UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Automatic generation and evaluation of platform games

Diogo Alexandre Da Silva Soares

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor Luís Manuel Ferreira Fernandes Moniz

2023

Acknowledgments

This project has been an incredible journey for me, it gave me the opportunity to explore and learn so much about game design and content generation. I am grateful for the opportunity to have worked on this project and to have been able to share my work with other people. But this project would not have been possible without the help and support of some extraordinary people in my life, I would like to express my gratitude to them.

First and foremost, I would like to express my gratitude to Professor Luís Moniz for making this thesis topic available, as well as for his patience, encouragement, project guidance, and constructive criticism.

I am also deeply grateful to my family for supporting me throughout this project. I am especially grateful to my parents and brother for their unconditional love, support, and guidance throughout my life. They have always believed in me and encouraged me to keep pushing for my dreams and never give up. Their love, advice, and wisdom have been invaluable to me, and I will be forever grateful.

I would also like to give a very special thanks to Sofia Pereira, for being there for me through all the highs and lows during this project. She has been my rock and my biggest supporter, and I am so grateful to have her in my life. Without her unwavering encouragement and love, this project would have been much more difficult.

I would also like to thank all my friends for their unrelenting support, encouragement and humour. Especially David Pereira, Miguel Marcelino, Vasco Castro, Samuel Ferreira and Andreia Batista. They made the last five years of college a lot more enjoyable, fun and interesting. The experiences we shared together will live with me forever.

I am also quite thankful to my work colleagues for all the support and time they have provided me throughout this project. Thank you for being such an amazing team!

Finally, I would like to thank all those who provided me with materials and resources that enabled me to complete my work.

This work is dedicated to my father and mother, whose love and support have been an inspiration throughout my life. They have always been there for me and I am so thankful for their guidance and encouragement. I would not be where I am today without them. Thank you for always believing in me, even when I doubted myself. I love you both dearly.

Abstract

A two-dimensional platformer game can be characterized by levels consisting of uneven terrain that requires jumping and climbing to traverse. This study explores the use of graph grammars as a rule system for procedurally generating and evaluating these kinds of games. It follows the technique of separating the objectives or mission of a level from the layout or space of a level into two different domains, where a mission is created following the syntax of a graph grammar and the space is generated based on a mission. During this generation process, both the missions and spaces generated are evaluated directly with the use of search-based evaluation functions and indirectly with an elementary player AI simulation. To evaluate the generator's capabilities, a simple prototype game using this generation technique was created along with three separate mission grammars. The output levels generated with these grammars were analyzed based on four data properties: linearity, leniency, density, and candidate feasibility. The grammar that showed the most favourable results in terms of feasibility was then tested with a small group of human players to validate the playability of the levels created by the generator. The obtained data showed that the generator is capable of generating playable and engaging levels, but that generating only the missions with a single grammar limits the possibilities of the content generated and makes the position of structures more difficult to validate, indicating that using various targeted grammars in tiny amounts might produce better outcomes.

Keywords: Procedural Generation, Platform Games, Graph Grammar, Level Evaluation, Replayability

Resumo

Com o crescimento da indústria de jogos eletrónicos, cada vez mais as empresas estão a investir na criação de jogos, esta competição causa um aumento tanto na escala, como na complexidade e no custo dos jogos, o que tem levado cada vez mais à necessidade de automatizar certos aspetos do processo de criação de jogos. Uma metodologia popular para conseguir essa automatização, é chamada de geração de conteúdo procedimental (GCP), sendo definida como uma forma algorítmica de usar software para criar conteúdos de um jogo com muito pouca intervenção humana. Existem inúmeras abordagens para criar e aplicar uma solução de GCP a um problema, muitas das quais são exclusivas do jogo para o qual foram planeadas, no entanto, existem padrões comuns que formam abordagens mais generalizadas. Uma destas abordagens generalizadas, é denominada de ”abordagem baseada em pesquisa”, e consiste em usar um algoritmo de pesquisa estocástica para pesquisar o conteúdo que vai sendo gerado, avaliando o conteúdo com a ajuda de uma ou várias funções de avaliação, por forma a descobrir qual o conteúdo gerado com as qualidades mais desejadas. Outra abordagem de GCP consiste em usar uma coleção de regras de produção normalmente na forma de uma ou mais gramáticas formais para instruir o programa sobre como gerar conteúdos para o jogo seguindo a sintaxe das linguagens estabelecidas.

Este estudo explora o uso desta segunda abordagem para gerar processualmente jogos de plataforma em 2D, usando um tipo específico de gramática chamada gramática de grafos como o sistema de regras. Ao mesmo tempo, também explora o uso de funções de avaliação para avaliar e validar se os níveis gerados são possíveis de completar por um jogador. Para tal, foi desenvolvido um gerador de níveis, usando uma versão adaptada para níveis de jogos de plataformas, a partir de uma técnica proposta por Joris Dormans, usada para gerar níveis de jogos de ação e aventura. Nesta técnica, a geração de um nível é separada em dois domínios diferentes: os objetivos ou missão e o layout ou espaço do nivel. Neste contexto, uma missão é um mapa representado por um grafo, onde os vértices representam os objetivos ou elementos do mapa e as arestas representam as ligações entre estes elementos, este mapa de missão é criado aplicando as várias regras de uma gramática de grafos. Além disso, o espaço é também um mapa representado por um grafo, onde os vértices são as estruturas que compõem o nível e as arestas são as ligações entre estas estruturas. Como as estruturas estão associadas aos elementos de um mapa de missão, a geração do mapa de espaço é feita com base nessa missão, por forma a aumentar as hipóteses de gerar um nível válido e que satisfaça os parâmetros escolhidos pelo programador, durante a geração de um nível são criados múltiplos mapas de missão. Cada mapa começa como um grafo simples apenas com um vértice inicial, depois é transformado aplicando regras da gramática de grafos até não ser mais possível aplicar regras. Para facilitar este processo, foi desenvolvido um sistema de reescrita de grafos. Este tipo de sistema é responsável por fornecer uma forma de criar regras de produção, reconhecer onde uma regra de produção pode ser aplicada, escolher uma regra com base em critérios pré-definidos e, por fim, aplicar a regra de produção escolhida alterando o grafo de acordo com a regra. Após a construção de todos os mapas de missão terminar, estes são avaliados e classificados diretamente usando as funções de avaliação de cada parâmetro pré-definido pelo programador. No caso do jogo protótipo criado para este estudo os parâmetros usados foram; o tamanho do nível, a verticalidade do mapa de missão e o número de inimigos e de moedas num nível. Os mapas de missão com melhor classificação são depois transformados em mapas de espaço. A criação de um mapa de espaço é feita percorrendo o grafo de uma missão seguindo um algoritmo de busca em largura. A medida que o grafo vai sendo percorrido, cada vértice V_m pertencente ao mapa de missão é convertido numa estrutura que seja associada a este. Depois cada vértice V_ad adjacente a V_m, que pertença às arestas que partem de V_m, também é convertido numa estrutura e a sua posição relativamente à posição V_m é calculada e validada. Esta validação passa por verificar se a estrutura E_ad associada ao vértice V_ad quando é posicionada não colide com outra estrutura E_b que já esteja no mapa. Se houver uma colisão, então a posição da estrutura E_ad é recalculada, mas desta vez relativamente a E_b. Este processo é repetido até ser encontrado um local para posicionar a estrutura ou até que o número de tentativas seja esgotado. Caso se esgote o número de tentativas, o vértice V_ad e a sua estrutura associada E_ad são descartados e os filhos de V_ad passam a pertencer ao vértice associado à estrutura E_b. Todas as estruturas criadas formam os vértices do grafo que representam o mapa de espaço. Após todos os mapas de missão estarem convertidos em mapas de espaço, estes também são avaliados e classificados segundo os seguintes critérios: a distância euclidiana entre o início e o fim do nível, o número de colisões no posicionamento de estruturas, o tamanho do caminho mais pequeno entre o início e o fim do nível e por último se o nível é viável ou não. Esta última verificação é feita indiretamente com uma simulação elementar de um jogador usando inteligência artificial (IA). Posteriormente, o mapa de espaço com melhor classificação é então transformado num nível. Esta transformação é dependente da maneira como a lógica do jogo funciona.

Para ajudar a avaliar as capacidades do gerador criado, foi implementado um jogo protótipo simples e em conjunto também foram criadas três gramáticas de missão diferentes. Foram feitos dois testes para avaliar o gerador. O primeiro teste consistiu em usar o gerador para gerar 100 níveis com cada uma das três gramáticas, estes níveis foram então analisados com base em quatro propriedades: a linearidade do terreno, a leniência ou o nível de facilidade de completar o nível, a densidade de colisões entre estruturas durante a geração do nível e por último a quantidade de níveis candidatos viáveis criados durante o processo de geração. No segundo teste, o jogo protótipo foi disponibilizado publicamente onde foi testado por um grupo pequeno de jogadores humanos para validar a jogabilidade dos níveis criados. Nesta versão pública todos os níveis são gerados com base na gramática que apresentou os resultados mais favoráveis em termos de viabilidade no teste anterior. Os dados obtidos mostraram que o gerador é capaz de gerar níveis possiveis de jogar e cativantes, com a maioria dos participantes a dar um feedback positivo. No entanto, os resultados também mostraram que, na sua simplicidade, o gerador tem dificuldade em lidar com gramáticas onde o desenho das regras é mais complexo. A simplicidade da simulação com o jogador IA para determinar a viabilidade dos níveis também afetou a precisão desta validação, classificando uma grande percentagem de níveis como não viáveis quando de facto eram possíveis de serem completados, mostrando que o sistema de validação precisa de ser melhorado. Por último, os dados mostraram que gerar as missões apenas com uma única gramática limita as possibilidades e variedade do conteúdo gerado e também dificulta a validação do posicionamento das estruturas que compõem o nível, indicando que o uso de várias gramáticas direcionadas para gerar conteúdo mais específico e aplicadas em pequenas quantidades pode produzir melhores resultados.

Palavras-chave: Geração procedimental, Jogos de Plataforma, Gramática de Grafos, Avaliação de Nível, Repetibilidade

Contents

List of Figures

List of Abbreviations

AI	artificial intelligence
PCG	procedural content generation
2D	two-dimensional
GCG	generic content generation
L-system	Lindenmayer system
LHS	left-hand side
RHS	right-hand side
AGG	Attributed Graph Grammar
GGL	Graph Grammar Library
PROGRES	PROgrammed Graph REwriting Systems
DPO	double-pushout
SPO	single-pushout
SMB	Super Mario Bros.
NPC	non-player character
BFS	breadth-first search
GUI	graphical user interface

Chapter 1

Introduction

With the growth of the video gaming industry, more and more companies are investing in the making of games that will stay relevant for a long time or captivate players to buy their next games. This competition has resulted in an increase in game scale, complexity and cost, to the point that big companies with larger teams of developers expend many years developing a game. The amount of effort and human resources needed to add great quantities of content to an already large game is enormous compared to the extra bit of experience it adds for the player, this was pointed out by legendary game designer Will Wright in his talk *"The future of content"* at the 2005 Game Developers Conference[1]. This naturally leads to the necessity of automating certain aspects of the game creation process. For example, once a game's general level feel and design are specified, a level designer will follow the set of guidelines or rules defined to create new levels for the game; this process can be automated. A popular methodology for achieving this is called procedural content generation (PCG), which is defined as an algorithmic way of using software to create game content with limited human interaction [2], basically the game developer writes an algorithm or set of comprehensive instructions to instruct the game on how to generate content. Following this algorithm, the game will then generate something completely unique.

Games that rely on replayability like the original Rogue and the Rogue-like genre, use PCG to create new levels for the player to have a fresh experience each time they play. A good example is a game called *Spelunky*[3] in which designer Derek Yu implemented a system where each level is divided into a grid of rooms, then a path is drawn from the entrance at the top to the exit at the bottom, after that each room is picked from a set of room templates. The template is chosen based on where the room lies on the path from the entrance to the exit. On each template there are randomized chunks that give some variety to the level and then the level is completed by adding the extra bits like treasure and traps, a more detailed explanation of this algorithm can be found in chapter 3 of Shaker, et al. book *"Procedural content generation in games"*[4]. With this algorithm, a nearly infinite number of levels can be generated that are different enough to be perceived as new experiences, but keeping the core rules and feel of the game. This is the power PCG can have when combined with the creative minds of designers. And *Spelunky* is not a unique example, many other games use this technique as a core design of their game such as *RimWorld*, *Terraria*, *Starbound* and many others, or as a way to generate more repetitive parts of their design, for example, the use of the software *SpeedTree* to generate the vegetation in the game world. Apart from saving companies money and time with automation of certain parts of the creation process, PCG can enable smaller teams of developers to create content-rich games that can compete with the big titles, not to mention serve as inspiration for human developers to create new and different designs. Furthermore, PCG can be combined with player modelling techniques to monitor player responses to game elements and adapt levels to those responses, allowing the creation of infinitely replayable adaptive games[5].

But in order to use PCG the developer needs to create an algorithm or set of instructions to generate the desired content, but what if it was possible to alleviate that process and provide a framework that generated level layouts that followed the syntax of a predefined grammar, a developer could build said grammar by defining a set of production rules that uses an alphabet of symbols that represent the content in a level and specify where and when this content should appear. The main goal of this project is to test this generative grammar approach by building a PCG framework capable of generating two-dimensional (2D) platformer levels using an input predefined grammar, while also evaluating the content created based on its playability and a set of possible parameters defined by the developer. The next section will go into more detail about the objectives of this project.

