, the path created by cells, which does not belong to the curve, is handled in such a way that no branching from the main path is created. Therefore, the starting configuration and the evolution of the automaton are impacted by the SFC, that guides it to a map with a feasible path with an organic aspect. Apart from scientific applications, automata are also used in the game industry, besides from the Galak-Z game. In SimCity (WRIGHT, 1989), a city management simulation game, CA are used to model the patterns of changes during the development of cities (MILLINGTON; FUNGE, 2009).

As seen throughout this chapter, there are a plethora of procedural level generation applications in the scientific literature. Most of them aim only to create the play-space, while others focus on the generation of missions, and a third group, in generating both. This last group contains mostly grammar-based techniques, although the work from Liapis successfully uses an EA for this task (LIAPIS, 2017). Therefore, the successful cases of EA in PCG encouraged us to use it in our novel approach for dungeon generation, presented in this document.

METHODOLOGY

In this chapter, the methods used to generate dungeons and rooms are detailed from sections 4.1 to 4.2. A brief description of the game prototype is also provided by Section 4.3.

4.1 Dungeon Map Generation

This section describes the algorithms used to create the dungeon and its respective locked-door missions. All parameters introduced are followed by their values during the computational tests in Chapter 5. These values were empirically set after some initial tests. The graduate student from the PPG-CCMC program, Paulo Victor de Souza Prado, helped in the development of the dungeon generation algorithm and the first version of the EA that evolved its structure.

4.1.1 Tree Structure

A tree representation was chosen as a genotype encoding, illustrated by Figure 11. Each node is a room from one of three types: Normal Room (NR), Key Room (KR) and Locked Room (LR). A KR has a key inside, represented by the bold nodes, and LR has the edge between itself and its parent locked, represented by the dashed edges. Thus, we encode at the same time the mission and the play-space. The creation of the tree structure can be seen in Algorithm 5. The individuals randomly generated by this algorithm are evolved following the procedure described in Section 4.1.4. The tree will follow some parameter values obtained as the best results from previous experiments, like the maximum depth of the tree ($MAX\_DEPTH = 9$), maximum number of children for each node ($MAX\_CHILD = 3$), probability of the room having a key ($ROOM\_KEY = 15\%$), probability of the room being locked ($ROOM\_LOCK = 15\%$) and the probability of it being a Normal Room ($ROOM\_NORMAL = 70\%$).

The parameter $MAX\_CHILD = 3$ will allow a maximum of 4 connections with other rooms. The chance to decide the number of children for each node, $PROB\_HAS\_CHILD$, begins

Figure 13 – The resulting dungeon (Phenotype) from translating the genotype in Figure 11. Keys are represented by yellow squares, locks by red rectangles, each with their respective ids. Rooms are presented in white squares, and doors as green rectangles. The starting room is marked with a blue ‘S’ and the final room a black ‘B’.

added to a queue of rooms, which will be used in a breadth-first algorithm. The algorithm is applied to create the children nodes until no more rooms are in the queue, or the maximum depth of the tree is reached. Each new room is inserted as a child of the *currentRoom* and added to the queue of rooms to be visited. Also, the number of children is randomly drawn with the respective direction of each one. The tree structure was chosen over the normal graph/grid approaches, since it reduces the amount of validation and fix algorithms needed to guarantee the semantics of the level. In this case, the breadth-first approach is used to distribute special rooms assuring feasibility by itself. It is often necessary to execute route-finding algorithms in other approaches, making them computationally expensive when assuring the level feasibility.

4.1.2 Keys and Locks

The tree's nodes carry semantic information about the kind of room. It may contain a key that unlock doors, and these locked doors are represented as the edge connecting a room and a LR. We get rid of the need to store information about edges by always locking the edge between a LR and its parent. In this case, it is necessary to store only the rooms with their corresponding types, children and coordinates. An identifier (id) is given for each pair of key-lock combination since the pair can be unique. Moreover, designers may add a different aspect for each id in their final dungeon, or turn all of them into keys that open every door, discarding the id information,

Algorithm 5 – Generates Random Tree Structures**Ensure:** A new dungeon with several rooms

---

```

1: while toVisit.Count > 0 do
2:   actualRoom  $\leftarrow$  toVisit.Dequeue(ROOM_TYPE)
3:   // Get the depth of the current room in the tree
4:   actualDepth  $\leftarrow$  actualRoom.Depth
5:   if actualDepth > MAX_DEPTH then
6:     // The dungeon has already reached max depth, clean the list of rooms to be visited
7:     toVisit.Clear()
8:     break
9:   else
10:    prob  $\leftarrow$  Draw number to check if room will have children
11:    PROB_CHILD  $\leftarrow$  PROB_HAS_CHILD * (1 - actualDepth  $\div$  10)
12:    if prob  $\leq$  PROB_CHILD then
13:      // Draw number of actualRoom's children
14:      numberOfChildren  $\leftarrow$  draw(MAX_CHILD)
15:      // If the number of children drawn equals 3, the direction array will have all the
      allowed directions
16:      directionArray  $\leftarrow$  Draw each child's direction
17:      for i = 1  $\rightarrow$  numberOfChildren do
18:        prob  $\leftarrow$  Draw number to check room's type
19:        if prob < PROB_KEY then
20:          ROOM_TYPE  $\leftarrow$  KEY
21:        else if prob  $\geq$  PROB_NORMAL then
22:          ROOM_TYPE  $\leftarrow$  NORMAL
23:        else
24:          ROOM_TYPE  $\leftarrow$  LOCK
25:        end if
26:        child  $\leftarrow$  createRoom(ROOM_TYPE)
27:        // Insert the new room as a son of the currentRoom, calculating the [X,Y]
        position
28:        currentRoom.InsertChild(dirArray[i], child)
29:        child.Direction  $\leftarrow$  directionArray[i]
30:        // Add the new child in toVisit, so it'll pass through the same process to
        receive children
31:        toVisit.Enqueue(child)
32:      end for
33:    end if
34:  end if
35: end while

```

---

as did in our game prototype described in Section 4.3.

In Figure 11, we illustrate the dungeon's semantic for keys and locks. The KR has the key id, and the LR contains the key id necessary to unlock that path. A thick dashed line marks the destination room, chosen as the last LR to be created. It is guaranteed that this room will be the farthest LR from the initial node in terms of depth, as a result from the breadth-first approach

discussed earlier. This approach doesn't consider the real player's travel distance to reach the destination. If another LR, closer to the starting room, has many locked doors before it with their keys lying in different segments of the dungeon, the player's path to this room might be longer.

4.1.3 The Spatial Placement: Grid

The tree structure described in Section 4.1.1 can be translated into a grid with some adequate processing. This involves each node's coordinates calculation and the removal of overlapping nodes along with their children. To calculate the rooms' positions, the algorithm assumes that the root occupies grid's coordinates (0,0) as shown in Figure 15. Each child will occupy the corresponding space in the grid analogous to their place in the tree, as stated in Section 4.1.1 and shown in Figure 12. To make calculations easier, the relative orientation of every child node is changed according to Figure 12. In this case, the coordinates' calculation is done based on its relative orientation, i.e., the Right Child of a node at $[x,y]$ will not always be at this node's East relative to the grid's orientation, but may be at its South. Therefore, a rotation is applied before calculating its grid's position, which would usually be $[x+1,y]$. After the rotation of 270° , in this example, the new coordinates are correctly found calculating $[x,y+1]$. Figures 16 to 18 show some examples of the process applied during the decoding of a level's tree into a grid.

Each new child node is tested for overlapping during the creation of the grid. If overlap occurs, the new node is discarded. Thus, every genotype, like the one in Figure 11, generates a valid grid representation as in Figure 14. The grid structure was chosen since it is a common representation of dungeons in many action-adventures and role-playing games. Figure 19 shows another conversion from genotype to phenotype. Two possible representations of the same dungeon are illustrated in Figure 14: the one on the left doesn't consider the unique id of the lock-key combinations, and the other one matches each pair with the same colors.

Figure 14 – Left: The Grid Representation (Phenotype) of the dungeon in Figure 11. Right: The same dungeon, but with each key opening only its corresponding lock (color-matching)

Figure 18 – Fourth step of genotype-phenotype translation

Figure 19 – Fifth and last step of genotype-phenotype translation

4.1.4 Evolutionary Algorithm

This section introduces the EA that evolves the randomly generated levels created by Algorithm 5. We present its parameters and the selection, crossover and mutation operators. The EA is described by Algorithm 6. A closer look at the mutation and crossover operators will be provided through algorithms 8 and 7.

4.1.4.1 Crossover

Each pair of individuals selected for crossover has a Crossover Rate (CR) of 90% to be mixed using the one-point technique, meaning that two randomly selected sub-trees from each parent will be exchanged. This swap is able to redo the entire layout by moving a set of rooms from one dungeon to another. However, the crossover can lead to semantic errors, e.g., leaving a lock without a necessary key. Therefore, a fix method repairs the semantic of locks and keys by

Algorithm 6 – EA**Require:** A population *pop*, which is the list of dungeons.**Ensure:** Final population after *nGenerations*.

```

1: for  $i = 0 \rightarrow nGenerations$  do
2:   newPop // Create a new empty population
3:   // Fill the new population with new individuals
4:   for  $j = 0 \rightarrow Length(pop) \div 2$  do
5:     parent1, parent2  $\leftarrow Tournament(pop)$ 
6:     child1, child2  $\leftarrow Crossover(parent1, parent2)$ 
7:     Mutate(child1)
8:     Mutate(child2)
9:     Fix(child1)
10:    Fix(child2)
11:    newPop.Add(child1)
12:    newPop.Add(child2)
13:   end for
14:   bestIndividual  $\leftarrow Elitism(pop)$ 
15:   newPop.first  $\leftarrow bestIndividual$ 
16:   pop  $\leftarrow newPop$ 
17: end for

```

Algorithm 7 – Crossover**Require:** Two individuals *ind1* and *ind2* to be mixed and *cr*.**Ensure:** If the crossover happens, returns two new individuals.

```

1: cc  $\leftarrow$  draw a crossover chance
2: if  $cc \leq cr$  then
3:   calcDesiredKeys(ind1, ind2)
4:   roomCut1  $\leftarrow$  draw a random room in dungeon ind1
5:   roomCut2  $\leftarrow$  draw a random room in dungeon ind2
6:   //Substitute the cut branches between dungeons
7:   ind1.Swap(roomCut1, roomCut2)
8:   ind2.Swap(roomCut2, roomCut1)
9: end if
10: return ind1, ind2

```

---

first turning all rooms into NR, and recreating the special rooms (KR and LR pairs). The number of desired special rooms in the children is represented by the variable *DesiredKeys*, and it is the average of each parent's number of keys (Algorithm 7, Line 3). This repair is done only after mutation has been applied.

After the exchange of nodes, the grid structure must be rebuilt to validate the position of new rooms. The old coordinates of new rooms in the exchanged branch are replaced, following the same procedure explained in subsection 4.1.3. If a newly acquired node overlaps an existing one, it is removed from the tree along with its children. This step is executed during the Swap method in lines 7-8 of the Algorithm 7, which describes the crossover operation.

Algorithm 8 – Mutation

Require: Individual ind , Grid $grid$, mutation rate mr , and two types of mutation (mt_1 and mt_2).

Ensure: If mutation happens, a new, mutated, individual. Otherwise, the same as the input's individual.

```

1:  $prob \leftarrow$  draw probability to change DesiredKeys
2: if  $prob \leq mr$  then
3:    $dungeon.DesiredKeys \leftarrow$  new desired keys' value
4: end if
5: for Room  $r$  in  $ind$  do
6:    $mc \leftarrow$  draw a mutation chance
7:   if  $mc \leq mr$  then
8:      $mt \leftarrow$  draw a mutation type
9:     if  $mt = mt_1$  then
10:      if  $r$  has available room child space then
11:         $childPosition \leftarrow$  draw a feasible child position to receive the new room
12:         $newChild \leftarrow$  Create the new child room
13:         $newChild.parentDir \leftarrow$  parent direction relative to child
14:         $ind.Add(newChild)$  // Add child in the tree structure
15:         $grid.Add(newChild)$  // Add child in the dungeon's grid
16:      end if
17:    else if  $mt = mt_2$  then
18:      if  $r$  is a leaf node then
19:        remove the node from  $ind$ 
20:        remove the node from  $grid$ 
21:      end if
22:    end if
23:  end if
24: end for

```

4.1.4.2 Mutation

The mutation algorithm has 2 steps, both following a Mutation Rate (MR) $mr = 5\%$. In the first step, the mutation of the dungeon features can take place by replacing the *DesiredKeys*'s value by a random number between 1 and 9 (both inclusive). In the second step, which alters the play-space, all dungeon's rooms may change when $mc \leq mr$ in line 7, being mc a randomly drawn number representing the mutation chance. Two mutation types mt_1 and mt_2 have 50% of chance to be applied. A new child is added to the room in mt_1 , only if the room has less than *MAX_CHILD* children and the new one would not overlap an existing room. A room is removed in mt_2 , but only if it is a leaf node. Algorithm 8 describes this operator.