1.1 Objectives

The main objective of this research is to find a method for generating levels in 2D platformer games with minimal human intervention, by implementing a rule-based system where the developers can define the rules. Additionally, the generator has to be capable of evaluating the content generated to ensure that levels are playable. The approach used to build this generator, follows the method used by Joris Dormans in his study [6] where he divided the creation of levels for action-adventure games into two steps: the mission representing the objectives, items or player actions needed to complete the level and the space which represents the level geometry.

This study experiments if the same can be done in a side-scroller 2D platformer game, by exploring the generative grammars approach and creating a prototype software capable of generating levels given a predefined grammar. The level evaluation uses a search-based approach where multiple candidates are created at each stage of the generation process and evaluation functions are applied to select the best candidates for the next step. As such, the following objectives are set for this project:

- Defining an alphabet of symbols to be used by the rules, these symbols have to be mapped to a list of elements that represent the game objects in the level, called prototype structures;
- Define a rule structure to be used as the foundation of the generative grammar system;
- Create a graph rewriting system to implement the graph grammar approach;
- Apply Dormans' method of dividing the level generation into two steps, the mission map and space:
  - The mission map is generated using the generative grammar system and is a collection of steps that the player can take to complete the level;
  - The space map uses the mission map to select prototype structures to be included in the level, it also determines their position.

- Use a search-based approach to implement a validation system that can be tweaked by the developer using a combination of settings;
- Use the validation system to create levels with higher playability, where the playability of a level depends on three factors:
  - **Feasibility**: This measures if a player is able to complete the level. This is validated with the use of a very simple player AI, to check if a player can reach the next step of the level being generated;
  - **Interesting design**: This measures if a player wants to complete the level, whether the level structure is too monotonous, or whether the levels are sufficiently diverse from one another. This is validated by scoring multiple generated levels and selecting the best one, the scoring uses a set of parameters defined by the developer;
  - **Difficulty**: This measures if the level provides the player with enough challenge to keep him interested. This can be affected by two elements: the number of enemies in the level and the jumping or moving difficulty of the level. The former is controlled by the definition of the rule set and the latter is controlled by how far the simulated AI can jump while traversing the level.
- Make a game prototype that uses the framework developed to generate levels;
- Evaluate the generator expressivity according to four data points: linearity, feasibility, density and leniency;
- Test the prototype-generated levels with human players, to see how the levels perform in terms of level design, navigation, length, difficulty and enjoyment.

In the end, the generator created revealed that even applied in a simple form, the use of generative grammars as a rule system, is not only feasible but also flexible, enabling developers to easily modify the layout of the generated levels and introduce new layouts. The study also showed that the approach of dividing a level creation into distinct mission and space domains not only works for 2D platformer games but also streamlines the entire process. Moreover, the study indicates that utilizing multiple targeted grammars in small amounts can lead to improved outcomes while being more manageable.

1.2 Document structure

This document is organised as follows:

Chapter 2 - Background and Related work: An introduction to important subjects and terms that this project uses or mentions, followed by a summary of the previous work done on this subject;

Chapter 3 - Methodology: A reflection and analysis of some of the project's challenges and how they affect the project, followed by a description of the design and decisions made to build the generation system;

Chapter 4 - Implementation: A detailed description of the implementation used to develop this study's prototype game called ”*2D Platformer Generator*”;

Chapter 5 - Analyze and Results: A description of the procedure and results of an analysis performed to assess the generator's capabilities, as well as a description of the experiment done with participants to see how the levels would perform.

Chapter 6 - Conclusion: A final reflection on the project, how it fits in with previous work, and what exciting work remains to be done and researched.

Chapter 2

Background and Related work

This chapter provides an overview of the relevant background and related work related to the thesis topic. In particular, this chapter will discuss the key studies, theories, and concepts that provide the foundation of the current work. Additionally, this chapter will also review the related work from other researchers in the field and how that work has informed the current research. By discussing the related literature, this chapter will provide a contextual framework for the reader to understand the research topic and the current study.

2.1 Background

This section is a discussion of topics relevant to this study, it serves as a gentle introduction to important subjects and terms that this project uses or mentions.

2.1.1 Platformers games

Games come in all different shapes and sizes, so it is important to define the type of game that this project is trying to create. Two-dimensional platformer games range from simple side-scrolling Runner games to more complex Puzzle-platform games. They differ in terms of pace, style and goals. This research focuses on the development of platform-adventure games such as *Super Mario Bros.*, *Sonic the Hedgehog* and *Metroid*. The reason for this selection is that these games are iconic and extremely popular, with *Super Mario Bros.* being the subject of numerous artificial intelligence (AI) and PCG studies.

The game mechanics in a Platformer present a series of challenges since the player is bound by gravity; he can move left, right, and fall down, but he can only go up by jumping, and this is frequently a very short distance upwards. As a result, the dynamics between platforms, traps, enemies and gaps are crucial in these types of games.

2.1.2 Aspects of procedural content generation

Following the brief description of PCG in the introduction chapter, we will now delve into a more comprehensive analysis of this topic. Most of the information presented in this sub-section can be found in greater detail in the book by Shaker, et al. called *"Procedural Content Generation in Games"*[2].

PCG refers to computer software that can generate game content following a set of instructions constructed by a developer. The term "content" is defined by Shaker, et al. in their book as any parts of a game that has an impact on gameplay: the levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc. But excluding the AI behaviour of not non-player characters or the game engine itself. The term "game" is notoriously hard to define, from ancient civilizations to modern game studies, many definitions exist to describe this term. Philosophers like Plato and Aristotle contemplated the role of games in human society. The historian Johan Huizinga introduced a significant and influential concept of "game", known as the *"magic circle"*, which refers to a metaphorical boundary that players enter when engaging in play, setting temporarily aside real-life rules for the rules of the game and entering a safe, contained space for exploration and experimentation with a playful mindset. Despite the various attempts to define the term "game", the intricacies of this term continue to challenge our understanding, however, exploring these complexities is beyond the scope of this project. So in a more literal sense, for PCG, the term "game" refers to video games, board games, card games, puzzles, etc. The term "procedural" refers to a specific way of doing something, a set of steps that must be followed, or an algorithm. And finally, "generation" implies that something is created.

With the increase of the spectrum of PCG problems and solutions, there was a need to develop a taxonomy that could be used to compare and classify PCG approaches. Below is a brief version of the taxonomy provided by Togelius, et al. [7] and revised in the book mentioned above:

- **Online versus offline**: PCG approaches can be used to generate content online while the user is playing the game, or offline during game development or before the start of a gaming session.
- **Necessary versus optional**: PCG can be used to create either necessary game content for level completion or auxiliary content that can be discarded or traded for other content.
- **Degree and dimensions of control**: PCG content generation may be controlled in a variety of ways. One method for gaining control over the generation space is to use a random seed; another method is to employ a set of parameters that regulate the content creation along a number of dimensions.
- **Generic versus adaptive**: The term generic content generation (GCG) refers to the method of creating content without taking into account player behaviour. In contrast, adaptive, personalized, or player-centred content production generates material based on player interactions with the game.
- **Stochastic versus deterministic**: In contrast to stochastic PCG, where replicating the same content is typically not possible, deterministic PCG allows for the regeneration of the same content given the same starting point and generating parameters.
- **Constructive versus generate-and-test**: Constructive PCG generates content in a single pass, as is frequent in roguelike games. In contrast, generate-and-test PCG algorithms alternate generating and testing in a loop, repeating until a good solution is created.
- **Automatic generation versus mixed authorship**: PCG allowed for very limited involvement from game designers, who typically changed algorithm settings to regulate and steer content development while creating endless versions of playable content. However recently a new paradigm

has emerged where a human designer or player collaborates with the algorithm to develop the required content, called the mixed-initiative paradigm.

An important distinction between this kind of generation and something like art generation is that the content generated must take into account the game's design, requirements, and constraints, and one of the most notable characteristics of a game is that it should be playable. In the case of 2D platformers, for example, a player should be able to complete a level from beginning to end. Aside from meeting the constraints of a game, there are other desirable features that a developer may seek in a PCG solution. In their book Shaker, et al. provided a list of some of the most common desirable properties of PCG:

- **Speed**: The speed requirements vary greatly, depending on the game's complexity and level of detail and whether the content generation is done during gameplay or during the development of the game. For this project, speed is not a priority because the generation is done in the level selection screen and not while the player is exploring the levels.

- **Reliability**: Some generators just output content with any validation, others can guarantee that the content they generate meets certain quality criteria. This is more important depending on the types of content; for example, a missing exit on the level is a catastrophic failure, whereas an odd-looking tree simply looks odd. It is critical for this project to ensure that a level has at least one path that allows the player to travel along it and reach the goal.

- **Controllability**: It is often desirable that a content generator can be manipulated so that a human user or an algorithm can specify some aspects of the content to be generated. There are many possible dimensions of control, such as changing the properties of an element or a level capable of adapting to the skill of a player. The generator created for this project takes as an input a set of parameters and construction rules that alter certain aspects of its generation.

- **Expressivity and diversity**: There is a necessity to develop a diversified range of content in order to avoid the content appearing to be small variations on a theme. There must exist a balance between changing so little in the content that is barely noticeable and changing so much that it becomes something completely different and chaotic. For this project, this balance is primarily achieved in the construction of the rules; they must allow the levels to expand in many ways, but not so arbitrarily that the validations become too complex or impossible to perform. Building level generators that provide different content without sacrificing quality is no trivial task.

- **Creativity and believability**: In most cases, it is undesirable for the content to appear to have been generated by a procedural content generator. However, this is not a major concern for this research work. Because, in most situations, this necessitates a fine-tuning of art and design that extends beyond the scope of this project.

2.1.3 The search-based approach

There are numerous approaches to tackling a PCG problem, many of which are unique to the game for which they were designed, but one approach that is frequently used and studied in PCG research is the search-based approach. This section gives a brief overview of the approach and the components that comprise it.

In search-based PCG, a stochastic search algorithm with the help of an evaluation function is used to search for content with the desired qualities. The supposition is that, regardless of how random the initial answers are, by picking the solutions that are better suited to the task and tweaking them for the next iteration while discarding the rest, the process will eventually arrive at a local maximum for the ideal solution. This method requires three key components to function:

- **A search algorithm**: This is the core component of this approach, it represents the way the content is searched to find the optimal solution. Evolutionary algorithms are often very suited for this purpose, the basic principle behind them is to retain a population (also known as chromosomes or candidate solutions) by evaluating them with an evaluation function in each generation, only the fittest (highest evaluated) individuals are allowed to reproduce, meaning they are copied with small changes (mutation) or combined (recombination or crossover), while the least fit are discarded from the population.

- **A content representation**: In order to be searched and evaluated the content has to be represented in some form, the type of representation chosen has an impact on the efficiency of the generation algorithm and the space of content the method will be able to cover. In evolutionary algorithms, solutions in the generation space are usually encoded in simple forms that can be evaluated more efficiently called genotypes that later become phenotypes. In a case where game content is generated, the genotype may be the instructions for building a game level, and the phenotype could be the actual game level.

- **One or multiple evaluation functions**: An evaluation function provides a score (a fitness value or evaluation value) to each possible solution that is encoded in a representation. If a proper evaluation function is not employed, the evolutionary process will fail and no good content will be found. In general, the evaluation function should be constructed to evaluate some desired aspects of the artefact, such as its playability, regularity, entertainment value, and so on. Evaluation functions may categorize into three types:

  - **Direct**: Where extracted features from the generated content are mapped to a fitness value, meaning that the phenotype representation is directly evaluated by the function. This kind of function is normally fast to compute and easy to implement because it tends to evaluate a single aspect of the game content. However, establishing a direct evaluation function for some aspects might be difficult.
  - **Simulation-based**: Where AI agents are utilized to assess the quality of generated content, this is done generally by scoring the game content according to statistics calculated from the agent's behaviour and play style. The proficiency and capability of the AI can be changed depending on the type of evaluation task and desired outcome, for example in this project an AI that has a smaller jump distance is used to create levels that have less difficult jumping challenges.

– **Interactive**: Where the content is evaluated based on data gathered through human interaction, there are two types of data collection: implicit and explicit. Implicit is accomplished by collecting statistics on the user's employed actions and behaviours, such as how long it took to complete a level, how many coins were collected, and so on. Explicit involves the user directly providing input by filling out a form or reporting preferences. However, both approaches have limitations, hence a hybrid strategy is sometimes employed to mitigate these by gathering information across different modalities.

2.1.4 The generative grammar approach

Another approach is to instruct the program on how to evolve a solution using a collection of established rules; generally, one or more formal grammars are defined and utilized to generate solutions that follow the language(s) rules established by the grammar(s). This section will first introduce the notion of grammar, followed by an explanation of the intricacies of using them in a PCG context.

Formal grammar

In language theory, a formal grammar describes how to use a language's alphabet to form strings that are valid according to the language's syntax. It is called formal because it does not specify the meaning of the strings or what may be done with them, but merely their form. Noam Chomsky in his generative grammar theory introduced the concept of formal grammar as a more powerful method of describing language[8]. However, they also have many other applications, for example, the in field of biology a type of formal grammar called Lindenmayer system (L-system) is used for research in multicellular organisms [9] and also to generate realistic-looking vegetation[10]. In PCG grammars can be used to generate levels that follow a desired structure of syntax, a few examples of this are presented in the related work section.