4.1.4.3 Fitness

The fitness function evaluates four features of the dungeon: total number of rooms d_r , keys d_k , locks d_l and the Linearity Coefficient (LC) of the play-space d_{Lin} . The value of d_r determines the size of the dungeon, while d_k and d_l are related to the complexity of the locked doors puzzle. If there are more locked doors, the player must remember more locations, find

more keys and, eventually, backtrack more paths. Also, for the configuration where any key can open any door, the user may want to make the puzzle easier by creating more keys than locks.

Thus, our algorithm contains both values to satisfy these criteria. The last feature, linearity, is similar to the average degree of the nodes in a graph. This measure was adapted to be rapidly processed in the tree structure, as given by Equation 4.1, where $nInner$ is the number of the tree's inner nodes. Therefore, it leads to a linear dungeon for d_{lin} values closer to 1, and it becomes maze-like for greater values, with more rooms and multiple doors. The fitness value f is obtained by calculating the Manhattan distance between the desired input values for each feature and the dungeon's current values. The desired input values are u_r , u_k , u_l and u_{lin} in Equation 4.2, each one defined by the human designer creating the level.

$$d_{lin} = (d_r - 1) / nInner \quad (4.1)$$

$$f = |u_r - d_r| + |u_k - d_k| + |u_l - d_l| + |u_{lin} - d_{lin}| \quad (4.2)$$

4.1.4.4 Evolution and Fix Algorithm

The population of dungeons (individuals) will evolve by the EA until a maximum number of generations is reached. In each generation, the EA will select individuals, through the tournament selection from 2 individuals, who will create offsprings. Thus, for every couple of parents chosen by tournament, the crossover and mutation operations are executed to generate two children. After all operations are finished, the children go through a fix algorithm to correct the semantic errors by re-labeling all the rooms as NR, KR, or LR. This is done by using a slightly modified version of Algorithm 5. The new children are added to the new population until such population reaches the same size of the old one. After the new population is ready, an elitism selection is applied on the previous population, where only the best individual is passed on to the new population. This procedure assures that the best solution so far will not be lost.

The aforementioned fix algorithm will first turn all rooms into NR, deleting all special rooms, and then execute a breadth-first approach over the tree, similar to Algorithm 5. However, the difference is the chance to generate a room as NR. The probability to create a NR is decreased by 10% for each key missing in the dungeon compared to the number of desired keys. Thus, increasing the chance to create special rooms. If more keys than the desired ones are created, the probability to create a NR is increased by 10% for each extra key in comparison with the desired number, reducing the chance to create more keys than desired.

4.1.5 Constrained Evolutionary Algorithm

The results achieved by applying the EA described previously are presented in Subsection 5.1.1 and Section 5.2. They are also in the paper recently published in the **2018 IEEE Congress on Evolutionary Computation** (IEEE CEC 2018) (PEREIRA; PRADO; TOLEDO, 2018). However, evaluating the results obtained so far, one of the conclusions taken was that the stochastic nature of the crossover algorithm could affect negatively the evolution process. The previous crossover removes all semantic information and recreates it later through the fix algorithm.

Therefore, we developed a new approach for the genetic operators that preserves the mission's structure, bringing more stability to the dungeons' design. Another aspect included in the EA was a path-finding algorithm. This method finds a way from the initial room to the last one, while obtaining the necessary keys to open the corresponding locks. Thus, the ratio of rooms visited by the path-finding algorithm could now be added to the fitness function. We also included in the fitness the amount of locks needed to be opened when defining such path.

These operators are described in Subsection 4.1.5.1. We will refer to this new EA as Constrained Evolutionary Algorithm (CEA). The path-finding algorithms used in the research are presented in Subsection 4.1.5.2 together with the new fitness function. The computational results of the CEA using the new techniques are present in Subsection 5.1.2.

4.1.5.1 Constrained Crossover and Mutation

To better preserve the semantic information of dungeons after the generation of the children solutions, the crossover algorithm was altered to keep the same number of special rooms of the removed branch when exchanging it between parents. Therefore, the new crossover is also done by selecting two random cut points, one from each individual, and exchanging the branches. The main modification is that the breadth-first sequence of the special rooms placement is preserved after the trade: the order is rebuilt applying a similar version of the Algorithm 5 shown in Subsection 4.1.4.4. Instead of repairing the whole individual, the fix algorithm is executed only in the individual's newly-acquired branch. In this version of Algorithm 5, the chance to generate room types NR, KR or LR is the same as described in Subsection 4.1.1, but both KR and LR are treated as a single special room. This special room has a probability to be created equals to the sum of the probability of KR and LR, and the type of room the special room will become is defined by the order they existed in the previous branch. In order to do so, the visit order following breadth-first procedure is also applied.

If all the special rooms saved from the previous branch have already been placed in the new one, every new room will now be a NR. On the other hand, no special room can be discarded in the process. For example, if we assume a total of n special rooms in the removed branch and number of not yet visited rooms by the breadth-first algorithm during the fix execution reaches n ,

all these n rooms in the branch are labeled as special rooms following the same order they were taken from their original branch.

Algorithm 9 describes the procedure. The individuals are first copied so if any constraint has not been met during the crossover, the copies are discarded and the process starts again. If all special rooms can be saved from one branch to the other after all steps are computed, the copies are then saved in the place of the original ones. The repair algorithm is executed, and the grid is updated with the newly acquired rooms and by discarding the ones from the old branch.

In case the crossover cannot happen, both nodes are placed in a list and are not used as cut points again. If there are no more nodes available (that is, nodes in the tree out of the list), the crossover does not occur. In this scenario, the children are the exact copies of their parents. We select this approach instead of inspecting every combination between each node from one individual against all others to reduce computational time. Considering two parents with x and y nodes respectively, with $x \geq y$, our approach needs to verify, in the worst scenario, y combinations of cut points for the crossover. While if we would consider every possible combination of nodes, we would need to verify $x * y$ crossover attempts. For two dungeons with 20 rooms, for example, this reduces the attempts from 400 to 20. A clear downside of this approach is that some children will not be altered by the crossover. However, they are still eligible to take part in the mutation operation and, even if not mutated, the selection pressure will reduce their chances of giving birth to offspring in the next generation if they are bad solutions. Thus, we decided the trade-off was worth as it reduces execution time and we plan to upgrade this algorithm for real-time applications in the future.

The play-space mutation happens in a similar way as described by Subsection 4.1.4.2. The only difference is the removal mutation, where the selected node will only be removed if it is a NR. Otherwise, the mutation does not occur and the node is kept in the tree. The mutation in the number of keys is done in a different approach since the number of desired keys does not exist anymore. It is now divided into two types: addition or removal of a key-lock pair. The two types have equal chance to occur, if the mutation condition is met.

To add a pair, the mutation algorithm visits the tree in a breadth-first fashion, giving each visited NR-node a chance to be converted into a KR. This probability is the same as the tree-creation algorithm. After the KR has been created and given an id, the next special room is a LR, with the same id. If the pair is created, the mutation stops immediately. Otherwise, it proceeds until the tree is fully visited. If no LR is created, then just a KR is added, creating a surplus of keys. The removal of a pair happens as follows: a random key-lock pair is selected by their id to be removed from the tree and both KR and LR become NR. If the selected id belongs to a surplus key, only the KR is converted to a NR. Algorithm 10 describes the whole mutation process.

When the crossover and mutation operations are completed, the CEA will follow the same procedure as described in Algorithm 6, except that lines 9 - 10 will not be executed. Instead,

Algorithm 9 – Crossover with Constraints

Require: Two individuals *ind1* and *ind2* to be crossed over, *cr*, two lists of rooms to save the special rooms that will be exchanged between branches: *spc1* and *spc2*.

Ensure: If crossover occurs, return two new individuals.

```

1: cc  $\leftarrow$  Draw crossover chance
2: if cc  $\leq$  cr then
3:   CanTrade  $\leftarrow$  false
4:   while CanTrade = false do
5:     roomCut1  $\leftarrow$  Draw random room from dungeon ind1
6:     roomCut2  $\leftarrow$  Draw random room from dungeon ind2
7:     saveSpecialRooms(outspc1, outspc2)
8:     clone1  $\leftarrow$  clone(ind1)
9:     clone2  $\leftarrow$  clone(ind2)
10:    clone1.Swap(roomCut1, roomCut2)
11:    clone2.Swap(roomCut2, roomCut1)
12:    RemakeGrid(clone1)
13:    RemakeGrid(clone2)
14:    if RoomsInBranch(clone1, roomCut2)  $\geq$  spc1 and
15:      RoomsInBranch(clone2, roomCut1)  $\geq$  spc2 then
16:      CanTrade  $\leftarrow$  true
17:    end if
18:  end while
19:  //Exchange the branches cut from dungeons
20:  ind1 = clone1
21:  ind2 = clone2
22:  ind1.FixBranch(roomCut2)
23:  ind2.FixBranch(roomCut1)
24: end if
25: return ind1, ind2

```

the path-finding algorithms are applied to compute a route from the initial to the final room. We used two different algorithms as described in Subsection 4.1.5.2.

4.1.5.2 Path-Finding

An adapted version of the A* and the Depth-First Search algorithms were implemented, to simulate the behavior of a player traversing the dungeon created by the CEA. Both versions account for the existence of locks and keys in the path. Thus, all locked doors are considered unreachable nodes until the key that opens each of them is found by the algorithm, during its visits to the nodes.

The A* was chosen because it is one of the most computationally efficient path-finding algorithms and it is widely used in the game industry. The A* behaves similarly to a player who knows where the goal of the dungeon is, both going through a comparable number of locked doors. Thus we will use this information for our fitness, as it simulates players' behavior. Meanwhile, the DFS algorithm explores a path until a dead-end is reached. This is a very similar

behavior to most players who have no previous information about the dungeon, and are trying to map it. So we also use this information in the new fitness function.

Both algorithms must consider the rooms behind locked doors unreachable until the KR containing its key is visited. When a KR is visited by one of the path-finding approaches, the corresponding LR is immediately altered as "open" for the algorithm to visit. Its parent room, if already changed to "closed" (that is, already visited and its neighborhood explored), has its status changed to "open". This will allow the algorithm to search its neighborhood again and visit the once-locked room. As the DFS algorithm can be executed by choosing any order in nodes with multiple children, we needed to choose one. As the players can go to any children when playing, we do not to set a fixed visit order. The child will be randomly visited from each node with more than 1 child not yet explored.

The A* algorithm returns how many locks it opened while searching for the final room ($used_l$) and the DFS gives how many rooms were visited during the search ($used_r$). These statistics are then used as part of the new fitness function, where all rooms and locks ideally should be explored by the player. If not, then some of the paths of the level aren't part of the locked door mission, which is generally not desired in game design. To reduce the impact of the randomness of our DFS approach, we execute the algorithm 3 times for each dungeon, and use the average of the number of visited rooms for each execution ($used_r$). Thus, the new fitness function is given by Equation 4.3. We add the difference between the total number of locks (d_l) and the number of locks opened by the A* ($used_l$), named as Δ_l . It is also included the difference between the average of the visited rooms by the DFS ($used_r$) and the total number of rooms (d_r), named as Δ_r .

Another important change was the addition of a weight when measuring the proximity to the user input. The weight puts more pressure in creating levels similar to the input provided by the game designer, giving priority for this matter over the full exploration of levels. The weight was necessary since preliminary experiments have shown that some solutions converged to fully explored dungeons (both Δ_r and Δ_l equal or very close to 0), but with the other parameters not closer enough to the game designer inputs. Thus, a double weight multiplies the user-related parameters as shown in Equation 4.3.

$$f = 2 * (|u_r - d_r| + |u_k - d_k| + |u_l - d_l| + |u_{lin} - d_{lin}|) + \Delta_l + \Delta_r \quad (4.3)$$

4.2 Room Generation Techniques

This section describes how the SFCs and the CA were applied to create dungeon's rooms. We also explain the experimental procedures along with the metrics for evaluating the results achieved. The approach described in this section was developed together with the scientific

initiation project from the ICMC undergraduate student Giovanna Oliveira Guimarães.

4.2.1 Sectioning Space-Filling Curves

In order to generate rooms, the first step is to define a matrix representing the SFC. The cells in the matrix have value 1 or 0, where cells with 1 stand for block tiles, and 0 are walkable tiles. A tile representation, instead of a pixel by pixel one, was chosen since it reduces the amount of information to create the room. Figure 20 illustrates the discretization process with walkable cells in green and blocks in black. Our representation fixes the line's width to 1 block, but the distance between points can change. The distance between points will be referred as *width* from here on, as the width of the gaps between curves. In Figure 20, the order 2 curve has a *width* of 4 (left), and the *width* of order 3 curve is 1 (right). In that way, greater *width* values turn into levels where the player has larger corridors to walk into. They need to be large enough to allow more variety of sections, since we will extract later sections of this matrix to create rooms.

Figure 20 – Visual representation of the discretization process of two Hilbert curves of different orders (2, on the left, and 3, on the right). The green cells are walkable tiles, and black ones are blocks.

After the matrix has been built, it can be used to create as many rooms as needed by selecting sections with the desired size for rooms. We randomly select the coordinates of the section and proceeds as shown in Figure 21. The section will have width w and height h , both parameters defined by the user, and corresponding to the amount of tiles in the generated room.

If the selected coordinates would create a section smaller than $w \times h$, another location is selected. In the example, both have $w = 7$ and $h = 7$.

Next, information about the doors in the room are received and, to simplify the selection, four choices become possible: North (N), South (S), East (E) and West (W). All doors are positioned as illustrated in Figure 21. They are put near the middle tile closest to the wall of their corresponding direction. The user can select how many doors the level will have (1 to 4) and their positions. After the placement of doors, a simple fix algorithm is executed for each pair of doors, if there is no path between them. The fix algorithm starts from each door, measuring the distance between each reachable tile from that door to the reachable tiles from the other one. A path is drawn to connect both tiles, first closing the distance horizontally and then vertically. Any block tiles (value 1) in the path are turned into walkable ones (value 0). The approach does not try to connect them using a diagonal path in order to preserve the aesthetic characteristics of the SFCs. Figure 21 has an example of the full room creation process using SFCs. The images on the left use the order 2 curve from Figure 20, and the room has North and South doors. The images on the right use the order 3 curve and the room contains 3 doors. In this case, the fix algorithm is executed since East and West doors are unreachable from the other doors.

4.2.2 Adding the Game of Life

First, we create a matrix of cells, preferably larger than the room's size, randomly setting the alive or dead states. The Game of Life rule is applied to the cells during several generations. After that, any generation can be randomly chosen and its grid is sectioned. In order to merge this procedure with the SFC, we combine different sections of the automaton with multiple distinct SFC sections. A total of three logic operations were tested to execute such combination: AND, OR and XOR. However, only OR achieved satisfactory results. We also test the expressive range of the Automaton compared to SFC only and SFC aided by CA methods. In the last case, the CA applied both A12B1 and A23B3 rules.

4.3 Game Prototype

We built an adventure game prototype in the Unity game engine, with the help of a fellow graduate student from ICMC, Rafael Miranda Lopes. The game, named as ProCoG (an acronym for Procedural Content Generation) is able to read a text file containing information of the dungeon generated by the EA (sections 4.1, 4.1.4 and 4.1.5). It proceeds to read another file, filled with the content of each room, as generated by our SFC approach (Section 4.2). The rooms are then instantiated according to the specifications in the files. Keys are placed the closest to the center of KR as possible, always in an accessible place. LR has the doors between it and its parent room locked. For this game, the key-lock pairs are considered as generic, meaning that any key can open any lock.

Figure 21 – Creation of two 7x7 rooms using order 2 (left) and 3 (right) Hilbert curves. The section is extracted starting at point (4,2) in both corresponding Hilbert matrices. In the second to last row, the doors are placed in previously defined exits. Then, in the last row, paths to connect the doors in the right room are created.

After everything has been instantiated, the player's avatar is loaded in the starting room and the game begins. The player can move the avatar using the arrow keys or the *WASD* keys, and the only goal is to navigate the rooms and reach the item in the final room. Figure 22 shows a screenshot from the dungeon generated executing the game in Unity. The sprites for the game

were all created by the authors and carry a similar visual style to the first *The Legend of Zelda* game. The room highlighted in green is the starting room, and the one in red is the final room. Section 5.4 shows details of the game prototype, its execution and the experimental data obtained through its use.

Figure 22 – Screenshot from the prototype game in the Unity engine. The screen shows a dungeon generated using the outputs of both the dungeon and rooms PCG algorithms proposed in this research

Algorithm 10 – Mutation with constraints

Require: Individual *ind*, Grid *grid*, mutation rate *mr*, types of mutation (*mt*₁ and *mt*₂), special room rate *spr*.

Ensure: If mutation happens, a new individual. Otherwise, the same as the input's individual.

```

1: prob  $\leftarrow$  draw probability to mutate keys
2: if prob  $\leq$  mr then
3:   mt  $\leftarrow$  draw a mutation type
4:   if mt = mt1 then
5:     //Adds a key-lock pair
6:     for Room r in ind do
7:       spprob  $\leftarrow$  draw chance to change room
8:       if spr  $\leq$  spprob then
9:         if HasNotCreatedKey then
10:          r.type  $\leftarrow$  Key Room
11:          HasNotCreatedKey  $\leftarrow$  false
12:        else
13:          r.type  $\leftarrow$  Locked Room
14:          Break For
15:        end if
16:      end if
17:    end for
18:  else if mt = mt2 then
19:    //Removes a key-lock pair
20:    pairIndex  $\leftarrow$  draw which pair will be removed
21:    Find pair and remove both Key and Lock
22:  end if
23: end if
24: for Room r in ind do
25:   //Run through nodes to check if space mutation happens
26:   mc  $\leftarrow$  draw a mutation chance
27:   if mc  $\leq$  mr then
28:     mt  $\leftarrow$  draw a mutation type
29:     if mt = mt1 then
30:       if r has available room child space then
31:         childPosition  $\leftarrow$  draw a feasible child position to receive the new room
32:         newChild  $\leftarrow$  Create the new child room
33:         newChild.parentDir  $\leftarrow$  parent direction relative to child
34:         ind.Add(newChild) // Add child in the tree structure
35:         grid.Add(newChild) // Add child in the dungeon's grid
36:       end if
37:     else if mt = mt2 then
38:       if r is a leaf node AND is not a special room then
39:         ind.Remove(r)
40:         grid.Remove(r)
41:       end if
42:     end if
43:   end if
44: end for

```


RESULTS

This chapter presents the results from our research. We divide the results in four sections: Section 5.1 reports the EA results for both proposed versions (subsections 5.1.1 and 5.1.2). The achievements with human evaluations over the generated maps, running the first version of the EA, are presented in Section 5.2. Section 5.3 shows results obtained by measuring, with different metrics, the rooms generated by our SFC-based method. Finally, we present the game prototype created as a proof of concept for applications of the algorithm in the game industry. This prototype was also used to gather data from human players in our last experiment. Both the prototype images and the results achieved by the last experiment with players are present in Section 5.4.

5.1 Dungeon Generation Results

This section reports the results of the proposed EAs. We evaluated their accuracy in generating levels that match the input configurations given by a game designer. All tests were run with the same parameter values for the EA. These values were empirically defined based on previous tests: 90% of crossover rate, 5% of mutation rate (for both mutations), 50% chance to choose between the mutation which adds or removes a node. The population has 100 individuals and the stop criterion is 100 generations. The other parameters were the same as already described in sections 4.1 and 4.1.4.

5.1.1 EA Results

In the first experiment, a total of 6 different input configurations are tested and the EA is executed 100 times over each configuration. The mean and Standard Deviation (Std. Dev.) are obtained as shown in Tables 1 and 2. The *Features* columns show 6 different input configurations, which are the parameters: u_r , u_k , u_l , u_{lin} . For all sets of parameters, the average execution time was not greater than 2.7 seconds and it was omitted from the tables to simplify the visualization.

Table 1 – Number of rooms (d_r) and keys (d_k) obtained after 100 executions of the EA for each combination of features

Features (u_r, u_k, u_l, u_{lin})	d_r		d_k	
	Mean	Std. Dev.	Mean	Std. Dev.
(15, 3, 2, 2.0)	15.00	0.00	3.00	0.00
(20, 4, 4, 1.0)	20.00	0.00	4.00	0.00
(20, 4, 4, 2.0)	20.00	0.00	4.00	0.00
(25, 8, 8, 1.0)	25.23	0.63	7.92	0.44
(30, 4, 4, 2.0)	30.00	0.00	4.00	0.00
(30, 6, 6, 1.5)	30.00	0.00	6.00	0.00

Table 2 – Number of locks (d_l) and Linear Coefficient (d_{lin}) obtained after 100 executions of the EA for each combination of features

Features (u_r, u_k, u_l, u_{lin})	d_l		d_{lin}	
	Mean	Std. Dev.	Mean	Std. Dev.
(15, 3, 2, 2.0)	2.00	0.00	2.00	0.00
(20, 4, 4, 1.0)	4.00	0.00	1.23	0.06
(20, 4, 4, 2.0)	4.00	0.00	1.93	0.07
(25, 8, 8, 1.0)	7.00	0.62	1.42	0.11
(30, 4, 4, 2.0)	4.00	0.00	1.83	0.08
(30, 6, 6, 1.5)	6.00	0.00	1.51	0.01

The EA has almost perfect precision in creating dungeons with the desired number of rooms, keys and locks. For the majority of tests, the dungeons achieved the desired conditions and, when they did not, it was statistically similar except by the number of locks for the 4th parameter configuration in Table 2. For this configurations, the number of keys was close to the desired, while the mean value for locks was close to 7, one unity below u_l . This can be explained by the stochastic property of the fix algorithm, which does not guarantee that the post-fix dungeon will return the desired number of locks and keys.

The only parameter where the algorithm had difficulty to achieve the desired value in most input configurations is the linearity coefficient (u_{lin}). For three cases, there is an average value deviating more than 0.15 from the desired value. This arises a difference as up to 12% of the target value and a standard deviation equal to 1.2%. These results are somewhat expected to happen since the structure of the grid does not support many adjacent rooms with more than 1 child (that is, rooms with more than 2 exits). They would occupy the same space in the grid.

5.1.2 Constrained EA Results

The same 6 sets of configurations are used now to test the Constrained EA (CEA), with the new crossover and mutation operators, and the addition of the A* and DFS path-finding algorithms as stated in Section 4.1.5. As in the previous experiment, a total of 100 executions for each configuration was averaged and the standard deviation estimated. We also added 2 new sets

of configurations in the experiment, testing the robustness of our approach. These sets consider the generation of dungeons with 100 and 500 rooms, something very uncommon in real games, but are good for stress tests. The results achieved for d_r, d_k, d_l and d_{lin} are shown by tables 3 and 4. We also included results for Δ_r, Δ_l , fitness f and execution time Δ_t in tables 5 and 6. The layout of the table remains the same from the previous experiment.

Table 3 – Number of rooms (d_r) and keys (d_k) obtained after 100 executions of the CEA for each combination of features

Features (u_r, u_k, u_l, u_{lin})	d_r		d_k	
	Mean	Std. Dev.	Mean	Std. Dev.
(15, 3, 2, 2.0)	15.00	0.70	3.01	0.17
(20, 4, 4, 1.0)	20.09	0.89	4.00	0.00
(20, 4, 4, 2.0)	20.10	0.59	4.00	0.00
(25, 8, 8, 1.0)	25.00	0.14	8.00	0.00
(30, 4, 4, 2.0)	30.12	1.25	4.01	0.10
(30, 6, 6, 1.5)	29.86	1.11	6.02	0.14
(100, 20, 20, 1.5)	100.00	3.04	20.26	0.56
(500, 100, 100, 1.5)	491.52	28.87	42.37	5.26

Table 4 – Number of locks (d_l) and Linear Coefficient (d_{lin}) obtained after 100 executions of the CEA for each combination of features

Features (u_r, u_k, u_l, u_{lin})	d_l		d_{lin}	
	Mean	Std. Dev.	Mean	Std. Dev.
(15, 3, 2, 2.0)	2.01	0.17	1.96	0.12
(20, 4, 4, 1.0)	4.00	0.00	1.14	0.05
(20, 4, 4, 2.0)	4.00	0.00	1.88	0.07
(25, 8, 8, 1.0)	7.94	0.34	1.22	0.08
(30, 4, 4, 2.0)	4.00	0.00	1.88	0.10
(30, 6, 6, 1.5)	6.01	0.10	1.53	0.02
(100, 20, 20, 1.5)	19.66	0.78	1.47	0.05
(500, 100, 100, 1.5)	36.78	5.02	1.41	0.05

For the first 6 input configurations, the algorithm can still converge in a very accurate way to the desired number of rooms and keys (Table 3), but also to the number of locks (Table 4). The CEA improved the previous results for the 4th set of parameters (25, 8, 8, 1.0). This is explained by the stability during the distribution of locks and keys in the dungeons, which leads CEA to generate levels that converge to the desired number of locks for the majority of the executions. In the average, d_l is 7.94 with an standard deviation of 0.34. The same happens for d_{lin} parameter. Although CEA has a small decrease in convergence for some sets of parameters (1st, 3rd and 6th), it increased the convergence for the 4th one, as can be seen in Table 4.