But first, a grammar G can be defined more formally as a tuple (N, Σ, S, P) where: N is a finite set of non-terminal symbols that don't appear in the strings produced by G; Σ is a set of terminal symbols disjoint from N; P is a set of production rules and S is the start symbol and $S \in N$; Σ and N form the grammar's alphabet. The process of applying a grammar is done by searching through a string, and each time a symbol or sequence of symbols that appear on the left-hand side (LHS) of a production rule is found, those symbols are replaced by the right-hand side (RHS) of that rule. This procedure is repeated until either a specified number of iterations is reached or all symbols are terminal. For example, Fig. 2.1 shows a very simple grammar with two production rules, the symbols "S", "A" and "B" are non-terminal symbols often represented by upper case letters, and "r", "gra" and "mma" are terminal symbols normally represented by lower case letters. Starting with a string containing only the symbol "S", in the first iteration rule 1 transforms "S" into "ABr". In the second iteration, two rules could be applied to the string "ABr" depending on the way it is processed, if the search is done left to right rule 2 is applied first, but if the search goes right to left then rule 3 is applied first, in this example rule 2 was applied first turning "ABr" into "$graBr$". The third iteration then applies rule 3 making the final string "$grammar$". This describes a sequential rewriting of the string, all the changes made by the rules are written in the same string before the next symbol is considered, another method of applying production rules is parallel rewriting where all the possible rules are applied at the same time by not changing the original string but instead saving the results of applying the rule in different separate strings. If a grammar has exactly one rule that each symbol sequence, such that there is no doubt what rule will be applied for a given string, then the grammar is called deterministic, otherwise the grammar is non-deterministic resulting in several conceivable outcomes for a given string.

Grammar:
1. $S \Rightarrow ABr$
2. $A \Rightarrow gra$
3. $B \Rightarrow mma$

Solution:
s. $S \rightarrow ABr \rightarrow graBr \rightarrow grammar$

Figure 2.1: An example of a simple grammar

The same concepts apply to the creation of game levels; by establishing an alphabet made up of symbols representing different aspects of a level, such as enemies, traps, gaps, platforms, and so on, it is feasible to build a set of rules governing how these symbols can be combined to form a level. Fig. 2.2 shows a grammar that could create a level where the player would enter, find a key, open a lock and reach the end (s1) or it could create a level where the player would enter, defeat an enemy, find a treasure and reach the end (s2). A more complex example can be found in Shaker, et al. work [11] where string grammars were used to generate levels for the game *Super Mario Bros.*.

Grammar:
1. $L \Rightarrow start\ C\ end$
2. $C \Rightarrow key\ lock$
3. $C \Rightarrow enemy\ treasure$

Solutions:
s1. $L \rightarrow start\ C\ end \rightarrow start\ key\ lock\ end$
s2. $L \rightarrow start\ C\ end \rightarrow start\ enemy\ treasure\ end$

Figure 2.2: An example of a grammar describing the syntax to create a simple level

However, grammars are not limited to having their content represented as strings. For instance, shape grammars[12] work with geometric shapes, transforming the form of a shape according to their production rules. Graphs can also be used in grammars, graphs are more versatile than strings at expressing non-linear game levels. While a string grammar, such as the one depicted in Fig. 2.2, can efficiently represent a linear level, it falls short when it comes to representing more complex scenarios such as levels that require multiple keys to open a lock or have a second secret path to the end. In contrast, graphs provide a more suitable representation for these scenarios. Fig. 2.3 illustrates the representation of the previous two examples in both graph and string grammar formats. As evident from the figure, the graph form is more intuitive at conveying information, whereas the string form appears more complicated. To represent multiple paths in the string version, certain assumptions were necessary. For instance, the use of the symbol ”//” was introduced to indicate that all nodes connected by this symbol are accessible from the previous node separated by an empty space. The need for such assumptions contributes to the increased complexity of representing non-linear maps using only strings, making them more challenging to work with.

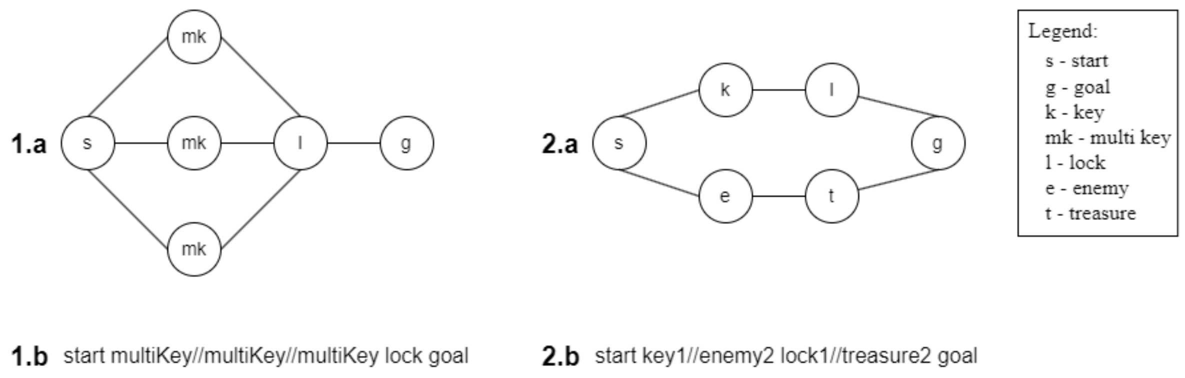


Figure 2.3: Shows two levels 1 and 2 represented as graphs in 1.a and 1.b and the equivalent represented as strings in 1.b and 2.b. Level 1 is a level where multiple keys are needed to unlock a lock and reach the goal and level 2 is a level with multiple paths to the end

Graph Grammar

Graph grammars are a powerful tool for generating game levels. They allow developers to create complex and varied levels with a simple set of rules. A graph grammar defines a graph structure, which is composed of nodes (or vertices) and edges (or arcs). Each node represents a game element and edges are used to define the relationships between the elements. Graphs come in a variety of shapes and sizes. Edges can be both directed and undirected. Some graphs can have their nodes with labels, while other graphs can be attributed by having nodes that contain variables associated with them. Developers can use graph grammars to specify the basic structure of game levels, such as the placement of walls, platforms, and obstacles. Graph grammars may also be used to specify the different types of characters, items, and enemies that can occur in a level. By applying different rules to the graph structure, developers can generate unique and varied levels with minimal effort.

The mechanics of applying a graph grammar to transform/rewrite a graph are similar to that of using a string grammar to change a string, a graph called host is repeatably searched to find a subgraph that is isomorphic to a rule's LHS graph and if found, that subgraph is replaced by the rule's RHS graph. However, the process of searching for a subgraph that is isomorphic with the rule's LHS, is considerably more complex[13] than searching for a sequence of symbols in a string. There are a few algorithms for solving this subgraph isomorphism problem, of special notice are the Ullmann algorithm[14] that uses a recursive backtracking procedure, the VF2[15] an advanced form of Ullmann algorithm that is faster and requires substantially less memory, and more recently the Glasgow Subgraph Solver[16] which adopts a constraint programming approach, using bit-parallel data structures and specialized propagation algorithms for performance. The Ullmann algorithm was used for this project's prototype because it is simpler to implement and performs well enough.

The process of rewriting a graph by replacing the subgraph with the rule's RHS is also different for graphs. One approach is the algebraic approach, when a match is detected in the host graph, any edges connecting the nodes in the match to the rest of the host graph are destroyed, as are any matching nodes in the host graph that are not also present in the rule's RHS. The nodes on the RHS but not on the LHS of the rule are then added to the host graph, and new edges are created to connect the newly added nodes to the remainder of the host graph. To identify which nodes in the LHS are the same nodes in the

RHS a relation between nodes on the two sides is created by numbering the nodes and making matching nodes have the same number. To know how to link the new nodes to the rest of the graph, a technique called *pushout* is used, where context elements connect the new nodes to the rest of the graph. A context element is a node that can be found on both the left and right sides of a rule. Due to the presence of such nodes, the new nodes are immediately connected to the remainder of the host graph. There are two approaches that use the *pushout* technique, the single-pushout (SPO) and double-pushout (DPO), the main difference between the two is that the DPO gives more control over what edges should be deleted even in cases were the node connected by this edges is removed, whereas in the SPO if a node is removed all edges connected to that node are removed. In the DPO a rule also requires a third graph called the *interface graph* that contains the context elements. Given that the rewriting needed for the prototype was simple, a SPO approach was used.

Grammars can be a powerful tool for generating game content, but the process of creating a level by hand has many different steps. So trying to construct every aspect of a level in a single generation layer can be limiting and very difficult to maintain. Instead, a different approach is to use multiple layers or phases, where each phase is responsible for a different step in the level generation process. Joris Dormans for example, divided the creation of levels for action-adventure games into two phases[6]: the mission phase, which represents the objective or player actions needed to complete the level, and the space phase, which represents the level geometry. Dormans' approach serves as a stepping stone for this project.

2.2 Related Work

A fair bit of research has been done on the topic of PCG, this section focus on presenting some of the work that inspired this project. The section is divided into subsections, each representing a separate research project.

2.2.1 Automatic dungeons generation

In 2002, David Adams demonstrate the use of graph grammars to generate dungeons games[17], he did this by developing a graph grammar system that took in generation strategies and a set of predefined game-specific grammar rules and generated a graph representing a topological description of the level, which is a description that only represents the order in which objects in the level are encountered rather than their physical appearance. Adams used object-oriented abstraction to allow the developer to define their own strategies for picking production rules and also to define parameters such as the size or difficulty of a level that was used by a level strategy to output a level with the desired parameters.

2.2.2 Patterns in platform game level design

Kate Compton and Michael Mateas in their study[18], proposed a four-layer hierarchy to represent levels of platform games that focus on repetition, rhythm, and connectivity. The basic layer is components, the building block of a platformer level, which should be composed of an obstacle and a resting stop, like a gap followed by a platform to land on. On the second layer, patterns would dictate how the components should be grouped together and still maintain the rhythmic movement for the player. But since this pattern would only be linear, in the third layer they would be encapsulated into cells that would then be connected to one another following a cell structure to create a non-linear level. The fourth layer is the level itself, which is made up of cell structures that were constructed by studying the level structure patterns of previously released platform games.

2.2.3 Adventures in level design

As briefly mentioned at the end of section 2.1.4, Joris Dormans in his study[6] investigated strategies to generate levels for action-adventure games and proposed the use of generative graph grammars to procedurally generate missions for action-adventure games, which, due to their nonlinear structure, are a good fit for games that involve exploration. In this research, Dormans suggested that in order to build more sophisticated level layouts, the generation process should be separated into the creation of a topological map called ”mission” which represented the steps needed to complete the level, and then use that mission to build a geometric map called ”space” holding the physical information of the level. Graph grammars were used to generate the missions and Shape grammars were used to generate the space. He came to the conclusion that by breaking the operation down into these two parts, we may capitalize on the strengths of each type of grammar, and that these methods can be used to create levels on the run, allowing the game to adapt to the player’s actions.

2.2.4 Evolving levels for *Super Mario Bros.*

Noor Shaker, et al. [11] demonstrated a way of generating *Super Mario Bros.* levels using generative string grammars, by building an alphabet where the terminal symbols abstracted the game content into function calls. The functions had the same name as the element they represented and took as parameters the properties of the elements. Using that alphabet they constructed a set of rules that could generate levels. In other to test the grammar generator’s capabilities they used two other different generators to create levels. Each generator’s output was analyzed and evaluated using a set of level design metrics, namely linearity, density, leniency, and compression distance. A comparison was drawn between the generators, this approach introduced a general framework that game designers could use for comparing content generated by different generators.

2.2.5 Putting it all in one place

In 2016 Shaker, et al. [2] published a textbook book called ”*Procedural Content Generation in Games*” containing an overview of the research field of PCG the first of its kind. The book was written based on a syllabus of academic papers on the subject the authors put together for a course at the IT University of Copenhagen, most chapters within the book introduce new methods together with fascinating and relevant domains that illustrate their practical use. It also contains a revised taxonomy based on the one done by Togelius, et al. [7], as well as important concepts about PCG. It acts as a guide for PCG researchers from many fields to learn about the approaches developed in other communities as well as a good learning textbook.

2.3 Summary

In this chapter, we introduced several fundamental concepts that will serve as the basis for the subsequent sections of the dissertation. Specifically, we discussed the taxonomy that will be utilized, as well as the properties that are typically desired in PCG. We also explored two different approaches for addressing PCG problems, which will be integrated into this project. Furthermore, we referenced prior research conducted on the subject, including Joris Dormans' investigation of strategies for generating levels in action-adventure games. This research will serve as a foundation for the work presented later in the dissertation.

Chapter 3

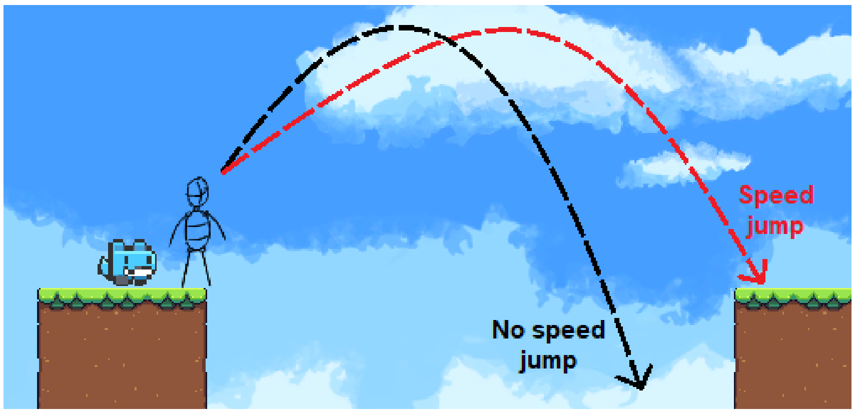
Methodology

The idea behind this project was to explore the creation of a framework capable of using a set of rules defined by a developer to generate playable two-dimensional levels for a platformer. This chapter presents the approach taken in this study to begin developing such a framework. It will begin with an analysis of potential problems and challenges that may arise during the development of such a framework, followed by a discussion of the framework design and how some of the challenges were addressed.

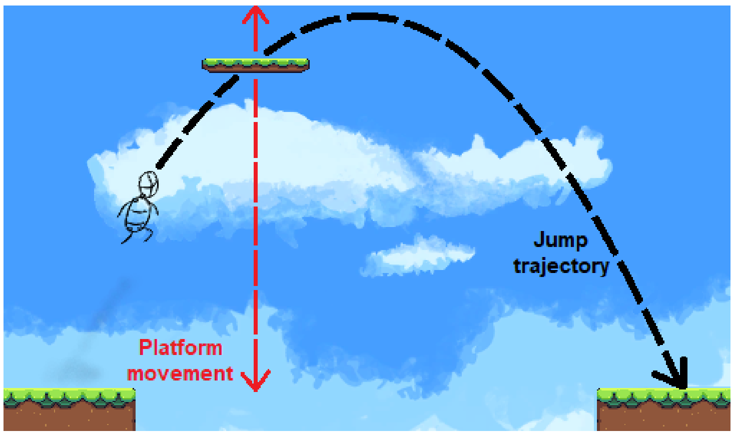
3.1 Problem Analysis

3.1.1 The force of gravity

The creation of a two-dimensional side-scroller platformer level brings a set of new challenges when compared to the creation of a level from a dungeon crawler game with a top-down view like Rogue. In the latter kind, the player's movement is not affected by an external force, this implies that as long as there is a guarantee that all the areas in a level are connected by a pathway the player can move freely anywhere there is a walkable path. In a platformer, however, the player's movement is limited by the artificial gravity provided by the physics engine, which means the player can only move left or right and move down by falling. The only way to move up is to jump, which often pushes the player a short distance upwards and initiates an arc trajectory, meaning that the player cannot freely control where he's going to land when he jumps, he can only predict. Validating if a level is playable becomes more complicated with the existence of a physics engine, the length of the player's jump is affected by the player's momentum at the initial moment of the jump, making it harder to calculate if a jump is possible or not. The validations become even more complicated if in order to make levels more challenging and add more variety, obstacles are placed in them, which can directly affect the player's path and momentum, making a platform that was within reach of a full-length leap, now not reachable because the player is being blocked from reaching the necessary speed (see Fig. 3.1a). Moreover, if the obstacles are not static their movement can also block the player from reaching the platform (see Fig. 3.1b), implying that to determine if the player can reach the next platform, the algorithm would have to calculate every jump imaginable at every speed the player may achieve, as well as every object placement combination in a radius around the jump gap at each timestep, making it a very daunting task. Because of the number of variables, it's more practical to use a player AI simulation. This simulation can be used as an evaluation function to filter out playable levels from a group of generated levels by having the AI play them and picking the ones where it can reach the end while discarding the rest. This player simulation could also serve as a simulation-based evaluation function (see section 2.1.3), to help generate playable levels in the level generation process. As part of the generation process, the function would determine a location on the map that is reachable by the AI player, this location would then be used for placing a new element on the map. This does implicate that the performance of an AI player influences the position of elements in the level, implying that a better performing AI will create more challenging levels. The AI can also be given handicaps to change the way it affects the generated levels, such as lowering its jump force to create levels with smaller jumps. The use of this type of evaluation function does have an increased cost in performance, this could pose a problem in games that need to generate levels during runtime. However, in the case of this project, the prototype generates all the levels when the player first enters the level selection menu, so this performance cost is not an issue.



(a) A scenario in which the player cannot get enough speed to complete the jump



(b) A scenario in which the player's jump is temporarily obstructed by a moving platform

Figure 3.1: Two scenarios in which the player's jump fails because of the environment around him

3.1.2 The price of abstraction

Creating a reliable generator that can generate and validate good levels for a specific game is difficult, creating one that could be used for any style of platformer game is even more challenging and comes with costs. No matter how well the software is made, there will still be some particular games that require a very specific type of level layout or validation that the generator will not be able to perform.

This is because there is no perfect balance between ”good” and ”challenging” levels. There is no one true best-level creator that fits all games perfectly, even if they are from the same genre, due to the nature of how game designers make games that play differently. Each game has different types of challenges. Creating a level of abstraction in the generator in order to make it customizable, such that a developer could not only provide the generator with the elements he wants in the level and how they should be connected, but also stipulate the validations that he wants the generator to make in order to create the best fitting levels, could potentially solve the problem, but at a certain point of abstraction, the generator would be so generic that the developer would have the same work creating his own. So there must be a balance between the time-saving qualities a generic generator framework can provide and the limitations of expressivity that come with those qualities.

3.1.3 Generation control

Another aspect to consider is the controllability of the generator, how much control should the developer have in the generation of a level. Should it be a one-to-one relationship between an input and its output, giving the developer complete control over every aspect of the creation of a level, or should the developer have no control whatsoever. Since the levels are built from a grammar, the layout of the level will reflect the syntax defined by that grammar's production rules, giving the developer that designs that grammar some control over the design of the level, however normally the cause and effect of changing a production rule in a non-deterministic grammar is not very clear since the rules associated with a symbol or set of symbols are picked at random, more control can be given to the developer if the rules themselves have a priority value so that the developer can specify which patterns should be picked more often, tailoring the generation process in the desired direction. Another way of increasing the controllability besides allowing the developer to tweak certain aspects of the generation with parameter values is to give the developer the option to define their own custom parameters which gives them the opportunity to make specific evaluations of their generated levels. Using these optional methods it is possible to create a generator that provides both a hands-off and a somewhat controlled approach to generating levels.

3.2 Design

As previously stated, this study follows Dormans' method of separating the level creation into two different parts: the mission map, which contains the basic composition and objectives of the level, and the space map, which contains the information and location of all the structures that will make up the level. This separation of priorities brings a more refined control to each stage in the creation of a game's level, by first focusing on the level's main objectives and elements and then worrying about where to place them and what their physical form should be like. As such, the level generation process is divided into three main steps (see Fig. 3.2), each of which is responsible for setting up certain aspects of the level, which get increasingly detailed as the process progresses. The first step is to evolve a mission map using a generative graph grammar, then in step two, a mission map is used to guide the creation of a space map, and finally, in step three, a level is built from the space map.

Create Mission Map
Create Space Map
Create Level

Figure 3.2: The three main steps of the level generation process
(Icons adapted from www.flaticon.com)

During the level generation, the generated content is validated using an evolutionary algorithm, meaning that multiple candidates are created at each stage of the level creation, and then evaluation functions are used to score the created content, the best candidates are then filtered out and used on the subsequent steps. Fig. 3.3 shows a flow diagram of the level generation process, it also shows that given n mission maps, the top m maps are used to generate m space maps, and from these m maps the best one is used to create a level.

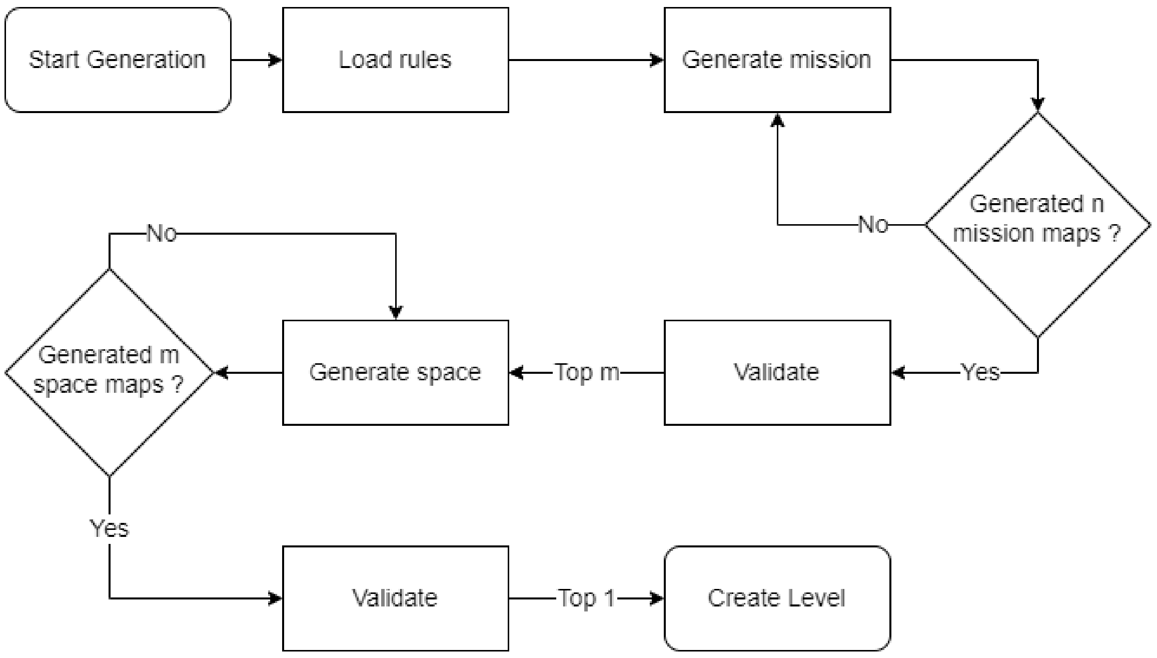
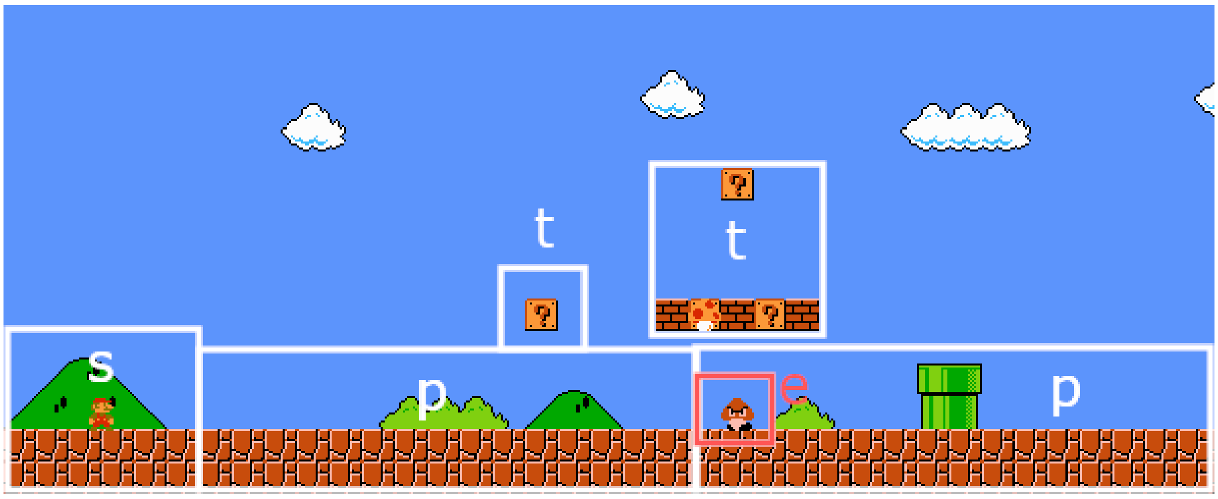


Figure 3.3: The level generation flow diagram

But to create a framework capable of generating levels for more than one specific platformer game, a level of abstraction has to be created to separate the generation process from the actual game details and elements, but still give the generator enough information for it to be able to evaluate and create usable levels. With this in mind, the framework was designed to work as independently from the game content and logic as possible, so a developer using the framework would first establish a grammar to build mission maps by defining an alphabet where the terminal symbols represent objectives or abstract elements of the level and a set of production rules that will dictate how the symbols in the alphabet should be joined together. And second, the developer would have to create a list of prototype structures that would be used to create the space map, these prototypes structures represent templates for the actual game objects used in the game, they hold the basic information like type, size and any other special property the element might have, but know nothing of their actual graphical representation. A developer can also establish specific parameters for the created level, such as the level's maximum size or the level's overall slope. The level validator then uses these parameters to rank levels and select the ones with the best fit.

To help explain all the steps of the generating process let's try to generate the start of World 1-1 in *Super Mario Bros. (SMB)* (see Fig. 3.4a). First, we need to divide the level into pieces to create the prototype structures. A prototype structure be be defined by a tuple $(type\ of\ structures,\ width,\ height,\ ...,\ other\ special\ properties)$. The start of this level contains a type of entrance, platforms, treasure boxes and an enemy *”Goomba”*, so at least four different prototype structures can be defined (see Fig. 3.4c). Then we must establish the terminal symbols for our mission grammar, since the mission represents a set of possible steps or objectives the player has to go through, for our example, the symbols could be ”s” for the start of the level, ”p” for any type of platform the player has to go through, ”t” for any type of treasure the player can pick up and ”e” to represent any type of enemy that might appear (see Fig. 3.4b). These symbols could then be associated with their logical counterpart prototype structure (see Fig. 3.4c). Although it is not exemplified here a mission grammar terminal symbol can and should be associated with more than one prototype structure, to create variety, the reciprocal is also true, one type of prototype structure could also be associated with multiple terminal symbols.



(a) The start of World 1-1 in *Super Mario Bros.*
(Adapted from www.nesmaps.com)

s	$-$	$start$
p	$-$	$platform$
t	$-$	$treasure$
e	$-$	$enemy$

(b) Terminal symbols

s	$\rightarrow$	$(”entrance”,\ width,\ height)$
p	$\rightarrow$	$(”platform”,\ width,\ height)$
t	$\rightarrow$	$(”treasure”,\ width,\ height,\ numBoxes,\ items)$
e	$\rightarrow$	$(”Goomba”)$

(c) Terminal symbols associated with prototypes structures

Figure 3.4: Representing the start of World 1-1 in *Super Mario Bros.* as terminal symbols for a mission grammar and as prototypes structures

3.2.1 Mission maps and grammars

A mission map is a graph that represents an abstract layout of the level, meaning that each node on the graph represents a step in the level, and the edges represent the connections between these steps. The term “step” is used here to mean an action or element the player needs to interact with or go through in the level. In the case of the start of *SMB World 1-1* (3.4a) a potential mission map could look like the one represented in Fig. 3.5.