Table 5 shows that the CEA converge with a very accurate precision to the case where the modified A* algorithm needed to open every lock to reach the goal ($\Delta_l = 0$). The only exception was for the last set of parameters, which presents a high standard deviation and an average value

Table 5 – Difference between number of locks in the level and opened by the A* (Δ_l) and difference between number of rooms visited by the DFS (Δ_r) obtained after 100 executions of the CEA for each combination of features

Features (u_r, u_k, u_l, u_{lin})	Δ_l		Δ_r	
	Mean	Std. Dev.	Mean	Std. Dev.
(15, 3, 2, 2.0)	0.02	0.14	0.40	1.04
(20, 4, 4, 1.0)	0.01	0.10	0.13	1.17
(20, 4, 4, 2.0)	0.01	0.10	0.32	0.62
(25, 8, 8, 1.0)	0.00	0.00	0.05	0.22
(30, 4, 4, 2.0)	0.01	0.10	0.25	0.65
(30, 6, 6, 1.5)	0.11	0.78	0.41	1.79
(100, 20, 20, 1.5)	1.84	0.39	3.47	0.82
(500, 100, 100, 1.5)	5.16	4.44	16.21	6.94

Table 6 – Resulting fitness (f) and execution time (Δt) in seconds obtained after 100 executions of the CEA for each combination of features

Features (u_r, u_k, u_l, u_{lin})	f		Δt (s)	
	Mean	Std. Dev.	Mean	Std. Dev.
(15, 3, 2, 2.0)	0.08	0.22	16.45	0.75
(20, 4, 4, 1.0)	0.27	0.10	20.77	1.70
(20, 4, 4, 2.0)	0.24	0.12	18.93	0.89
(25, 8, 8, 1.0)	0.55	0.70	21.27	0.83
(30, 4, 4, 2.0)	0.25	0.28	22.28	0.69
(30, 6, 6, 1.5)	0.05	0.00	23.44	0.59
(100, 20, 20, 1.5)	1.42	1.63	72.39	2.62
(500, 100, 100, 1.5)	250.96	19.72	610.71	66.43

of 0.11. Thus, for some cases, the EA could not find an individual (dungeon) where all locks should be used to reach the goal. It was more difficult for CEA to generate dungeon's structures where the DFS needs to visit all the rooms to reach the goal ($\Delta_r = 0$). In this case, the standard deviation is greater than 1.0 and the averages are greater than 0.2 in 4 cases. Besides the non perfect convergence, these results are still interesting since the difference is still low, meaning that for a reasonable number of executions the CEA was able to fit well in this condition. Also, these results show that the number of unused rooms was low based on a standard deviation not greater than 2.

Table 6 shows that the convergence of the algorithm was satisfactory. The CEA returns an average fitness of 0.55 with a standard deviation of 0.7. Thus, the fitness did not exceed 1.0 for most cases, that is, a slight difference between desired and created dungeons was present. The mentioned difference was a lack of single normal or special room, or a single room or locked door not visited. Thus, even for the worst sets of parameters, the dungeon was almost exactly as the user desired, and almost every lock and room was needed.

In the same table, we can see that the execution time increased significantly from less

than 3s on average, in the first EA, to more than 23s in some cases for the CEA. This is explained by the path-finding algorithms used to simulate players' behavior. Although significant, this time may be considered irrelevant even for online PCG, if executed in the background together with the gameplay (generating the next dungeon concurrently as the player plays the last one). Also, the algorithms were run in a single-thread. If we execute them concurrently for many levels in the EA's population, we can significantly reduce this time, although further tests are required to measure the time reduction obtained when both approaches are used.

As for the 2 stress-test input configurations, we can observe that the algorithm converged accurately to the desired number of rooms, keys, locks and linearity for the 7th line, with 100 rooms, although it left some locks unopened (1.84, on average) and some rooms unvisited (3.47, on average). That lead to an average fitness of 1.42, which is still acceptable, and converged after a little more than 1 minute of execution. These results show that for large dungeons with as many as 100 rooms, the algorithm can still converge very accurately.

However, for the configuration in the 8th line, with 500 rooms, the algorithm could only converge with some accuracy to the desired number of rooms and linearity coefficient. The number of locks and keys didn't even go as much as half as the desired number, on average more than 5 locks were left unopened and more than 16 rooms unvisited (which, proportionately to the dungeon size, is not as much of a problem, as it corresponds to less than 4% of error). This lead to an average fitness of 250.96, implying the algorithm is not able to create good gargantuan dungeons. As for the execution time, we observe it can create levels after an average of 10 minutes, which should not be considered as good enough for any kind of online generation.

5.2 Experiments with Users

The second experiment aims to gather player's feedback about the generated maps, by evaluating how similar they are to those designed by humans in terms of perceived difficulty and fun. The level of diversity generated by the algorithm is also measured. The overall quality of a level is hard to measure, since dungeons cannot be evaluated only by their maps and locked door missions but also by the content of the rooms, the game mechanics, game design, artwork, audio, etc. In order to avoid as much bias as possible, the experiment shows to humans a total of 6 different maps: 3 adapted from *The Legend of Zelda* and *The Legend of Zelda: Oracle of Ages* games, and 3 generated by the EA. The levels created by the EA use fitness' parameters taken from the Zelda's dungeons analysis, aiming to generate similar levels. The adapted levels were remade with the same original dungeon's layout, but all the content of the rooms was removed. Also, the art style for all dungeons was changed to make them visually equivalent.

The maps were built with a template sprite of an empty room from one of *The Legend of Zelda*'s dungeons, and each connection between rooms was represented with the same sprite for a door of the dungeon. The keys were represented with an enlarged key sprite from the game,

the locks with an altered sprite from a locked door within the game, and its colors changed to be more easily perceived. The starting room was painted in blue and pointed with a red arrow, while the final room has an enlarged golden triangle sprite. The resulting maps presented in the survey can be seen in Figure 23, which names each human-made dungeon and the parameters used to create its EA-generated pair. Each dungeon is identified by a number representing the sequence in which they were presented to users. This number was chosen at random to avoid bias in the sequence of evaluation. Finally, the online survey¹ asks some questions following the image of each map. Annex B shows the maps in more details.

Figure 23 – Left, from top to bottom: *The Legend of Zelda's* The Demon Dungeon adaptation (Dungeon 5), *The Legend of Zelda's* The Lizard Dungeon adaptation (Dungeon 2), *The Legend of Zelda: Oracle of Ages'* Spirit's Grave Dungeon adaptation (Dungeon 4) . Right: EA-generated dungeon and its parameters. From top to bottom: 32 rooms, 4 keys, 4 locks and linear coefficient of 1,22 (obtained from LC calculations over the Zelda's dungeon) (Dungeon 1). 23 rooms, 7 keys, 6 locks, and LC of 1,09 (Dungeon 6). 21 rooms, 4 keys, 4 locks and LC of 1,05 (Dungeon 3).

¹ <<https://goo.gl/forms/GVYtiXutrRq5aq112>>

This representation made the maps much similar to the ones present in older *The Legend of Zelda* games, and easier for players to identify elements and make analysis. By eliminating all elements from the rooms, the analysis would only take into account the map, not the rooms' contents. The use of a common art for all levels would also remove bias from the user's personal likes and dislikes on artwork. A total of 3 questions were asked for each map and the users had to answer following a 5-point Likert scale, from strongly disagree to strongly agree: "*This dungeon would be fun to play*", "*This dungeon would be difficult to complete*" and "*This dungeon was created by a human*". The first question aims to measure if the design of the map and the puzzles were at least as fun as it could be perceived from the human-designed ones. The second was used to evaluate if the fitness parameters chosen for the EA and the resulting levels were enough to guarantee a similarity in terms of perceived difficulty. The last question works as a Turing test. The idea is to check if the levels could trick users as thinking it was made by humans instead of an algorithm. For every dungeon, its image, as in Figure 23, is presented and immediately followed by the 3 questions.

The online survey was answered by 97 anonymous users, which had access to the form via Facebook posts in groups of game players and university-related groups. A total of 95 users was from 18 to 35 years old, and the other 2 above 35 years old. The frequent players are 50%, while another 30% called themselves as hardcore players and 17% as casual players. The other 3% represented people that never played once or almost never played a video-game. The adventure game players were more than 80% and 3% never played a game from the genre. *The Legend of Zelda* games were played for many hours by 48% of them, another 30% had played for few hours, and 22% never played a game from the series. About PCG, a total of 27% knew how PCG worked, 42% knew only the basics about it, and 31% did not know what it is.

Figure 24 shows results for Turing test and fun. The results do not differ much between original and procedurally generated maps, but the Zelda maps have a few more votes for the positive opinions about being made by humans. The EA-generated ones have more votes on the indecisive option (3) and the weak opinion of it not being made by human. For the fun evaluation, Zelda's levels have some more votes for the weak opinion of it not being fun and the strong opinions of it being both fun and not fun. The EA-levels have more votes in the neutral opinion.

Each EA-map should be also compared to the Zelda map that originated its fitness criteria. Thus, we evaluate the answers pairwise, considering the human-made dungeon and the one generated using its parameters as input. The answers are presented in Table 7 where all dungeons are presented in their respective pairs, separated by '-'. The evaluation is done using a t-Test analysis to verify if the null hypothesis that the responses for each pair of dungeons differ significantly can be accepted or rejected. The t-Test was chosen since it is suitable for a five-point Likert scale (WINTER; DODOU, 2010). The pairs of answers had their variance tested by an F-Test. All of them rejected the null hypothesis that the variances in each set of answers were equal.

Figure 24 – Average of the sum of votes in each group of levels for the Turing test (left) and the evaluation of fun (right). 5 means a strong positive opinion evaluation of fun (strong agreement that the map was made by a human/was going to be fun to play), and 1 means a strong negative opinion (strong disagreement that the map was made by a human/was going to be fun to play).

Table 7 – F-Tests followed by t-Tests for all 3 pairs of dungeons for the 3 questions in the survey. All F-tests had a Degree of Freedom of 96 and t-Tests, 192. The t-Tests considered a Hypothesized Mean Difference of 0.0.

3 - 4	Human	Fun	Difficulty
F	1.1782	1.1313	1.2661
FC	1.4013	1.4013	1.4013
tStat	1.8145	<u>-2.1186</u>	<u>2.6421</u>
tC	1.9724	<u>1.9724</u>	<u>1.9724</u>

1 - 5	Human	Fun	Difficulty
F	1.2812	1.2852	0.6663
FC	1.4013	1.4013	0.7136
tStat	1.9208	0.7396	-0.7694
tC	1.9724	1.9724	1.9724

2 - 6	Human	Fun	Difficulty
F	1.0989	1.0028	1.1065
FC	1.4013	1.4013	1.4013
tStat	-0.9914	-0.6271	1.0125
tC	1.9724	1.9724	1.9724

Table 7 shows the inequality $F < FC$ is true for the Critical(C) value of F, satisfying the one-tail test. Therefore, the t-Test assuming equal variances was conducted. Except by the evaluation of Fun and Difficulty for pair of levels 3 and 4 (underlined in the table), all the other tests couldn't reject the null hypothesis that the means of the populations are equal. They satisfied the two-tail test inequality: $|tStat| < tC$ for the Critical (C) value of t. Thus, except for both cases underlined in Table 7, we can consider users' evaluation of levels to be statistically the same.

5.3 Room Generation Results

The results achieved by the approaches for room generation, based on SFCs and CA described in section 4.2, are presented in this section. Figure 25 gives some examples of rooms created. The art used in it is well-known by players and developers since it uses sprites from *The Legend of Zelda* game. It is also illustrated rooms created using different configurations of parameter setting in our generator, all of them for creating rooms with size 12x12. The execution time of every approach has been tested and it has never spent more than 1 millisecond.

Figure 25 – Examples of rooms created applying SFC and CA. Both top levels were created using Hilbert curve, order 4, *width* = 2. The left one has 2 doors and the right one has 4. Second row consists of two levels created with Moore curve with the same parameters, but left one has 4 doors and right one has 3. Third row, on the left, contains Hilbert-generated room with same rules, 4 doors, but aided with the CA rule B3/S23. On the right the level is generated with Hilbert, order 4, width 1 and 3 doors. Then, the last row contains another pair of Hilbert-created rooms with 3 doors. On the left, width 3 and order 4, and on the right, width 2 and order 3.

The computational experiments were executed 10000 times for each set of parameters evaluated. To simplify the naming of the configurations, it will follow the pattern: DXCXOXWX, where X is the number placed as input for parameters: Number of Doors (D), Space-filling curve (C) (1 is Hilbert and 2 is Moore), Order of the curve (O) and Width of the passage (W). The automaton rules, when applied, are abbreviated following the rules shown in section 4.2.

We use already established and tested metrics from literature to evaluate the proposed algorithm. The first one is density (d) (HORN *et al.*, 2014) which, adapted to the context of our levels, will measure the number of blocks (nB) divided by the total number of tiles (nT) as defined by equation (5.1).

$$d = nB/nT \quad (5.1)$$

Different from platform levels, where linearity and density can be combined to measure the distribution of hills along levels, the linearity of top-down dungeon levels is not so intuitive.

Moreover, we decide to use density for expressing the amount of free space that the players have to navigate through the rooms. Another metric is symmetry (s) (SUMMERVILLE *et al.*, 2017) but excluding the area from the original formula in our calculations, since all tiles have the same area. This metric captures the intuitive notion of symmetry, by dividing the level in half horizontally and vertically, calculating the distance between objects from one quadrant to the other, and summing all the distances.

In the case of using the same set of objects to create a level, a lower value means the level is visually more appealing to human players (MARIÑO; LELIS, 2016). It is important to note that, when using different objects, the influence of symmetry may be different (SUMMERVILLE *et al.*, 2017). The final metric is exploration (e), related to learning the layout of the play-space (LIAPIS; YANNAKAKIS; TOGELIUS, 2014). The metric uses a flood-fill algorithm to explore the level, starting from a special tile called reference tile and ending in another one. In our case, the reference tiles are doors since we measure the exploration of the whole dungeon from one exit to another. The higher the exploration value, more the player needs to explore the room to reach the exit.

Combining those three metrics, we expect to measure the expressive range of the method mainly with the density and exploration metrics. The game designer may want to create dungeon rooms with different values of both metrics to set the level of difficulty. For instance, a dense room with many enemies may be harder than a room with low density but the same amount of enemies. In the former, the player will have less space of running and positioning its character. A room with lower exploration value will be much faster to traverse. The symmetry value will also be tested for the expressive range, but we focus on examining its mean value for the generated rooms as a way to measure its visual quality. Boxplot visualizations compare the three parameters for all sets of input as presented in (SMITH; WHITEHEAD, 2010; HORN *et al.*, 2014).

First, it is evaluated the influence of the number of doors in a room by executing the algorithm 10000 times selecting a random number of doors: 2, 3 and 4 rooms selected for each execution. The results showed that the number of doors has a direct impact in the expressive range of the results. As the number of doors increases, the range of each metric is reduced. In order to avoid bias from this parameter, all other experiments were conducted only on levels with 2 doors.

Next, we analyze the distribution of the quartiles and average of each metric for each combination of parameters using a box & whisker visualization. Previous results showed no relevant differences between Hilbert or Moore curves for all sets of parameters as illustrated by Figure 26 for D2C1O4W1 (Hilbert) and D2C1O4W1 (Moore). Thus, the next analysis will be done considering only levels generated by Hilbert curve (C1). The symmetry values were normalized according to the maximum range of values for all data, aiming to keep the same range for all metrics.

Figure 27 has a box & whisker visualization grouping all tested parameters, where the

Figure 26 – Box & Whisker visualization of rooms generated with different parameters. Orange box is the density, blue one is the exploration value and green one the symmetry. Each 3 sequential metrics correspond to a combination of parameters, identified according to our previously established nomenclature.

most expressive generator for symmetry and density is achieved using the set of parameters D2C1O4W3. For exploration, there was no significant difference between most of the parameters. From the same visualization, the rooms generated only with the CA are those that create the most aesthetically pleasing levels on average. They concentrate levels with lower symmetry values which is a expected results. The CA returns less populated levels that favor symmetry, and its rules tend to create balanced structures. The CA also presents the lowest values for density and exploration, as the rules lead to sparsity and small structures over well-defined lines as the SFC. For the SFC-generated rooms, parameters D2C1O4W1 returned the most symmetrical ones. Finally, the rooms built with the aid of automata present a decrease in the density and exploration metrics. This is also expected, since the OR rule always keeps or adds blocks to the level, increasing density and exploration values while decreasing the chance to generate lower-valued ones.

For density and exploration metrics, we also present heatmap visualizations for all previous combination of parameters. The y-axis is the density and the x-axis is the exploration value within each heatmap, both ranging from 0 to 1. The color of each cell measures the amount of created levels that was evaluated with the corresponding metrics. All the colors were normalized to the range of the maximum and minimum values for each experiment, becoming easier the visualization. The name of each set of parameters evaluated is shown bellow each figure in Table 8.

The center column of the table compares the differences generated by changing the order of SFC. We can see the expressive range is almost the same, although the order 3 curve (top center) generates more diverse levels than the other curve orders, which concentrate more

Figure 27 – Box & Whisker visualization of rooms generated with different parameters, all of them with 2 doors. Blue box measures the density, the green one is the exploration value and the red one is the symmetry. Each 3 sequential metrics correspond to a combination of parameters, which are identified at the bottom of the graph.

levels in density values from 0.4 to 0.6 and exploration values from 0.5 to 0.7. The middle row presents the results for varying the width of the curve, showing that it has a greater impact on its expressiveness since they are directly proportional mainly considering the density of rooms. These results confirm our evaluation from the box & whisker visualization analysis.

Thus, the SFC-only method can be used to generate both levels for games with intended maze-like levels and combats in corridors (high density) or arena-like stages, with exploration of more open areas (low density), while maintaining complexity of the exploration free to change between easy (low values) and difficult (high values).

The bottom left and right images in Table 8 show the results for the use of the automaton together with the SFC. Both automata increased the overall density of generated levels, while maintaining most of the exploration range of values, as already shown in Figure 27. The A12B1 automaton specially reduces a lot the expressive range of the generated levels. This mixed approach may be better to generate maze-like levels. Finally, the top left and right images show the heatmap for automata-only generated levels. They are also able to maintain a large range of exploration values, but they are very limited to generate only low density levels. Thus, the CA-only approach may fare better generating levels with large exploration areas.

Table 8 – Exploration (x-axis) vs Linearity (y-axis) Heatmap Visualizations for SFCs with different Order and Width, for SFCs aided with CA, and for levels generated only using the CA. All heatmaps were generated using 10,000 randomly generated levels for each set of parameters. The closer to the white color, more levels have been created that satisfy the corresponding combination of metrics

5.4 Experiments with the Game Prototype

An adventure game prototype (ProCoG) was developed and described in section 4.3, which will allow to play the dungeons generated by this research. The game was uploaded to a website², and anyone with a PC's browser could access it. The link was advertised via social media without restriction for the target audience. Figure 28 shows 4 different in-game rooms from the player's view of the game. The top-left image shows one of the starting rooms, the top-right shows the final room and the item the player must pick to finish the level. The bottom-left image contains a room with a key in its center, and the last one has a locked door. The key-lock pair is identified by its color, keeping in this way the visual representation from Figure 14. Therefore, the yellow key opens the yellow locked door in the bottom image. As the

² <jogos.icmc.usp.br>

original Zelda games did not have unique key-lock pairs, all the dungeons in this experiment had generic keys and locks, all of them colored in yellow.

Figure 28 – Screen-shots from the player’s view in-game. Top-left: the starting room. Top-right: the final room. Bottom-left: a room with an key; Bottom-right: the room with a locked door.

We conducted 2 different experiments using our prototype: the first evaluated the quality of the dungeons generated by the CEA alone; the second one measured the performance of the CEA together with our room generating method. Each part of the experiment had the same setting: the user would play 6 different levels, 3 adapted directly from *The Legend of Zelda*, and 3 generated by our CEA using the same parameters of the level as input (number of rooms, keys, locks, and linearity coefficient). The levels adapted from the original game had only minor alterations to account for non-adjacent rooms, connected by staircases in the source (thus, teleporting the player), or rooms opened only by bombs. In the first scenario, the rooms were translated in order to be adjacent, and a door was placed between them to replace the staircase. No other doors were added or removed, but some rooms were rotated to avoid overlapping. For the second scenario, the destructible walls were replaced by an open door. The sequence that the

levels were introduced to the player for each part was randomized to eliminate any bias based on such sequence.

For each original *The Legend of Zelda* level, we used the same procedure to create analogous levels with our CEA as presented in Section 5.2. In order to better distinguish the original dungeon from ours, from here on the original levels will be referred by their name followed by the *Zelda* word (as in *MoonZelda*). Their counterpart created by our CEA will be referred by the original level's name followed by the *CEA* word (as in *MoonCEA*). This will be maintained for the second part of the experiment described in 5.4.2. The fitness' input for each CEA-generated level is presented in Table 9, as well as the fitness of dungeon. All the rooms, keys and items of the created dungeon converged perfectly to the input value, so it was omitted from the table. The same applies to the linearity coefficient, except for the *SnakeCEA* map, that converged to $d_{lin} = 1.18$, a 0.01 difference from the input.

Table 9 – Input parameters and fitness for each CEA-generated dungeon presented in ProCoG.

Dungeon	u_r	u_k	u_l	u_{lin}	f
DragonCEA	25	5	5	1.33	0.007
LizardCEA	23	6	6	1.38	0.01
MoonCEA	18	4	4	1.89	0.002
EagleCEA	17	6	6	1.33	0.007
LionCEA	25	4	4	1.41	0.004
SnakeCEA	20	4	4	1.19	0.005

For each map, we saved some important gameplay data from the user after the level completion, such as number of keys taken, number of locks opened, total number of rooms visited, number of different rooms visited and the time elapsed to complete the dungeon. Also, after each dungeon was completed, the user was asked the same three questions from 5.2 in a survey developed inside the Unity engine, making it a part of the game itself. Figure 29 shows the in-game survey in Portuguese. The questions, all of them in a 5-point Likert scale, from top to bottom are: "This dungeon was created by a human", "This dungeon would be fun to play", and "This dungeon would be difficult to complete".

All dungeons extracted from *The Legend of Zelda* were recreated using the *Tiled Map Editor* tool ³, and processed to adequately represent a map and rooms files equivalent as the ones generated by the CEA and SFC algorithms. These files were then used as input for the game prototype in Unity and loaded as a dungeon in the same way as the procedurally generated ones. The resulting dungeons are presented in Annex B.

We will report next each of the two parts of the experiment. They were played sequentially by the user from the beginning of the game. Moreover, the 6 first levels the user player were from the first experiment, and the last 6 for the second one. As the main focus of our research

³ <<https://www.mapeditor.org/>>

was the CEA, we divided the test in such way to have the least amount of dropouts as possible for the first part, as the players could close the game anytime they wanted to.

Figure 29 – Screenshots from the in-game survey. The previously presented questions were translated into Brazilian Portuguese as the survey was published in medias mainly accessible to Brazilian users.

Parabéns! Você completou o nível!
Agora, gostaríamos de saber a sua opinião sobre ele. É só responder às perguntas abaixo, clicar em "Enviar", que um novo nível será carregado para você jogar :)

O quanto você concorda com a afirmação: "Esse nível foi criado por um humano?"

1 = Discordo Fortemente   <->   5 = Concordo Fortemente

1   2   3   4   5

O quanto você concorda com a afirmação: "Esse nível foi divertido de jogar?"

1 = Discordo Fortemente   <->   5 = Concordo Fortemente

1   2   3   4   5

O quanto você concorda com a afirmação: "Esse nível foi difícil de ser completado?"

1 = Discordo Fortemente   <->   5 = Concordo Fortemente

1   2   3   4   5

Enviar

5.4.1 Part 1 - CEA-Only Maps

As previously stated, in this first experiment 6 different maps were sequentially played by the user, drawn in a random order. A total of 3 maps were extracted from *The Legend of Zelda* game: *The Lizard*, *The Dragon* and *The Moon*. The other 3 maps were created by the CEA using the original level's parameters. For this stage of the experiment, all the rooms in each map were empty squares of 6x6 tiles.

Thus, we eliminate any interference from the configuration of the room in the players' opinions of the dungeon. Thus, the player will evaluate only the map configuration of the dungeon and the locked-door missions created for it. The total number of players which completed the levels was different, since the sequence that each player found the levels was different. Therefore, to evaluate all levels with the same amount of data, we used the minimum number of players that completed all levels: 70. The data from those players were analyzed as follows.

First, we evaluate the results from the survey. Figure 30 shows the average answer (ranging from 1 to 5) for the users for each of the 3 questions. The *Zelda* dungeon and its counterpart share the same color hue, with the original one presented in a darker shade, and the CEA one with a lighter one. We can observe that our CEA levels were all, on average, evaluated as more human-made, funnier and more difficult than the *Zelda* ones. This is a very positive result, since our algorithm was able to make dungeons that looked like human-made while entertaining the players. On the other hand, the rise in difficulty by itself cannot be interpreted as a positive or negative factor. We can argue that the difficulty is one of the features that may have increased the fun in the level, considering players that enjoy challenges.

Before delving further in the game data analysis, we will better explore the data obtained through the survey by conducting an statistical analysis on it with the same methodology applied in Section 5.2, using an α value of 5%. For each *Zelda*-CEA pair we conduct an *F-test* analysis

Figure 30 – Average Survey Answers for the First Part of the Experiment

to check if the variance of both population of answers from the same topic can be considered equal or not. If the *F-test p-value* in Table 10 is lesser than 0.05, we conclude the variances as different. Otherwise, the null hypothesis that they are the same cannot be discarded. Therefore, we apply a homoscedastic or heteroscedastic t-test for the data based on the previous result.

For the t-test, we apply an *one-tailed t-test* with the null hypothesis being the means of both population are equal and the alternative hypothesis being that the mean of the population from the *CEA* data is greater. This can confirm if the data observed in Figure 29, that the *CEA* levels had higher scores, holds true or not. If the *p-value* for the *t-test* is lesser than 0.05, we can conclude that the results for the *CEA* are statistically greater than the ones from the *Zelda* dungeons. Otherwise, they can be considered equal.

Table 10 – F-tests followed by one-tailed t-tests for all 3 pairs of dungeons for the 3 questions in the in-game survey in the first-half of the experiment with the game prototype. All tests considered the answers from 70 different users and an α value of 5%.

Lizard	Human	Fun	Difficulty
F-test p-value	0.7394	0.0253	0.0220
T-test p-value	0.1213	0.0942	0.0065

Moon	Human	Fun	Difficulty
F-test p-value	0.3405	0.5013	0.3991
T-test p-value	0.0063	0.0025	0.0000

Dragon	Human	Fun	Difficulty
F-test p-value	0.9572	0.9907	0.3475
T-test p-value	0.0452	0.0151	0.0226

From the results in Table 10, we can conclude that the *CEA*'s was statistically, on average, considered more difficult for the *Lizard* pair of levels. However, it was considered equally developed by a human designer than the *Zelda* dungeon, and equally fun. On the other hand, both *Moon*'s and *Dragon*'s pair of dungeons held the same results: the *CEA*-generated maps were considered as more likely to have been created by a human, funnier and more difficult.