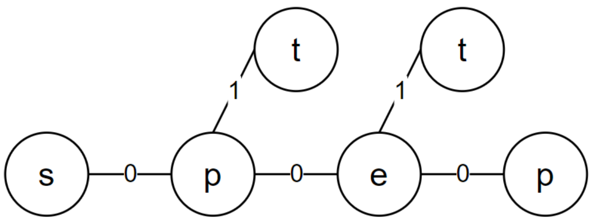


Figure 3.5: A mission map representation of the start of World 1-1 in *Super Mario Bros.*

Having defined the terminal symbols or ”steps” that should exist in our level map. The next stage is to create one or more grammars that can rearrange these symbols in interesting ways to create mission maps. The type and structure of the grammars should reflect their purpose, in Dormans’ study[6] the mission map could only affect the layout in cases where a node was associated with another by a special edge, such as a link between a key and a locked door, the key has to be accessible in a place not blocked by the locked door it should unlock. The type of grammar he used to evolve and create these maps was a graph grammar, where the productions rules were built to form interesting mission maps for action-adventure games, the layout of the level was purely the responsibility of the space map, which was built using a shape grammar where the production rules change the shape of the level. In the case of this study, only the mission map is generated through the use of a grammar, more specifically a graph grammar. The space map is built using only a mission map as a reference and is not transformed by any grammar. So to help form the space map, the edges that link the nodes on a mission map graph have a slope attribute, that denotes the direction of that connection line indicating where the target node of that edge should be placed in relation to the source node. This does not negate the need for a space map, however, because it is merely indicative of the direction it should be in, not defining the actual position. To keep things simple for this study, the slope of the edge can only have three values: -1, 0 and 1. Indicating whether the target should be placed at a lower, the same, or a higher height, respectively (see Fig. 3.6). But going forward it would be interesting to give the connection edge an angle so that a target node could be placed anywhere around the source node. It is also useful to give the edge some information about its nature, just like in Dormans’ work it is important to have a way of telling the generator that a key node opens a lock node, this could have been done by giving the edge between these two nodes a property type.

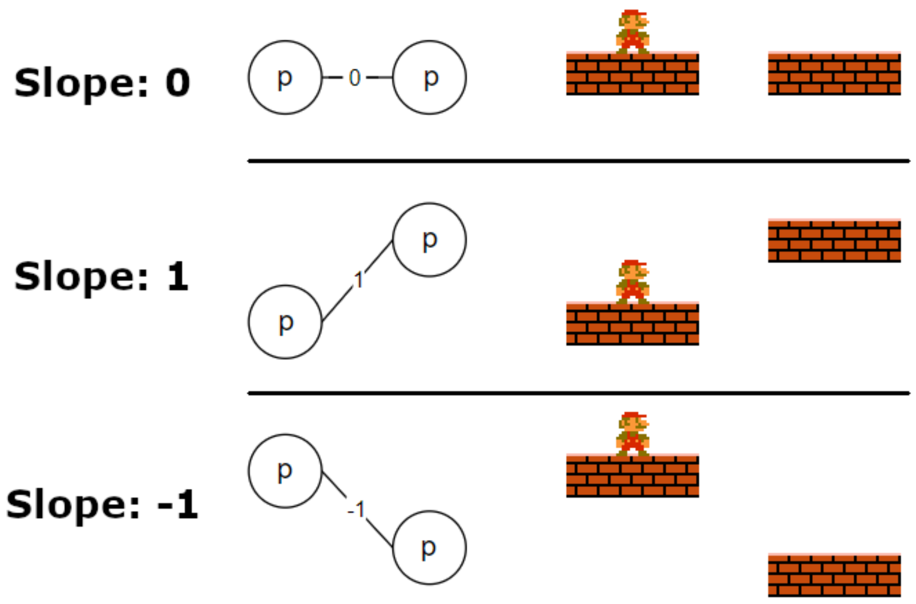


Figure 3.6: The three types of edge slope used in this study
(*Super Mario Bros.* sprites adapted from www.nesmaps.com)

The production rules of graph grammar consist of a LHS graph pattern that is replaced by that rule's RHS graph (see section 2.1.4 - Formal grammar). However, when creating a rule set capable of generating different-looking levels, it is advantageous to have multiple choices of replacements for the same pattern, this way even if the rules are static the outcome is different. This can mean multiple rules with the same left pattern but different right patterns or a single composite rule that maps the left pattern to multiple right patterns. The latter option was used for this study, each composite rule is composed of a left-side graph pattern that can be replaced by multiple right-side graph patterns, and each right-side of the rule has a probability attribute that indicates the chance of it being selected to be applied, meaning that right-sides with a higher probability value will be picked more often. This attribute is set by the developer during the creation of a rule. Since the edges of a mission map's graph have the slope attribute, this permits the definition of patterns when defining production rules, such as having a parallel path going up and down or having a hub where three paths diverge in different directions. This causes the levels generated with a particular set of rules to share a resemblance in patterns between each other, which might be advantageous if the idea is to build a game with a certain aesthetic resemblance between the levels. Fig. 3.7 shows an example of a simple grammar with composite production rules that could be used to form the SMB mission map in Fig. 3.5.

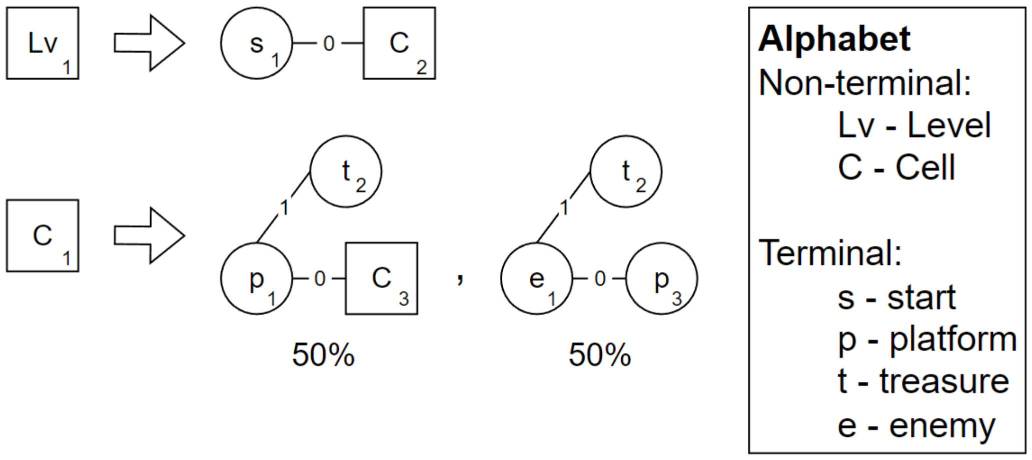


Figure 3.7: The production rules of a grammar that can create the mission map seen in 3.5

3.2.2 Step one: evolving mission maps

Every mission map starts has a graph called ”host” with a single node, this host is transformed by applying rules from one or more grammars until there are no more possible rules to be applied or another predefined condition is met, the only constant condition is that the host graph contains only terminal nodes at the end of the morphing process. During each step of this transforming process (see Fig. 3.8), the host is first searched to find what rules can be applied, then a rule is selected based on a strategy, random selection being the default strategy. After a rule has been selected, if it can be applied in more than one spot, then a random spot is picked to apply the rule. Finally, that spot is morphed to one of the rule’s RHS according to their priority attribute.

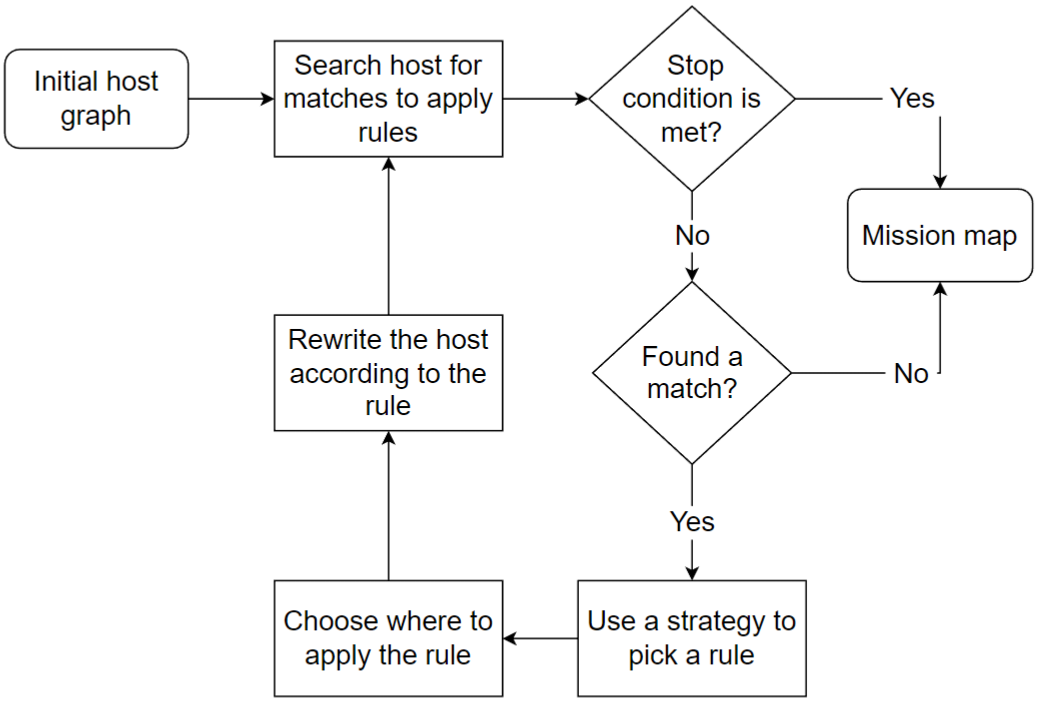

Initial host graph
Search host for matches to apply rules
Stop condition is met?
Yes
Mission map
No
Found a match?
No
Yes
Rewrite the host according to the rule
Choose where to apply the rule
Use a strategy to pick a rule

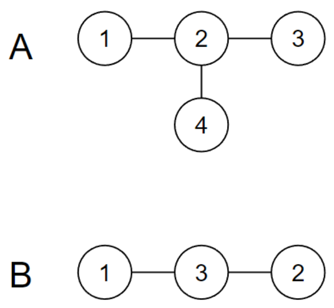
Figure 3.8: Mission map creation flow diagram

Searching for applicable rules

To discover which rules can be applied to the host graph, it is necessary to discover if the host contains any subgraph that is isomorphic to one of the rules' LHS, this is called the subgraph isomorphism problem, and even though it is an NP-complete problem there are algorithms that can solve this problem relatively fast for specific types of practical problems. One algorithm is Ullmann's subgraph isomorphism algorithm[14] developed by J.R. Ullmann in 1976. Given graph A and graph B, to find if B is subgraph isomorphic to A, first represent both graphs as adjacency matrices (see Fig. 3.9), then create a correspondence matrix M^0 of size $|V_B| \times |V_A|$ where:

$$m^0_{i,j} = \begin{cases} 1, & \text{if } deg(V_{Aj}) \geq deg(V_{Bi}) \ \wedge \ type(V_{Aj}) = type(V_{Bi}) \\ 0, & \text{otherwise} \end{cases}, m_{i,j} \in \{0, 1\}$$

The correspondence matrix shows all the possible nodes from graph A that can be considered candidates for nodes in graph B. A node j in graph A (denoted as V_{Aj}) is only considered a candidate for a node i in graph B (denoted as V_{Bi}) if node V_{Aj} has at least the same number of adjacent neighbours (denoted as $deg(V_{Aj})$) and is also of the same type (denoted as $type(V_{Aj})$) as V_{Bi}. If the former two conditions are met then node V_{Aj} is marked as a candidate to node V_{Bi} by setting $m^0_{i,j} = 1$, otherwise $m^0_{i,j} = 0$. It is important to note that the conditions to consider a node as a candidate are specific to each use case of the algorithm, for simple problems, checking for the number of adjacent nodes might be enough, but in cases where the nodes have properties such as a "type", these properties need to be checked as well. Fig. 3.10 shows the correspondence matrix M^0 constructed from Adj_A and Adj_B.



$$Adj_A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$Adj_B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Figure 3.9: The adjacency matrices of graphs A and B

$$Adj_A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad Adj_B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \Longrightarrow M^0 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3.10: Building a correspondence matrix M^0 from Adj_A and Adj_B

With the correspondence matrix, a search tree can be constructed by designating a potential node candidate from graph A for every node in graph B and verifying whether an isometric correspondence exists. In practice, this is achieved by creating permutation matrices M', where each row of the correspondence matrix has only one non-zero element. Then testing each one to see if they are a valid permutation matrix. An example search tree can be observed in 3.11. The root of the tree is the correspondence matrix, and each branch is constructed by setting all values in a row to zero, except for one. This process is repeated for each row until a leaf matrix with only one non-zero element in each row and column is reached. During this process, any branches that result in more than one non-zero element being present in a column can be discarded, as it indicates that the same candidate node would be mapped to two distinct nodes of the other graph. Once the search tree is complete, all the possible associations between nodes of the two graphs are represented by the leaf matrices.