Next, we analyze the average usage of keys and locks by the players in each level, as well as the total number of different rooms visited. Figure 31 presents these data in a similar fashion than the previous image. Each group of columns shows the average difference between the total number of available keys, locks and unique rooms and the number of each of them taken/used/visited by the player. For every pair of dungeons, we can observe that the *CEA* counterparts lead to a better usage of resources, leading the player to use, on average, almost every available resource on the map before completing it. At least in terms of the layout and locked-door missions, those results also confirm that the path-finding algorithms, used as part of the fitness function by the *CEA*, helped to create dungeons that demanded a complete exploration by the player. The results confirm what was observed in the previous results reported in section 5.1.2, where it was showed that the convergence of the *CEA* for desired traits was always almost perfect.

Figure 31 – Average Rooms, Locks and Keys Usage for the First Part of the Experiment

Another interesting aspect of our dungeons, when compared to the original *Zelda* ones, is that they demand considerably more *backtracking* through the player's path. This can be observed in Figure 32, where the *Zelda* maps had an average revisiting ratio (total rooms visited by the players divided by the number of unique rooms visited by them) slightly above 1.5, while the *CEA* ones presented a ration above 2.0 that almost reaches 2.5 for *DragonCEA*. Although

Figure 34 – Bubble Chart Representing the Amount of Votes for Each Fun and Difficulty Combination in the In-game Survey for the MoonZelda (left) and MoonCEA (right) Maps

Figure 35 – Bubble Chart Representing the Amount of Votes for Each Fun and Difficulty Combination in the In-game Survey for the DragonZelda (left) and DragonCEA (right) Maps

One of the first observations that can be made is that both the fun and difficulty votes in the *CEA* maps (right of each figure) are concentrated in higher values, as expected since they were better graded. Another interesting observation is about the *Zelda* levels (left of each figure). The majority of votes in such levels did not consider this type of level much difficult (hardly above 3 and more concentrated in the 1 and 2 answers). The easiest level of the *Zelda* ones, *MoonZelda*, also had the biggest concentration of votes on the lower side of the difficulty variable and labeled as not fun. This can be observed in the bubbles at coordinate (1, 1) and (2, 1) in Figure 34. Both other *Zelda* dungeons (figures 33 and 35) concentrated more neutral responses (ranging from 2 to 4) for *fun* in a region of lower *difficulty* (ranging from 1 to 3). This initial evaluation may be a clue that users have a tendency to enjoy more when the level has at least a certain threshold of difficulty.

Now, evaluating the *CEA*-generated levels, we can see a bigger concentration of neutral to fun-agreeing votes (3 to 5) in a central perception of difficulty (2 to 4). This hold true specially in the *DragonCEA* and *LizardCEA* dungeons, evaluated as the most difficult ones. When comparing each pair of dungeons, we can confirm this pattern of increasing fun when the difficulty also increases for every pair of maps.

Therefore, based on the collected gameplay data and the answers from our survey, there is a certain positive correlation between perceived difficulty and fun.

5.4.2 Part 2 - CEA + SFC Maps

In the second part of the experiment, the same procedures from the previous one were followed, except that the dungeons created by the CEA had all their rooms created by the SFC algorithm. We generated the rooms using *order 4 Hilbert curve* with *width* equals 2 and an OR overlapping after 10 iterations of the CA. The 3 maps extracted from *The Legend of Zelda* game were: *The Snake*, *The Lion* and *The Eagle*. The rooms from these dungeons were also extracted from the game and recreated using the *Tiled Map Editor*. Therefore, this second experiment evaluates the influence of each room's layout over the players opinion about the dungeon. Again, since the sequence that each player is introduced to the levels is different, we also assumed the minimum number of players that completed all of them: 44 players.

First, we evaluate the results from the survey. Figure 36 shows the average answer of the users for each of the 3 questions. The chart configuration follows the same rules as in the first part of the experiment, although we now use the CEA together with the SFC room-generating technique. We can observe that now our *CEA* levels were all, on average, evaluated as less human than the *Zelda* ones. Our room-generating approach is not able to fool players into thinking it was made by a human designer. A probable motivation behind these results is that the generated rooms were less symmetric and denser than the human-made ones, as we can see in Annex A. Generally, levels from action-adventure games are less denser than ours and, as we already discussed from Mariño and Lelis' work, more symmetric levels are considered more appealing to players and commonly designed by developers (MARIÑO; LELIS, 2016). For the fun and difficulty variables, the results depended on each dungeon.

Figure 36 – Average Survey Answers for the Second Part of the Experiment

For the *Lion* pair, the *CEA* one was deemed as funnier and more difficult than the original

one. For the *Snake* pair, it was the opposite, with the *CEA* turning as less fun and slightly less difficult than the *Zelda* one. The last pair, *Eagle*, was evaluated as slightly more fun and slightly less difficult in the *CEA* variation. Before drawing any conclusions from this data, we will also analyze the data with the same statistical measures as in the first part of the experiment.

The α value of 5% was used with the *f-test* and *t-test* probabilities calculated for each *Zelda-CEA* pair. The results are presented in Table 11

Table 11 – F-tests followed by one-tailed t-tests for all 3 pairs of dungeons for the 3 questions in the in-game survey in the second-half of the experiment with the game prototype. All tests considered the answers from 44 different users and an α value of 5%.

Snake	Human	Fun	Difficulty
F-test p-value	0.9116	0.6498	0.1840
T-test p-value	0.0083	<u>0.2971</u>	<u>0.4660</u>

Lion	Human	Fun	Difficulty
F-test p-value	0.9154	0.6691	0.4760
T-test p-value	0.0063	<u>0.2558</u>	0.0338

Eagle	Human	Fun	Difficulty
F-test p-value	0.5149	0.8827	0.6448
T-test p-value	0.0012	<u>0.4570</u>	<u>0.4232</u>

From Table 11 we can assume that, statistically, all the *CEA* levels were, on average, less human-made than the *Zelda* ones. Which confirms that our approach for room generation was not organic enough to trick players into thinking it was not procedurally generated. However, for all dungeons the perceived fun could not be discarded as statistically equal. Moreover, even if the levels were perceived as not designed by humans, the players were able to have as much fun as the ones they perceived as human-made. For the difficulty, only the *LionCEA* map was significantly harder than the *LionZelda* one, and the other ones show the same level of difficult.

The average usage of keys and locks as well as the total number of different visited rooms was also evaluated. Figure 37 shows similar observations from the first part of the experiment for the 6 new dungeons. The number of backtracking inside levels was also similar from the first part of the experiment, except in the *Snake* pair, where the *Zelda* level demanded more backtracking than the *CEA* one, as shown by Figure 38. Comparing the resource usage data for this pair and the revisiting ratio, a possible conclusion is that the lock and key placement by the *CEA* was somewhat easy for the player to find in a linear path. This lead us to conclude that all players have found all keys and opened every lock to complete this level. They have, on average, skipped two rooms from the dungeon. However, the backtracking was still present, once the ration was of almost 1.5, meaning that the player revisited on average a number of rooms equal to half the total number of rooms.

Figure 37 – Average Rooms, Locks and Keys Usage for the Second Part of the Experiment

Figure 38 – Average Ratio of Room Revisiting for the Second Part of the Experiment

---

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The research presented in this paper introduces a multi-algorithm method to generate levels consisting of dungeons, rooms and locked-door missions. The levels can be used in different game genres, but an adventure game prototype was successfully built using the levels generated by the method as a proof of concept.

The primary algorithm is an EA able to generate dungeon maps and locked-door missions simultaneously. We represented the dungeon map as a tree structure, where each node was a room, and each edge the connection between rooms. The rooms could belong to three different types: normal, key (which contained a key inside it) and locked (meaning the edge between it and its parent was locked). The EA's operators consisted of crossing branches from different trees and mutating the number of keys and adding or removing leaf nodes.

In a first approach, the algorithm was able to create dungeons with the desired number of rooms, locks and keys, and linearity for different values of each of them. To further test the quality of these first results, we created graphic representations of some generated levels, some original *The Legend of Zelda* dungeons and asked users to grade them in levels of perceived fun, difficulty and if the map was created by a human or not. We concluded that there was no relevant difference about the levels being human-generated, and no difference in the perceived difficulty and fun for most of the cases. This step of our research was reported in an article published in the 2018 IEEE Congress on Evolutionary Computation (IEEE CEC 2018) ([PEREIRA; PRADO; TOLEDO, 2018](#)).

A second version proposed for the EA was able to apply two different path-finding algorithms (A* and DFS). The idea was to measure the exploration of the generated dungeon and to add this parameter in the fitness function. This version of the EA was able to create levels that matched both the designer inputs, and to place the special rooms in a way that leads the

player to explore the whole level to find the exit.

A secondary approach, based on space-filling curves and cellular automata, is tasked to create the content of the rooms for the EA-generated dungeon. These rooms were represented by a matrix of tiles, where each entry could represent a walkable way or a block. The room was created by sectioning a bigger matrix, created by the SFC. Then, the doors are placed and the path between them created, if needed. The room may also have its content overlapped by another matrix, created using the CA, adding even more variety. The room generation approach was evaluated by its expressive range, and it shows potential in returning rooms with different levels of density, exploration and symmetry. Thus, it may be used as a good PCG algorithm for random levels.

The maps generated by the EA had their rooms populated using the room-generating method and the whole dungeon was saved in a file. This file was loaded in an action adventure game prototype made with Unity and turned into a playable level. The player needed to collect the keys spread throughout the dungeon to open the colored locks and find the exit. Thus, providing a proof-of-concept that the algorithms can be applied to real games.

This game prototype was populated with 12 different levels, 6 extracted from *The Legend of Zelda* game and 6 created by our CEA using as input parameters from each of the original dungeons. The first 3 pairs contained empty rooms and tested the quality of the CEA method alone, while the last 3 pairs had their rooms extracted from the original levels and their counterparts' rooms were created by our room-generating algorithm. The dungeons were tested by many players and the resulting data showed that the dungeons generated by the CEA were at least as human-made as the original ones. They were also evaluated as funnier to player and more difficult than the ones from the *The Legend of Zelda* game. The CEA guaranteed a better use of the dungeon's resources, including the locks and keys that formed the main mission of the game, and more backtracking in the generated maps.

The room-generating technique, however, created rooms perceived as different from the human-generated ones. The dungeons in which they were inserted were still evaluated with the same level of fun and difficulty than their counterparts. Therefore, we can assume our PCG technique to generate dungeons and their mission as successful, being able to generate fun and challenging levels that could even trick humans into evaluating them as human-designed. For our SFC aided by CA algorithm, although it was not able to disguise itself from being procedurally generated, it was still perceived as fun. This means it can still be used to create interesting and feasible rooms in a very short amount of time.

6.2 Future Work

One of the future steps of this work is to integrate the PCG algorithm directly into the game prototype, reducing the time to create the dungeons as they won't need to be saved in files

and processed afterwards by the game. Also, we plan to add more narrative elements for the game and combat mechanics in order to become it more enjoyable. Next, we may release it as an entertainment game to validate the commercial value of our algorithms.

We also plan to model a generic output format for the generated dungeons and a friendly interface for it in the *Unity* engine. Thus, we can release a *Unity package* containing our algorithms in the *Unity Asset Store*, making them available for other game developers to boost the productivity of their games.

Some algorithm-related possible advances in the work include the use of other room-generation methods in literature to compare the quality of the rooms with our novel SFC-based method. Also, some other techniques can be applied to aid the method to be perceived as more human-made.

For the map-generating EA, some of the possible extensions could be to generate special rooms filled with enemies or power-up items in it and test if it can also converge to the designer's input for those new parameters and place those rooms in an interesting pattern for the player. Some constraints like the ones from (HORSWILL; FOGED, 2012) may be used for fitness evaluation and added to our equation 4.3. The EA itself may be optimized for real-time by using threads. We can add more elaborated fitness-based stop criteria such as when a specific fitness value is reached, or when there is no upgrade in the fitness after a fixed amount of generations.

BIBLIOGRAPHY

ADAMS, C.; PAREKH, H.; LOUIS, S. J. Procedural level design using an interactive cellular automata genetic algorithm. In: **ACM. Proceedings of the Genetic and Evolutionary Computation Conference Companion**. [S.l.], 2017. p. 85–86. Citations on pages 46 and 54.

ADAMS, E. **Fundamentals of Game Design**. 2nd. ed. Thousand Oaks, CA, USA: New Riders Publishing, 2009. ISBN 0321643372, 9780321643377. Citation on page 27.

ADAMS, E.; ROLLINGS, A. **Fundamentals of game design**. [S.l.]: Upper Saddle River, NJ, 2006. Citation on page 33.

ANTONIADES, T. **Hellblade: Senua's Sacrifice**. 2017. Microsoft Windows, PlayStation 4. Available: <<http://www.hellblade.com/>>. Citation on page 34.

ASHLOCK, D. Evolvable fashion-based cellular automata for generating cavern systems. In: **2015 IEEE Conference on Computational Intelligence and Games (CIG)**. [S.l.: s.n.], 2015. p. 306–313. ISSN 2325-4289. Citation on page 54.

ASHLOCK, D.; MCGUINNESS, C. Automatic generation of fantasy role-playing modules. In: **2014 IEEE Conference on Computational Intelligence and Games**. [S.l.: s.n.], 2014. p. 1–8. ISSN 2325-4289. Citation on page 57.

BELL, I.; BRABEN, D. **The Elite Home Page**. 1984. Available: <<http://www.iancgbell.clara.net/elite/>>. Citation on page 47.

BLESZINSKI, J. S. C. **Unreal**. 1998. Microsoft Windows; Mac OS. Available: <<https://web.archive.org/web/20010331080920/http://www.unreal.com/index2.html>>. Citation on page 49.

BORRELL, R.; CAJAS, J.; MIRA, D.; TAHA, A.; KORIC, S.; VÁZQUEZ, M.; HOUZEAUX, G. Parallel mesh partitioning based on space filling curves. **Computers & Fluids**, 2018. ISSN 0045-7930. Available: <<http://www.sciencedirect.com/science/article/pii/S0045793018300446>>. Citation on page 58.

BURSTEDDE, C.; HOLKE, J. A tetrahedral space-filling curve for nonconforming adaptive meshes. **SIAM Journal on Scientific Computing**, v. 38, n. 5, p. C471–C503, 2016. Available: <<https://doi.org/10.1137/15M1040049>>. Citation on page 58.

CALVIN, A.; MICHAEL, N. The quest in a generated world. In: **DiGRA &#3907 - Proceedings of the 2007 DiGRA International Conference: Situated Play**. The University of Tokyo, 2007. ISBN ISSN 2342-9666. Available: <<http://www.digra.org/wp-content/uploads/digital-library/07311.20228.pdf>>. Citations on pages 27 and 54.

CORMEN, T. H. Section 22.3: Depth-first search. In: _____. **Introduction to algorithms**. [S.l.]: MIT Press, 2008. p. 540–549. Citations on pages 19 and 42.

CRYTEK. **CRYENGINE | The complete solution for next generation game development by Crytek**. 2018. Available: <<https://www.cryengine.com/>>. Citation on page 51.

CUI, X.; SHI, H. A*-based pathfinding in modern computer games. **International Journal of Computer Science and Network Security**, v. 11, n. 1, p. 125;130, 2011. Available: http://paper.ijcsns.org/07_book/201101/20110119.pdf. Citation on page 40.

DAHLSSKOG, S.; TOGELIUS, J. A multi-level level generator. In: **2014 IEEE Conference on Computational Intelligence and Games**. [S.l.: s.n.], 2014. p. 1–8. ISSN 2325-4270. Citation on page 55.

DEALS, T. **This engine is dominating the gaming industry right now**. 2016. Available: <https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/>. Citation on page 51.

DESIGN, A. I. **Rogue for Amiga**. 1984. Available: <https://www.mobygames.com/game/rogue>. Citation on page 47.

DEVELOPMENTS, F. **Elite: Dangerous**. 2014. Available: <https://www.elitedangerous.com/>. Citation on page 47.

DOLONIUS, D.; SINTORN, E.; KÄMPE, V.; ASSARSSON, U. Compressing color data for voxelized surface geometry. **IEEE Transactions on Visualization and Computer Graphics**, PP, n. 99, p. 1–1, 2018. ISSN 1077-2626. Citation on page 58.

DORMANS, J. Adventures in level design: Generating missions and spaces for action adventure games. In: **Proceedings of the 2010 Workshop on Procedural Content Generation in Games**. New York, NY, USA: ACM, 2010. (PCGames '10), p. 1:1–1:8. ISBN 978-1-4503-0023-0. Available: <http://doi.acm.org/10.1145/1814256.1814257>. Citations on pages 49 and 57.

_____. Level design as model transformation: A strategy for automated content generation. In: **Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games**. New York, NY, USA: ACM, 2011. (PCGames '11), p. 2:1–2:8. ISBN 978-1-4503-0872-4. Available: <http://doi.acm.org/10.1145/2000919.2000921>. Citation on page 57.

DORMANS, J.; BAKKES, S. Generating missions and spaces for adaptable play experiences. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 3, n. 3, p. 216–228, Sept 2011. ISSN 1943-068X. Citations on pages 27, 34, and 57.

EIBEN, A. E.; SMITH, J. E. **Introduction to Evolutionary Computing**. [S.l.]: SpringerVerlag, 2003. ISBN 3540401849. Citations on pages 15, 19, 38, and 39.

ESCAYG, K. M. S. **Uncharted: The Lost Legacy**. 2017. PlayStation 4. Available: <https://www.unchartedthegame.com/en-us/games/uncharted-the-lost-legacy/>. Citation on page 34.

FERREIRA, L. N. **Uma abordagem evolutiva para geração procedural de níveis em jogos de quebra-cabeças baseados em física**. Master's Thesis (Master's Thesis) — Universidade de São Paulo - Campus São Carlos, São Carlos, São Paulo, Brasil, 2015. Citation on page 56.

FERREIRA, L. N.; TOLEDO, C. Tanager: A generator of feasible and engaging levels for angry birds. **IEEE Transactions on Computational Intelligence and AI in Games**, IEEE, 2017. Citation on page 56.

FREDMAN, M.; TARJAN, R. Fibonacci heaps and their uses in improved network optimization algorithms. **25th Annual Symposium on Foundations of Computer Science, 1984.**, 1984. Citation on page 40.

FRITZ, B.; PHAM, A. **Star Wars: The Old Republic — the story behind a galactic gamble**. 2012. Available: <<http://herocomplex.latimes.com/games/star-wars-the-old-republic-the-story-behind-a-galactic-gamble/#/0>>. Citation on page 28.

FUJIBAYASHI, H. **The Legend of Zelda: The Minish Cap**. 2004. Game Boy Advance, Nintendo 3DS, Nintendo WiiU. Available: <<https://www.nintendo.com/games/detail/Uq4mHxYWIHPUITIOd967IkJ11vpCuYak>>. Citation on page 36.

_____. **The Legend of Zelda: Breath of the Wild**. 2017. Nintendo Switch; Wii U. Available: <<https://www.zelda.com/breath-of-the-wild/>>. Citation on page 34.

GAMES, E. **Game Engine Technology by Unreal**. 2018. Available: <<https://www.unrealengine.com/en-US/features>>. Citation on page 50.

GAMES, F. **Civilization VI**. 2016. Available: <<https://www.civilization.com/>>. Citation on page 47.

GAMES, H. **No Man's Sky**. 2016. Available: <<http://www.no-mans-sky.com/about/>>. Citation on page 47.

GARDNER, M. The fantastic combinations of john conway's new solitaire game " life". **Sci. Am**, v. 223, n. 10, 1970. Citation on page 46.

GREGORY, J. **Game Engine Architecture, Second Edition**. CRC Press, 2014. ISBN 9781466560062. Available: <<https://books.google.com.br/books?id=LILSBQAAQBAJ>>. Citation on page 49.

HALL, J. R. S. P. A. M. S. G. T. **Doom**. 1993. MS-DOS. Available: <<https://www.mobygames.com/game/doom>>. Citation on page 49.

HENDRIKX, M.; MEIJER, S.; VELDEN, J. V. D.; IOSUP, A. Procedural content generation for games: A survey. In: . New York, NY, USA: ACM, 2013. v. 9, n. 1, p. 1:1–1:22. ISSN 1551-6857. Available: <<http://doi.acm.org/10.1145/2422956.2422957>>. Citation on page 47.

HORN, B.; DAHLISKOG, S.; SHAKER, N.; SMITH, G.; TOGELIUS, J. A comparative evaluation of procedural level generators in the mario ai framework. In: **FDG**. [S.l.: s.n.], 2014. Citations on pages 87 and 88.

HORSWILL, I.; FOGED, L. Fast procedural level population with playability constraints. In: **Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. AAAI Press, 2012. (AIIDE'12), p. 20–25. Available: <<http://dl.acm.org/citation.cfm?id=3014629.3014634>>. Citations on pages 55 and 105.

HOSOKAWA, J. L. M. T. **Metroid: Samus Returns**. 2017. Nintendo 3DS. Available: <<https://metroidsamusreturns.nintendo.com/>>. Citation on page 34.

ILACHINSKI, A. **Cellular Automata: A Discrete Universe**. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2001. ISBN 981238183X. Citations on pages 44 and 45.

IOSUP, A. Poggi: Generating puzzle instances for online games on grid infrastructures. In: . Chichester, UK: John Wiley and Sons Ltd., 2011. v. 23, n. 2, p. 158–171. ISSN 1532-0626. Available: <<http://dx.doi.org/10.1002/cpe.1638>>. Citation on page 47.

JACQUAYS, G. D. T. W. P. **Quake III Arena**. 1999. Microsoft Windows; Mac OSX; Linux; Dreamcast; PlayStation 2; Xbox 360; iOS. Available: <https://web.archive.org/web/20020718214708/http://quake3arena.com:80/>. Citation on page 49.

JASRASARIA, D.; PYZER-KNAPP, E. O.; RAPPOPORT, D.; ASPURU-GUZZIK, A. Space-Filling Curves as a Novel Crystal Structure Representation for Machine Learning Models. **ArXiv e-prints**, Aug. 2016. Citations on pages 42 and 58.

JOHNSON, L.; YANNAKAKIS, G. N.; TOGELIUS, J. Cellular automata for real-time generation of infinite cave levels. In: **Proceedings of the 2010 Workshop on Procedural Content Generation in Games**. New York, NY, USA: ACM, 2010. (PCGames '10), p. 10:1–10:4. ISBN 978-1-4503-0023-0. Available: <http://doi.acm.org/10.1145/1814256.1814266>. Citations on pages 46 and 54.

JONGE, M. de. **Horizon Zero Dawn**. 2017. PlayStation 4. Available: <https://www.guerrilla-games.com/play/horizon>. Citation on page 34.

KHALIFA, A.; PEREZ-LIEBANA, D.; LUCAS, S. M.; TOGELIUS, J. General video game level generation. In: **Proceedings of the Genetic and Evolutionary Computation Conference 2016**. New York, NY, USA: ACM, 2016. (GECCO '16), p. 253–259. ISBN 978-1-4503-4206-3. Available: <http://doi.acm.org/10.1145/2908812.2908920>. Citations on pages 29 and 56.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. **Artif. Intell.**, v. 27, p. 97–109, 1985. Citation on page 42.

KOZLOVA, A.; BROWN, J. A.; READING, E. Examination of representational expression in maze generation algorithms. In: **2015 IEEE Conference on Computational Intelligence and Games (CIG)**. [S.l.: s.n.], 2015. p. 532–533. ISSN 2325-4289. Citation on page 42.

KUCHERA, B. **Report: 7,672 games were released on Steam in 2017**. 2018. Available: <https://www.polygon.com/2018/1/10/16873446/steam-release-dates-2017>. Citation on page 28.

LERA, D.; SERGEYEV, Y. D. Deterministic global optimization using space-filling curves and multiple estimates of lipschitz and hölder constants. **Communications in Nonlinear Science and Numerical Simulation**, v. 23, n. 1, p. 328 – 342, 2015. ISSN 1007-5704. Available: <http://www.sciencedirect.com/science/article/pii/S1007570414005437>. Citation on page 43.

LESTER, P. A* **Pathfinding for Beginners**. 2005. Available: <http://homepages.abdn.ac.uk/f.guerin/pages/teaching/CS1013/practicals/aStarTutorial.htm>. Citations on pages 15 and 41.

LIAPIS, A. Multi-segment evolution of dungeon game levels. In: **Proceedings of the Genetic and Evolutionary Computation Conference**. New York, NY, USA: ACM, 2017. (GECCO '17), p. 203–210. ISBN 978-1-4503-4920-8. Available: <http://doi.acm.org/10.1145/3071178.3071180>. Citations on pages 29, 34, 57, and 58.

LIAPIS, A.; YANNAKAKIS, G. N.; TOGELIUS, J. Sentient sketchbook: Computer-aided game level authoring. In: **FDG**. [S.l.: s.n.], 2013. Citations on pages 29 and 55.

_____. Towards a generic method of evaluating game levels. In: **Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. AAAI Press, 2014. (AIIDE'13), p. 30–36. ISBN 1577356071, 978-1-57735-607-3. Available: <http://dl.acm.org/citation.cfm?id=3014712.3014718>. Citation on page 88.

LINDEN, R. van der; LOPES, R.; BIDARRA, R. Designing procedurally generated levels. In: . [S.l.: s.n.], 2013. Citations on pages 49 and 57.

LUBAN, P. **Designing and Integrating Puzzles in Action-Adventure Games**. 2002. Available: <https://www.gamasutra.com/view/feature/131326/designing_and_integrating_puzzles_.php>. Citation on page 33.

MACEDO, Y. P. A.; CHAIMOWICZ, L. Improving procedural 2d map generation based on multi-layered cellular automata and hilbert curves. Versão preliminar na página: <https://www.sbgames.org/sbgames2017/papers/COMPUTACAO/FULL_PAPERS/175350_2_versao_preliminar.pdf>. 2017. Citations on pages 29, 31, 43, 46, and 58.