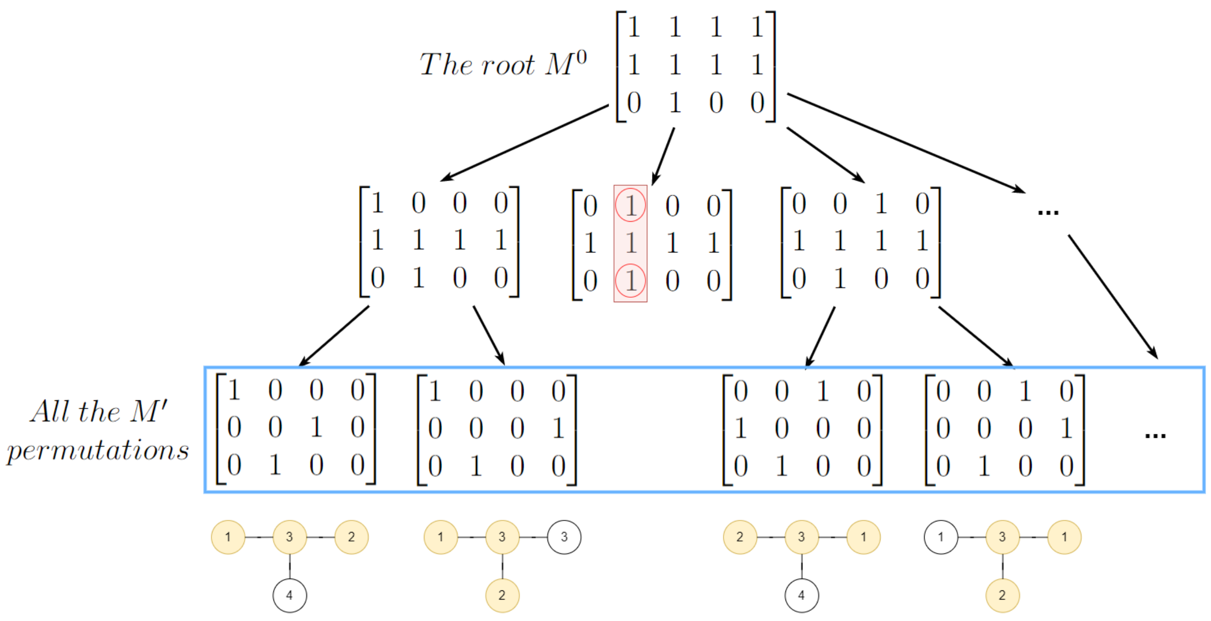


Figure 3.11: The search tree to find valid permutation matrices M'

Ullmann also proposed a refinement rule for pruning or cutting off invalid branches from the search tree by considering the following: if a node v_j from A is among the candidates for a node v_i in B, then for each adjacent node of v_i in B, denoted v_i^{Adj}, there must be at least one node in A, denoted v_j^{Adj}, that holds: v_j^{Adj} corresponds to v_i^{Adj} and v_j^{Adj} is adjacent to v_j in A. If it does not, v_j is removed from the candidates for node v_i by setting $m_{i,j}^0 = 0$. This check is performed until no more removals are possible even with the new changes.

Once a valid permutation matrix M' representing a candidate subgraph from A is found it can be tested using the criterion:

$$P = M'AM'^{-1}, \text{ iff } P\ is\ isomorphic\ to\ A,\ with\ a\ correspondence\ M'$$

By calculating $P = M'AM'^{-1}$ if the resulting matrix P is equal to B by $(b_{i,j} = 1) \Rightarrow (p_{i,j} = 1)$ then this subgraph is isomorphic to B. Using this algorithm we can search the host graph to find where and what rules can be applied. But using this algorithm can be costly, because of all the matrix manipulation and multiplications involved, this cost can be minimised using more efficient ways of doing these manipulations such as encoding the matrices as bit vectors and using bit-tweaking operations. Another way to reduce the number of searches in the host graph is to give each rule a list of associated rules, this list contains the other rules that could be applied if any of this rule's RHS graphs were to exist in the host graph. With this associated rule list when is time to pick a rule the algorithm already has knowledge of some of the rules that could be applied and one could be picked without having to search the host graph, it is still necessary to search where this rule can be applied, but the search is done for a specific rule instead of all of them. Additionally, it is possible to reduce the cost of the search even more by stopping at the first permutation matrix that satisfies the criterion above, not bordering on checking the other branches of the search tree, of course, this raises the possibility of always selecting the same first one; this can be avoided by selecting at random the first column of each row of the comparison tree to be fixed at 1 and setting all others to zero when creating the search tree. To resume, grammar production rules are created with the knowledge of their associated rules, then instead of searching for what rules can be applied, a rule is picked from the associated list, and then the host graph is searched to find the first place the rule can be applied.

Graph morphing

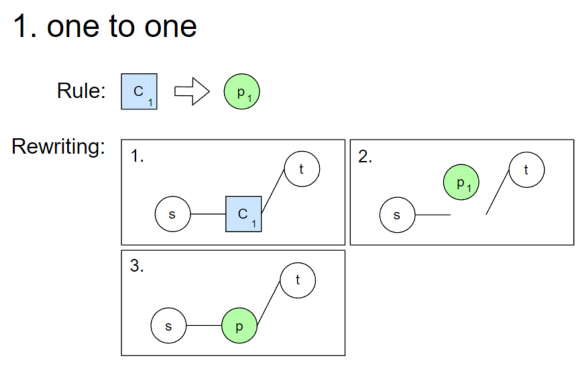
Once a rule has been chosen and a valid host subgraph match has been selected to apply it, the next step is to replace that match with the rule's RHS mutation graph. This process is known as graph rewriting. In this study, an algebraic approach was employed, utilizing a SPO method as outlined in section 2.1.4. The approach used in this study involves four distinct cases, depending on the number of nodes present in both the match and mutation graphs, an illustration of each of these four cases can be seen in Fig. 3.12. However, before delving into the specifics of each case, it is important to reference that when a subgraph match is found, the nodes in the subgraph are numbered exactly like the rule's LHS numbering, this is important to be able to see the link between the node on the LHS and on the RHS.

The first case (see Fig. 3.12a) is a context-free case where a single ”match” node is replaced by one ”mutation” node. This process is done by transferring all adjacent edges of the ”match” node to the ”mutation” node, then removing the ”match” node from the host graph, and finally adding the ”mutation” node to the host graph.

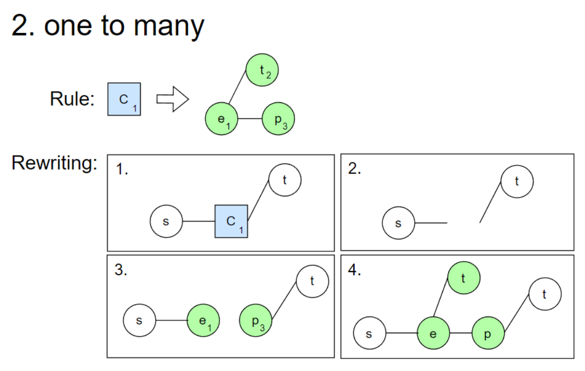
The second case (see Fig. 3.12b) is also a context-free case where a single ”match” node is replaced by a ”mutation” graph. This process involves designating the node with the lowest number in the ”mutation" graph as the "start" node, and the node with the highest number as the "end" node. Next, any incoming edges connected "match" node are transferred to the designated "start" node in the mutation graph. Similarly, any outgoing edges connected to the "match" node are transferred to the designated "end" node in the mutation graph. Finally, the "match" node is removed from the host graph, and the nodes and edges of the mutation graph are added to the host graph.

The third case (see Fig. 3.12c) is a context-sensitive case where a "match" subgraph is replaced by a single "mutation" node. This process is done by first removing all edges between the "match" subgraph nodes. Then transfer all other edges, that connect the "match" subgraph to the rest of the host graph, to the "mutation" node. Finally, the "match" subgraph nodes are removed from the host graph, and the mutation node is added to the host graph.

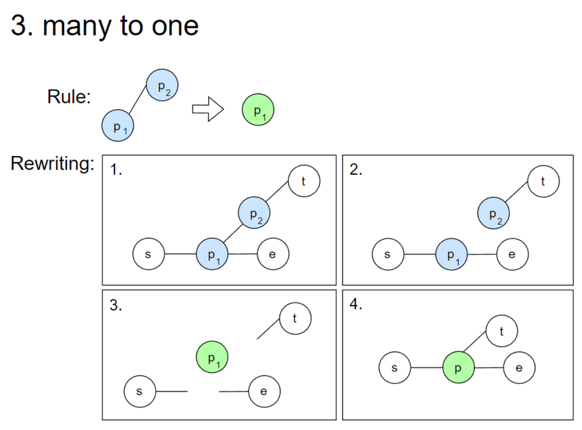
The fourth case (see Fig. 3.12d) is also a context-sensitive case where a "match" subgraph is replaced by a "mutation" graph. To accomplish this, first, all edges between the "match" subgraph nodes are removed. Then, each node in the "match" subgraph, which is linked to a node in the "mutation" graph by having the same alias number, has its adjacent edges transferred to that linked "mutation" node. Finally, the "match" subgraph nodes are removed from the host graph, and all nodes from the "mutation" graph are added to the host graph, along with any remaining edges.



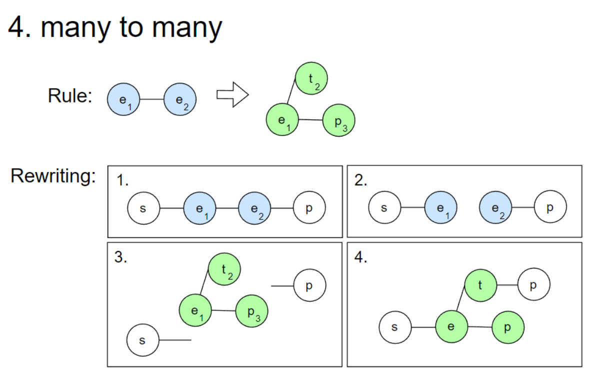
(a)



(b)



(c)



(d)

Figure 3.12: Four cases of graph rewriting

Choosing rules and where to apply them

When evolving a mission graph, three crucial decisions must be made: which rule to apply, where to apply it, and which RHS of the rule to use. All of these decisions will affect the way the mission is created, and as such there should be a system that allows the developer to give some direction in the algorithm choices. This can be achieved in a multitude of ways, rules can have a priority associated with them just like the RHSs of the composite rules have, or an evaluation function could take in parameters defined by the developer and classify each possible change and choose the one that as a better fit. The latter option was tested in the prototype created for this study, but it was found to be a bit slow and in need of optimization, so it was eventually compromised to generate multiple mission maps using a faster method by making the choice of which rule to use random and apply it on the first subgraph match found in the host graph by the searching algorithm, with the acRHS of the rule chosen based on its priority attribute. The evaluation algorithm then classifies the different mission maps using the parameters defined by the developer to select the best candidates to proceed to the next step and be transformed into space maps.

Validating mission maps

The mission map validation is done by scoring the map according to a few parameters defined by the developer to fit the style and feel of the game being created. The definition of these parameters also contains a way to evaluate and score the map. In this study's prototype the mission maps were scored taking into account the following parameters:

- **Level size**: used to influence the level scoring based on the number of platform nodes in the mission graph. Missions with more platforms will create large levels. It is worth noting that the size of a level is also linked to the number of rules that can be applied to the level, with the grammar developed for this prototype each time a rule is applied the number of nodes in the graph increases.
- **Graph linearity**: used to influence the level scoring based on the number of edge slopes that are different from zero. A higher value means that there are more side paths and parallel paths in the mission map.
- **Number of enemies**: used to influence the level scoring based on the number of enemy nodes on the mission map.
- **Number of coins**: used to influence the level scoring based on the number of coins nodes on the mission map.

All the mission maps created are scored based on the sum of these parameters then the m top maps are selected to be translated into space maps.

3.2.3 Step two: creating the space layout

The space map is a graph that represents the infrastructure of the level, each node represents a level element with width, height, position and other properties that the element might need, for instance, a moving platform would also have the direction and distance it travels. The edges of this graph represent the intended paths a player might take to go between these elements, however, they do not represent all possible paths, for example, a node representing a platform might be on a different branch of the graph representing a place height above another platform, the player can still fall to platform bellow even though they were not linked in the space map graph. From the mission map represented in Fig. 3.5 the following space map would be created 3.13.

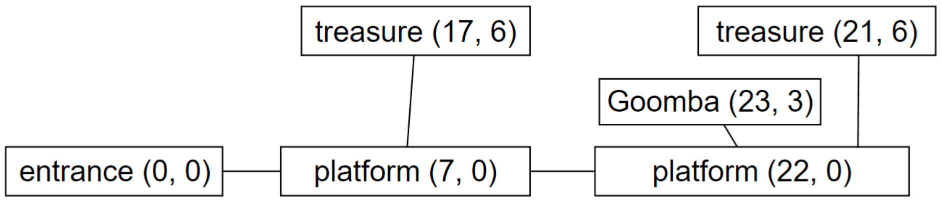


Figure 3.13: A space map representation of the start of World 1-1 in *Super Mario Bros.*

The space map is created by going through a mission map graph using a breadth-first search (BFS) traversal algorithm, which means that starting from the root node all neighbouring nodes are explored first. Using this search method is advantageous since the level is being built in a specific direction, and we can evaluate the pieces that make up each section of a level one step at a time. It also facilitates the generation of a level at runtime since the mission map could be segmented and only one segment be created until the other is needed. At each step of the mission map BFS, the current node or parent node and its adjacent out edges or child nodes are translated into prototype structures and all the necessary parameters of these structures are calculated. As explained, a prototype structure represents a template for the actual element to be used, it contains the basic properties of the element like position, size and any other special property the element might have. There are two types of prototype structures: placeable structures and normal structures. Placeable structures are elements that must be placed on top of other elements, possible examples are levers, traps and enemies. Normal structures do not have to be put on top of elements, examples would be platforms or the level's entrance and exit. It is important to highlight that if a mission node happens to be associated with more than one prototype structure, a strategy for selecting one should be specified. As the parent and children nodes are being translated the edges that connect them on the mission map are also checked to see if they have a special type that dictates relations between the nodes. For example, the connection between a key and the lock it opens, if an edge of this type is found, then both the prototype structure of the key and the lock would have an extra property to hold this connection.