MAITI, S.; GHOSH, S.; CHOWDHURY, D. R. On the security of designing a cellular automata based stream cipher. In: SPRINGER. **Australasian Conference on Information Security and Privacy**. [S.l.], 2017. p. 406–413. Citation on page 46.

MARIÑO, J.; LELIS, L. A computational model based on symmetry for generating visually pleasing maps of platform games. In: . [s.n.], 2016. Available: <<https://www.aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/14016>>. Citations on pages 88 and 99.

MATHEW, G. E. Direction based heuristic for pathfinding in video games. **Procedia Computer Science**, v. 47, p. 262–271, 2015. Citation on page 40.

MCDONALD, E. **The Global Games Market 2017**. 2017. Available: <<https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/>>. Citation on page 27.

MCGUINNESS, C.; ASHLOCK, D. Decomposing the level generation problem with tiles. In: **2011 IEEE Congress of Evolutionary Computation (CEC)**. [S.l.: s.n.], 2011. p. 849–856. ISSN 1089-778X. Citations on pages 29 and 55.

MCMILLEN, E. **The Binding of Isaac: Rebirth**. 2014. Microsoft Windows, OS X, Linux, PlayStation 4, PlayStation Vita, Wii U, New Nintendo 3DS, Xbox One, iOS, Nintendo Switch. Available: <<http://bindingofisaac.com/>>. Citation on page 48.

MCMILLEN, E.; HIMSL, F. **The Binding of Isaac**. 2011. Microsoft Windows, OS X, Linux. Available: <<http://bindingofisaac.com/>>. Citation on page 47.

MERRICK, K. E.; ISAACS, A.; BARLOW, M.; GU, N. A shape grammar approach to computational creativity and procedural content generation in massively multiplayer online role playing games. **Entertainment Computing**, v. 4, n. 2, p. 115 – 130, 2013. ISSN 1875-9521. Available: <<http://www.sciencedirect.com/science/article/pii/S1875952112000213>>. Citations on pages 49 and 57.

MILLINGTON, I.; FUNGE, J. **Artificial Intelligence for Games, Second Edition**. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. 549–553 p. ISBN 0123747317, 9780123747310. Citation on page 58.

MIYAMOTO, S.; TEZUKA, T. **The Legend of Zelda Series**. 1986. Available: <<http://www.zelda.com/>>. Citation on page 27.

MOTOKURA, K. **Super Mario Odyssey**. 2017. Nintendo Switch. Available: <<https://www.nintendo.com/games/detail/super-mario-odyssey-switch>>. Citation on page 34.

NØRMAK, K. **Example of recursion: Hilbert Curves**. 2013. Available: <http://people.cs.aau.dk/~normark/prog3-03/html/notes/fu-intr-2_themes-hilbert-sec.html>. Citations on pages 15, 43, and 45.

OUNI, T.; LASSOUED, A.; ABID, M. Lossless image compression using gradient based space filling curves (g-sfc). **Signal, Image and Video Processing**, v. 9, n. 2, p. 277–293, Feb 2015. ISSN 1863-1711. Available: <<https://doi.org/10.1007/s11760-013-0435-4>>. Citation on page 43.

PARDO-IGÚZQUIZA, E.; COLLADOS-LARA, A.; PULIDO-VELAZQUEZ, D. Future effects of climate change on the dynamics of the sierra nevada snowpack: conclusions from cellular automata models. In: **Groundwater and Global Change in the Western Mediterranean Area**. [S.l.]: Springer, 2018. p. 269–275. Citation on page 46.

PEDERSEN, R. **Game Design Foundations**. Jones & Bartlett Learning, 2009. 291 p. ISBN 9781449663926. Available: <<https://books.google.com.br/books?id=0flChIjb9IIC>>. Citation on page 49.

PEREIRA, L. T.; PRADO, P. V. S.; TOLEDO, C. Evolving dungeon maps with locked door missions. In: **2018 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2018. p. 1–8. Citations on pages 29, 69, and 103.

PEREIRA, L. T.; TOLEDO, C. F. M. Speeding up search-based algorithms for level generation in physics-based puzzle games. **International Journal on Artificial Intelligence Tools**, v. 26, n. 05, p. 1760019, 2017. Available: <<http://www.worldscientific.com/doi/abs/10.1142/S0218213017600193>>. Citation on page 56.

PLURALSIGHT. **Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose?** 2015. Available: <<https://www.pluralsight.com/blog/film-games/unity-udk-cryengine-game-engine-choose>>. Citation on page 51.

POCKETGAMER.BIZ. **App Store Metrics**. 2017. Available: <<http://www.pocketgamer.biz/metrics/app-store/submissions/>>. Citation on page 28.

QIN, Y.; LU, H.; XU, Y.; WANG, H. Saliency detection via cellular automata. In: IEEE. **Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on**. [S.l.], 2015. p. 110–119. Citation on page 46.

RASHID, Z. **3 Game Engines Every Indie Developer should try**. 2016. Available: <<https://www.linkedin.com/pulse/3-game-engines-every-indie-developer-should-try-zamilur-rashid-csm/>>. Citation on page 51.

ROBINETT, W. **Adventure**. 1979. Atari 2600. Available: <<http://www.warrenrobinett.com/adventure/>>. Citation on page 34.

ROGERS, S. **Level Up! The Guide to Great Video Game Design**. 2nd. ed. [S.l.]: Wiley Publishing, 2014. ISBN 1118877160, 9781118877166. Citation on page 97.

ROLLINGS, A. E. A. **Andrew Rollings and Ernest Adams on Game Design**. [S.l.]: New Riders Publishing, 2003. 446 p. ISBN 1-59273-001-9. Citation on page 33.

RUELA, A.; DELGADO, K. Evolving lock-and-key puzzles based on nonlinear player progression and level exploration. In: . [S.l.: s.n.], 2018. Citation on page 30.

RYAN, M.-L. Beyond myth and metaphor: Narrative in digital media. **Poetics Today**, v. 23, n. 4, p. 581–609, 2002. Citation on page 33.

SAGAN, H. **Space-filling curves**. [S.l.]: Springer, 2012. Citation on page 42.

SANTAMARÍA-IBIRIKA, A.; CANTERO, X.; HUERTA, S.; SANTOS, I.; BRINGAS, P. G. Procedural playable cave systems based on voronoi diagram and delaunay triangulation. In: **2014 International Conference on Cyberworlds**. [S.l.: s.n.], 2014. p. 15–22. Citation on page 54.

SCHIFF, J. L. **Cellular Automata: A Discrete View of the World (Wiley Series in Discrete Mathematics & Optimization)**. [S.l.: s.n.], 2007. ISBN 047016879X, 9780470168790. Citation on page 44.

SHAKER, N.; TOGELIUS, J.; NELSON, M. J. **Procedural content generation in games**. [S.l.]: Springer, 2016. 42–45 p. Citation on page 46.

SMITH, A. M.; ANDERSEN, E.; MATEAS, M.; POPOVIĆ, Z. A case study of expressively constrainable level design automation tools for a puzzle game. In: **Proceedings of the International Conference on the Foundations of Digital Games**. New York, NY, USA: ACM, 2012. (FDG '12), p. 156–163. ISBN 978-1-4503-1333-9. Available: <<http://doi.acm.org/10.1145/2282338.2282370>>. Citation on page 55.

SMITH, G.; WHITEHEAD, J. Analyzing the expressive range of a level generator. In: **Proceedings of the 2010 Workshop on Procedural Content Generation in Games**. New York, NY, USA: ACM, 2010. (PCGames '10), p. 4:1–4:7. ISBN 978-1-4503-0023-0. Available: <<http://doi.acm.org/10.1145/1814256.1814260>>. Citation on page 88.

SNODGRASS, S.; ONTAÑÓN, S. Controllable procedural content generation via constrained multi-dimensional markov chain sampling. In: **Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence**. AAAI Press, 2016. (IJCAI'16), p. 780–786. ISBN 978-1-57735-770-4. Available: <<http://dl.acm.org/citation.cfm?id=3060621.3060730>>. Citation on page 54.

STEPHEN, R. **Draw a Hilbert curve fractal in C#**. 2014. Available: <<http://csharpHelper.com/blog/2014/07/draw-a-hilbert-curve-fractal-in-c/>>. Citations on pages 19 and 44.

SU, T.; WANG, W.; LV, Z.; WU, W.; LI, X. Rapid delaunay triangulation for randomly distributed point cloud data using adaptive hilbert curve. **Computers & Graphics**, v. 54, p. 65 – 74, 2016. ISSN 0097-8493. Special Issue on CAD/Graphics 2015. Available: <<http://www.sciencedirect.com/science/article/pii/S0097849315001223>>. Citation on page 43.

SUMMERVILLE, A.; MATEAS, M. Sampling hyrule: Multi-technique probabilistic level generation for action role playing games. In: **2015 AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [s.n.], 2015. Available: <<https://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11570>>. Citations on pages 34 and 53.

SUMMERVILLE, A.; nO, J. R. H. M.; SNODGRASS, S.; nÓN, S. O.; LELIS, L. H. S. Understanding mario: An evaluation of design metrics for platformers. In: **Proceedings of the 12th International Conference on the Foundations of Digital Games**. New York, NY, USA: ACM, 2017. (FDG '17), p. 8:1–8:10. ISBN 978-1-4503-5319-9. Available: <<http://doi.acm.org/10.1145/3102071.3102080>>. Citation on page 88.

SUMMERVILLE, A. J.; BEHROOZ, M.; MATEAS, M.; JHALA, A. The learning of zelda: Data-driven learning of level topology. In: **Proceedings of the 10th International Conference on the Foundations of Digital Games**. [S.l.: s.n.], 2015. Citations on pages 34 and 53.

SYEED, A.; RASHID, T. U.; DHALI, M. A.; FARID, H. M.; SADIK, A. M. A comprehensive and comparative study of maze-solving techniques by implementing graph theory. In: **Artificial Intelligence and Computational Intelligence, International Conference on(AICI)**. [s.n.], 2010. v. 01, p. 52–56. Available: <doi.ieeecomputersociety.org/10.1109/AICI.2010.18>. Citation on page 42.

TARO, Y. **Nier: Automata**. 2017. PlayStation 4; Microsoft Windows. Available: <<https://www.niergame.com/>>. Citation on page 34.

UNITY. **Fast Facts**. 2018. Available: <<https://unity3d.com/pt/public-relations>>. Citation on page 29.

UNITY3D. **Unite 2014 - Generating Procedural Dungeons in Galak Z**. 2014. Available: <<https://www.youtube.com/watch?v=ySTpjT6JYFU>>. Citations on pages 31 and 58.

VALTCHANOV, V.; BROWN, J. A. Evolving dungeon crawler levels with relative placement. In: **Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering**. New York, NY, USA: ACM, 2012. (C3S2E '12), p. 27–35. ISBN 978-1-4503-1084-0. Available: <<http://doi.acm.org/10.1145/2347583.2347587>>. Citations on pages 29, 34, and 56.

WINTER, J. C. F. D.; DODOU, D. Five-point likert items: t test versus mann-whitney-wilcoxon. **Practical Assessment, Research & Evaluation**, p. 11, 2010. Citation on page 85.

WOLFRAM, S. Statistical mechanics of cellular automata. **Rev. Mod. Phys.**, American Physical Society, v. 55, p. 601–644, Jul 1983. Available: <<https://link.aps.org/doi/10.1103/RevModPhys.55.601>>. Citation on page 44.

_____. **A new kind of science**. [S.l.]: Wolfram Media, 2002. Citation on page 45.

WRIGHT, W. **SimCity**. 1989. Available: <<https://www.ea.com/pt-br/games/simcity>>. Citation on page 58.

XU, Q.; TREMBLAY, J.; VERBRUGGE, C. Generative methods for guard and camera placement in stealth games. In: **Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. AAAI Press, 2014. (AIIDE'14), p. 87–93. ISBN 1577356810, 978-1-57735-681-3. Available: <<http://dl.acm.org/citation.cfm?id=3014752.3014766>>. Citation on page 54.

DUNGEONS AVAILABLE IN THE SURVEY

Figure 39 – The Demon - Dungeon 5

Figure 40 – The Demon EA Generated Counterpart- Dungeon 1

Figure 41 – The Lizard - Dungeon 5

Figure 42 – The Demon EA Generated Counterpart- Dungeon 6

Figure 43 – Spirit's Grave - Dungeon 4

Figure 44 – Spirit's Grave EA Generated Counterpart- Dungeon 3

DUNGEONS AVAILABLE IN PROCOG

Figure 45 – Lizard Map

Figure 46 – Lizard Map Counterpart Created by the CEA

Figure 47 – Moon Map

Figure 48 – Moon Map Counterpart Created by the CEA

Figure 49 – Dragon Map

Figure 50 – Dragon Map Counterpart Created by the CEA

Figure 51 – Eagle Map

Figure 52 – Eagle Map Counterpart Created by the CEA

Figure 53 – Lion Map

Figure 54 – Lion Map Counterpart Created by the CEA

Figure 55 – Snake Map

Figure 56 – Snake Map Counterpart Created by the CEA