After having translated both parent and children nodes, calculations are made to find their position in the map, the position of the parent structure is already known either because it was calculated in a previous step or because it is the initial default position. The child positions are calculated in relation to the parent position with the help of a player simulation and depend on the player jump distance, on the slope of the edge that connects the respective child mission node to the parent mission node and where or not a connection prototype structure is going to be placed between them. A connection node is a special prototype structure that connects two impossible-to-reach elements without blocking the player's movement, examples include, moving horizontal and vertical platforms, ladders that the player can climb by pressing up and walk through by pressing left or right, a static platform that the player can cross but also fall or jump through if he wants, a rope that the player can climb horizontally but go through vertically, and so on. Normally, for a position to be legal, the player must be able to reach it, this means that new elements cannot be placed further apart than the maximum distance the player simulation can jump both vertically and horizontally (see Fig. 3.14a). However, connection structures allow the elements to be put further apart while still ensuring that the player can reach them (see Fig. 3.14b).

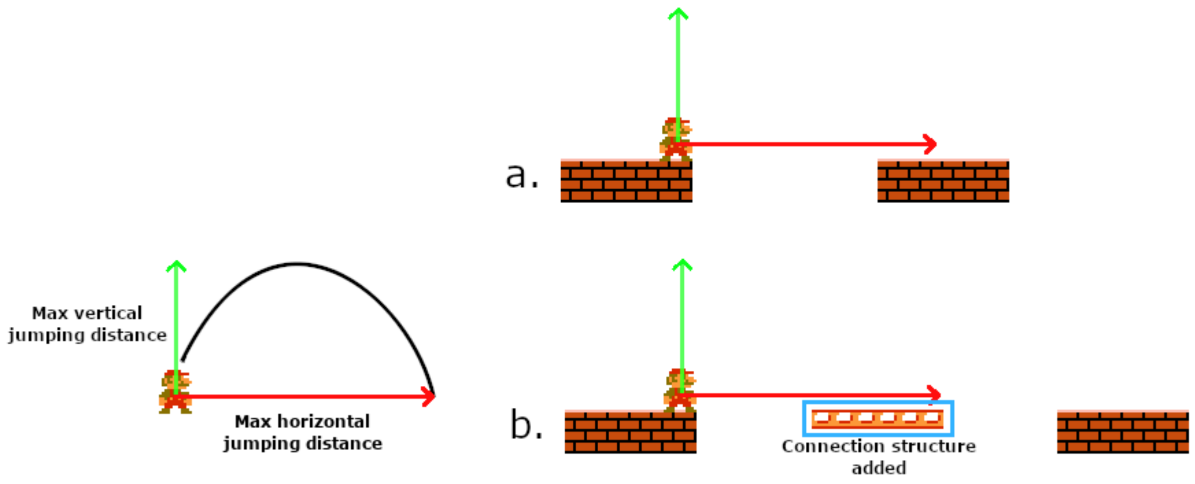

a.
Max vertical jumping distance
Max horizontal jumping distance
b.
Connection structure added

Figure 3.14: A visual representation of how element positions are determined with and without connection structures (*Super Mario Bros.* sprites adapted from www.nesmaps.com)

When calculating the positions for prototype structures, it is important to check if the area the new structure will occupy already contains another structure, either completely or partially inside it. If it does, there are at least three solutions: either move the other structures to make room, skip and don't include the element being placed or while no valid place is found shift the element's placement to the end of the map by altering its parent mission node to another node in the mission graph that has already been translated and placed the farthest in the direction the level is being created and then recalculating the new position. The last option was the one used in this study's prototype, it is basically a trial and error strategy to find a valid place for the new element. A space map is complete once all of the nodes in the mission map have been translated to prototype structures, assigned a place and all other parameters they might contain have been determined. The next step is to validate it.

Validating space maps

The space map validation is done by scoring the map according to a few parameters, similar to the mission map, these parameters can be defined by the developer to fit the style and feel of the game being created. In this study's prototype the space maps were scored taking into account the following parameters:

- **Distance to goal**: the bigger the euclidean distance from the entrance to the goal of the level the better, this helps prevent levels that have the start and goal right next to each other;

- **Number of collisions**: the number of times the calculated position of an element had to be shifted because there has already an element in that area. Levels that have a low collision count are preferable because every collision essentially changes the original mission map layout and can create graphs that don't follow the mission grammar syntax;
- **Length of the player path**: essentially the length of the shortest path from the entrance to the goal. The euclidean distance indicates how far apart these two points are, but it does not consider the obstacles along the way, so it is important to check if an actual path exits and how long this path is.
- **Feasibility**: if the simulated player can go through the level's shortest path from the entrance to the goal without being blocked. This is accomplished by traversing the path's nodes, when a node represents any type of platform it is inspected to determine whether the player can cross the platform from one side to the other without colliding with anything marked as an obstacle. It is also verified whether the player can reach the next node in the path by assessing whether the next node element is within an estimated player jumping range, which is obtained by calculating the player's jump trajectory.

Other parameters that were not used in the implementation of this study's prototype but are worth considering when creating levels with better gameplay are the number of side paths and the number of dead-end branches. Favouring levels that have more than one way to reach the end goal is beneficial because it provides the player with choice and motivates exploration. Reducing the number of pointless dead-ends in the level improves the quality of the exploration as well. Since the player is not punished by having to backtrack empty-handed when exploring a side path. Both of these aspects are also affected by how the mission grammar is designed, production rules can be written to replace side branches that terminate in empty node leaves with treasure and challenges, or they can be written to avoid side paths entirely.

During the level generation process, to improve the likelihood of generating the most appropriate map according to the parameters defined. Several space maps are built from various mission maps and then rated, the space map with the greatest score is chosen to build the level.

3.2.4 Step three: building the level

The final step is to build and draw the actual level. This depends on the game engine used to program the game, but it essentially involves going through the space map and transforming the prototype structures into actual game objects, taking into account any special connections between them, and translating those connections to the game logic.

3.3 Summary

In this chapter, we analysed and discussed some of the potential challenges that come with creating a framework software capable of generating playable 2D platformer games. Specifically, we discussed how the game's engine gravity affects the movement of the player, by reducing control of the playable character when the player jumps or falls. This, in turn, affects how the layout of the level should be planned since the movement of the player is restricted. Next, we mention how creating a framework that can generate any type of 2D platformer game is a daunting task and that at some point compromises have to be made, otherwise the framework would have to be too generic leaving too much work and responsibility to the developers using it. The last topic discussed was the level of control that developers should have over the generation process in level design. Finding the right balance between complete control and no control is key. A generator using graph grammars can give the developer the ability to guide the level layout, by tweaking and changing the grammars rules, while still keeping a level of randomness in the generation process. This randomness can be managed through priority values and custom parameters, resulting in a generator that provides both a hands-off and a somewhat controlled approach to generating levels.

After the problem analyses, we presented a possible design for the framework. We discussed how Joris Dormans' work on generating action-adventure games forms the bases of this study's framework design. Following his work, the generation of 2D platformer levels was divided into Three steps. These are, deciding the goals and challenges of the level, creating the appropriate layout for the level, and finally, constructing the level. To do this two structures were created: A mission map, that contains a graph with the goals of the level. This mission graph is generated by applying grammar rules using a generative graph grammar. And a space map, which also contains a graph, but with the prototype structures that will be converted into actual game objects, these structures also have their positions on the level. During the generation process, multiple mission maps and space maps are generated and evaluated, and the best scoring space map is the one used to build the level. Next, we will discuss how this method was used to create a prototype game.

Chapter 4

Implementation

A small prototype game called ”*2D Platformer Generator*” was developed as a proof of concept following the previous section’s proposed design, the prototype was made using the *C#* language in the *Unity* engine [19]. *Unity* was chosen because it provides an out-of-the-box game engine with a very friendly learning curve, which made it easier to create the game aspects of the prototype. Using *Unity* also provided access to the *Unity Store* [20] which is a great source of resources to build games. Building the framework using the object-oriented language *C#* also provided access to features such as inheritance and reflection that helped separate the generating framework from the actual game engine. This chapter holds a detailed description of the architecture and logic used to develop this prototype software, with a special focus on the level generation logic since the other parts of the software deal with Unity and gaming aspects that go beyond this project spectrum.

The prototype’s architecture (see Fig. 4.1) is divided into four layers: the grammar engine; the generation engine; the validation engine; and the game engine. The grammar engine is responsible for loading the grammar production rules and all processes that have to do with the application of the grammar. The validation engine is responsible for all simulation, validation and scoring of mission maps and space maps. The generation engine takes care of generating the mission maps and space maps using the previous two layers. And finally, the game engine serves as a placeholder where the developer would hold the logic for the specific game, using the generation engine to create the levels.

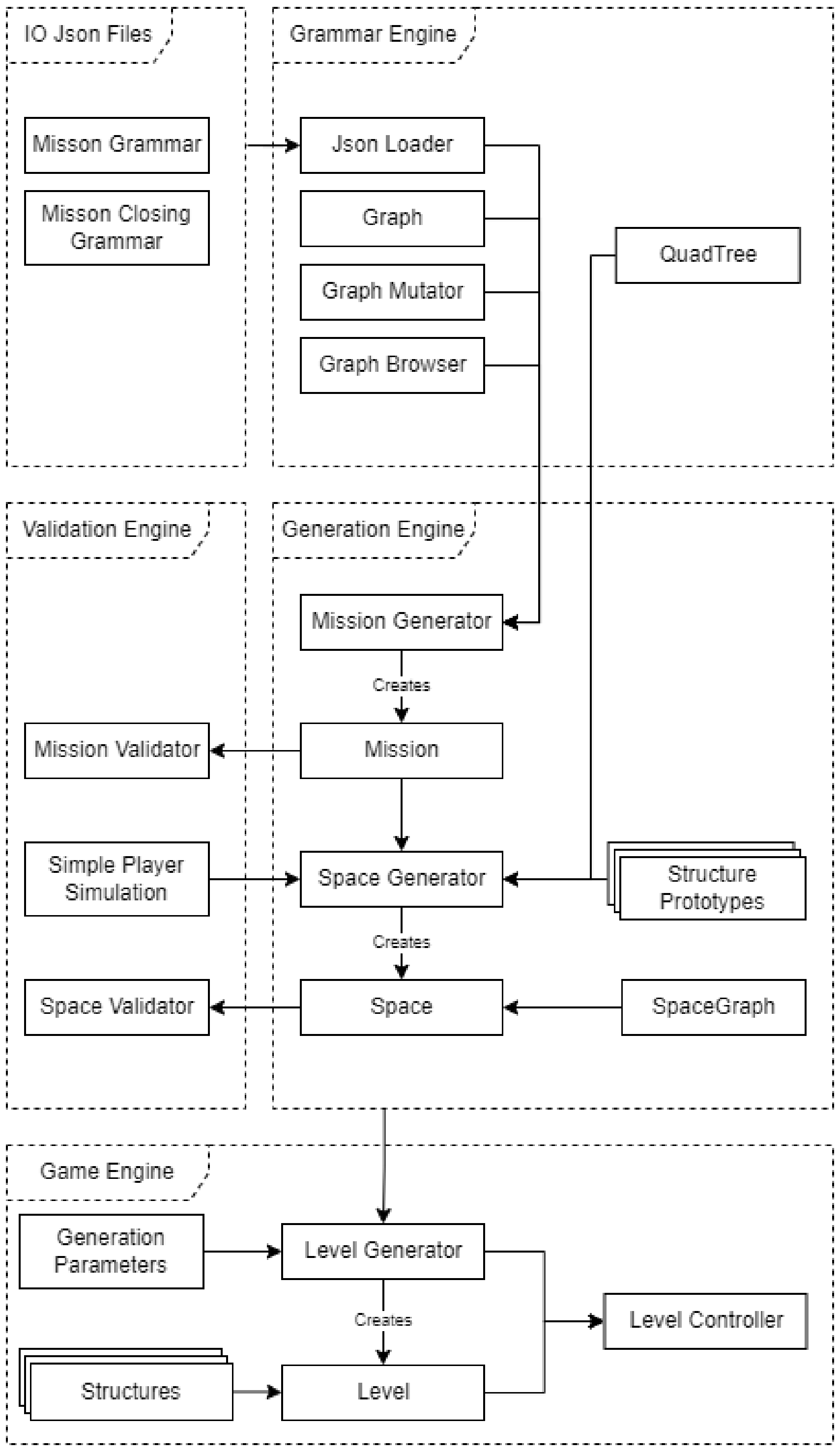


Figure 4.1: The prototype's architecture diagram

4.1 Configuration

There are a couple of configurations and external files associated with the generator. One such file is the engine configuration file, which contains the path to the two mission grammar external files needed to create mission maps and it also includes the name reference of the mission and space generation strategies classes that will be utilized during the level generation process. There is also a class called "*GeneratorParameters*" that holds all the parameters needed during the stages of level generation.

4.2 Grammar engine

The grammar engine function as the core system for loading graph grammars, allowing the creation and transformation of mission maps using the grammar's rules. As such, it functions as a graph rewriting system. This type of system is responsible for providing a way to create production rules, recognise where a production rule can be applied, choose what rule would make the best modification to the graph and finally apply the production rule by changing the graph according to the rule. In this section, we will first take a look at some of the tools already available for graph rewriting and why they were not used in this project, followed by an explanation of how the grammar engine works.

4.2.1 Available graph rewriting tools

Over the years many graph rewriting tools have been developed. In 2002, David Adams in his work [17] referenced two tools. They are, the *PROgrammed Graph REwriting Systems (PROGRES)* [21], a specification language that allowed the manipulation of graphs. And the *Attributed Graph Grammar (AGG)*[22], a development environment written in *Java* that supports an algebraic approach to graph transformation. More recently, the researchers at UT Austin have developed the tool *GraphSynth*[23], written in *C#* the tool offers a way to transform a graph using graph grammar rules, storing these rules and graphs in a common XML framework. Another recent tool is the *Graph Grammar Library (GGL)*[24] an object-oriented *ANSI C++* library that implements DPO approach and uses state-of-the-art algorithms to implement and apply graph rewrite systems and offers great support for grammar-based graph transformation in chemistry.

For this project, however, it was opted to create a custom system since the tools mentioned above are either too old or have too many features. The *PROGRES* was created in 1995 and as such its not easy to find references or support for its use, *AGG* is still being updated with the last patch being from 2021, but it is written in *Java* and does not provide that much control over the way rules are applied, only *GGL* and especially *GraphSynth* could have potential being used for this project. However, a simpler system with more control over the graph matching and replacement methods was preferred. The rest of this section will focus on explaining how this custom system developed works.

4.2.2 Defining a mission grammar

For this prototype, mission grammars are defined in external *Json* files, *Json* format was chosen because of its common use in the industry, meaning that many languages have support to parse these types of files. Since this is an initial testing prototype the rules have to be written in *Json* format manually, of course, this is not practical so ideally, they would be automatically parsed from a production rule creation GUI or a functional language like Standard ML. Appendix A.4 shows an example of a composite production rule defined in *Json* format. Every rule is composed of an id, a left-side graph and a list of right-side graphs. Each graph has a list of edges and nodes where nodes have a type and an alias and edges have a source node, a target node, a type and a slope. The right-side graphs also have a priority value and a list of associated rules. The mission grammar designed and used in this prototype is called grammar A and can be seen in appendix A.1. A secondary grammar known as closing grammar was also created, this grammar is an exact copy of grammar A but it does not contain any direct or indirect recursive rules. Illustrated in Fig. 4.2) is two examples of recursive rules.

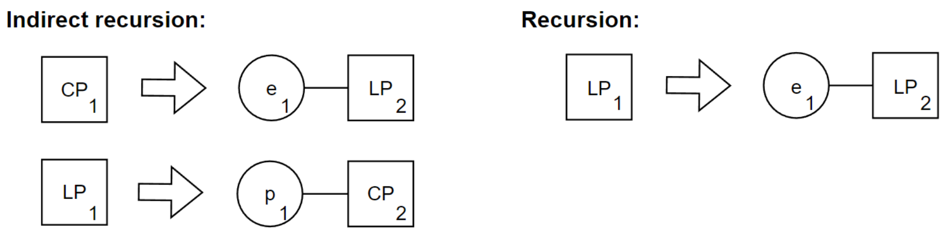


Figure 4.2: Looping rules example

Without proper validation or checks, recursive rules can create endless loops of rules being applied. Therefore, creating a grammar without recursive rules guarantees that a graph that still contains non-terminal symbols will be transformed into a graph with only terminal symbols as fast as possible while still respecting the original grammar syntax. Both of the grammars created contain the following terminal symbols:

- **start**: where the player spawns in the level;
- **goal**: the end of the level;
- **platform**: any empty platform or ground the player can stand on;
- **enemy**: any adversary non-player character (NPC);
- **coins**: a group of collectable coins;

4.2.3 Custom graph rewriting and grammar system

The grammar engine layer contains the core components of the grammar system, as well as a couple of interfaces that help with the use of the grammar system. One such interface is the *IGrammarLoader*, which helps load the *Json* files by proving means of parsing the *Json* rule data and transforming them into rule objects. This process can be better understood by looking at Fig. 4.3 which shows a class diagram of the grammar loading system. Following the diagram it is possible to see that a *Grammar* class contains an *IGrammarLoader* implementation called *JsonGrammarLoader*, this loader reads in an input *Json* rule file and converts the *Json* data into a list of *CompositeRule* objects that implements an *IGrammarRule* interface. Every *CompositeRule* contains one left side and multiple right sides, each side containing a graph. The graphs are represented using an external library called *QuikGraph* [25], this class provides methods to deal with all the internal graph operations, to keep this dependency to a minimum the class was wrapped in our own graph class.

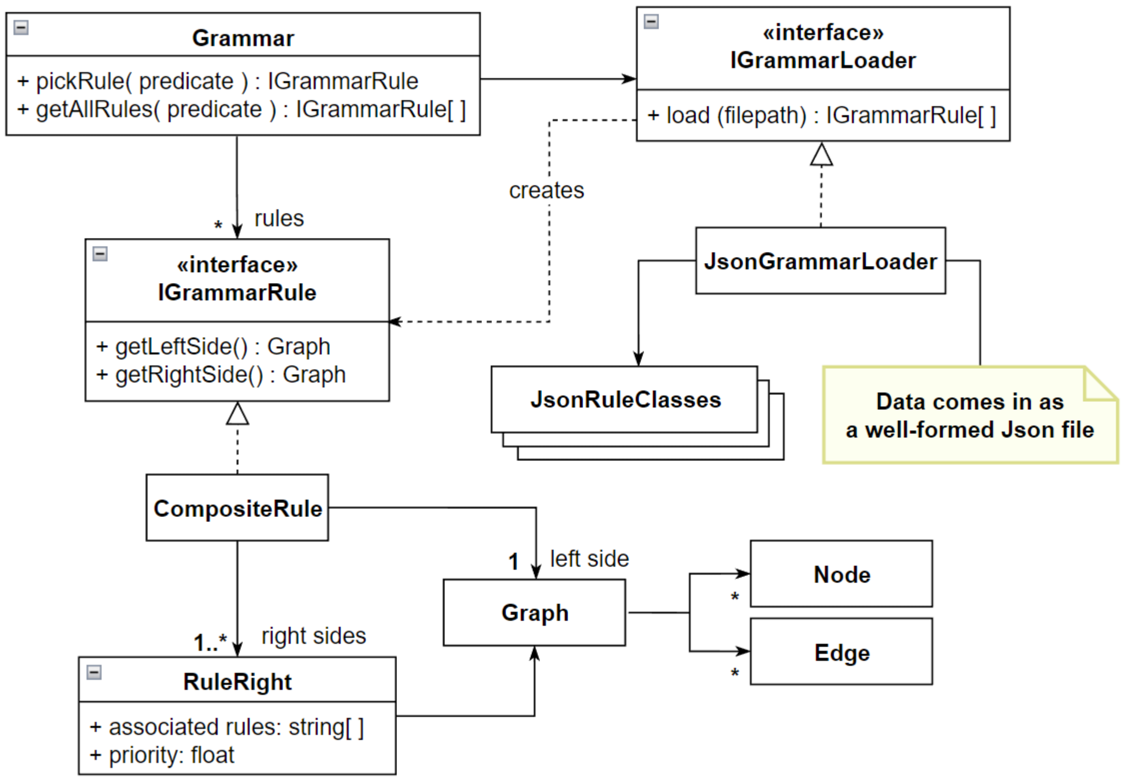


Figure 4.3: A class diagram of the grammar loading system contained in the grammar engine

Search graphs for rule patterns

This layer also provides a *IGraphBrowser* interface to search graphs for rule patterns, in the context of the prototype created this interface was implemented by a class called *GraphBrowser*, which makes use of a matching strategy interface called *IGraphMachingStrategy* to find subgraph patterns in graphs, this interface is then implemented by a class called *UllmanMatchingStrategy* that uses Ullmann's algorithm (see section 3.2.2 - Searching for applicable rules) to find pattern matches. A class diagram of these classes can be seen in Fig. 4.4.

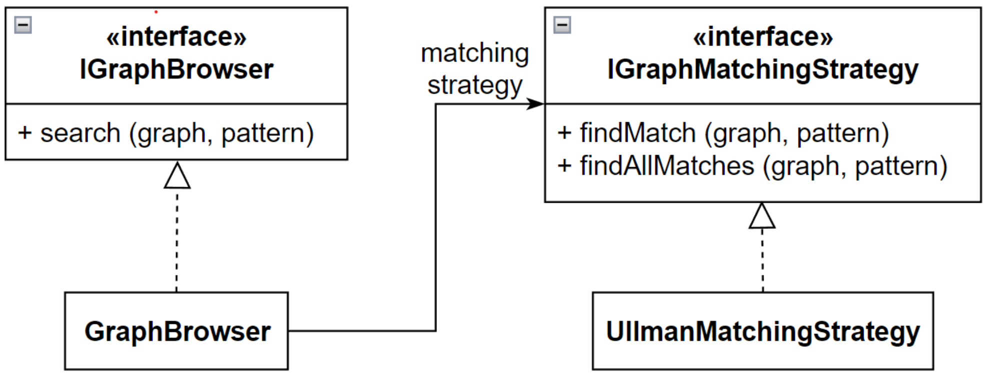


Figure 4.4: A class diagram of how the *IGraphBrowser* is implemented

The *UllmanMatchingStrategy* class implements a way to search through the host graph to find subgraphs that match with the rules LHS to discover what rules can be used and where they can be applied. The pseudo-code for the implementation of Ullmann's algorithm used can be seen below.

Algorithm 1 Ullman's subgraph isomorphism algorithm with pruning

```
1: procedure ULLMANSALGORITHM($hostGraph, patternGraph$)
2:     if $|V(patternGraph)| > |V(hostGraph)|$ then
3:         return
4:     end if
5:     $M^0 \leftarrow$ correspondenceMatrix($hostGraph, patternGraph$)
6:     subgraphs $\leftarrow$ permutationMatrices($M^0$)
7:     return subgraphs
8: end procedure
```

The algorithm takes two input graphs, $hostGraph$ and $patternGraph$, and returns a list of subgraphs of $hostGraph$ that are isomorphic to $patternGraph$. It starts with a check to make sure that the number of nodes in the pattern graph is not greater than the number of nodes in the host graph, $V()$ is a function that returns the nodes of an input graph. If the pattern graph has more nodes then there is no possible subgraph inside $hostGraph$, and the algorithm returns immediately. Next, a correspondence matrix, M^0, is created using the $correspondenceMatrix()$ function, which takes the host and pattern graphs as inputs. The pseudo-code for this auxiliary function can be seen below.

Algorithm 2 Creation of a correspondence matrix and applying pruning

```
1: procedure CORRESPONDENCEMATRIX($hostGraph, patternGraph$)
2:     $m \leftarrow |V(hostGraph)|$
3:     $n \leftarrow |V(patternGraph)|$
4:     $M^0 \leftarrow$ zeros($n, m$)
5:     for all $Vpattern \in V(patternGraph)$ do
6:         for all $Vhost \in V(hostGraph)$ do
7:             $isValidCandidate \leftarrow |Adj(Vhost)| >= |Adj(Vpattern)|$
8:                                 and $Type(Vhost) == Type(Vpattern)$
9:             $j \leftarrow index(Vpattern)$
10:            $i \leftarrow index(Vhost)$
11:            $M^0[j, i] \leftarrow (isValidCandidate)$ ? 1 : 0
12:        end for
13:    end for
14:    pruning($M^0, hostGraph, patternGraph$)
15:    return $M^0$
16: end procedure
```

The purpose of $correspondenceMatrix()$ is to create a correspondence matrix between the nodes of the two graphs. The function starts by initializing two variables, m and n, which represent the number of nodes in the host graph and the pattern graph, respectively. The matrix M^0 is then created with dimensions $n \times m$ and all elements are initialized to 0. The function then uses nested for loops to compare every node in the pattern graph with every node in the host graph. The $isValidCandidate$ variable is set to true if the number of adjacent nodes in the host graph node is greater than or equal to the number of adjacent nodes in the pattern graph node and if their types are the same. The indices of the nodes in the pattern and host graphs are stored in variables i and j, respectively. The matrix M^0 is then updated with a value of 1 at the i^{th} row and j^{th} column if the candidate is valid, and 0 otherwise. Finally, the function applies pruning on the matrix M^0, which is a process of simplifying a matrix by eliminating invalid candidates. The purpose of this is to reduce the amount of computation needed. The pruning process works by observing that if a node $A \in hostGraph$ is a candidate to a node $B \in patternGraph$, then an adjacent node to A should also be a candidate to an adjacent node to B. If this is not the case, then it is an invalid candidate and the value in the matrix should be changed from 1 to 0.

Returning to Ullman's algorithm, the next step after building the correspondence matrix is to check every possible permutation of the correspondence matrix to identify valid subgraphs. The *permutationMatrices()* function traverses the matrix creating a tree with all possible subgraphs candidates, these candidates are then evaluated to determine if they satisfy the criterion for being isomorphic to $patternGraph$. The function adds any valid subgraph to a list, which is built up as the algorithm continues to test every possible permutation of the matrix. The algorithm demonstrated in the pseudo-code below is an example of the *permutationMatrices()* function.

Algorithm 3 Checks every permutation on the correspondence matrix to find valid subgraphs

```
1: procedure PERMUTATIONMATRICES($M, row, validSubgraphs$)
2:     if $row >= |rows(M)|$ then                                  ▷ Check if current row is the last one
3:         if satisfiesIsomorphicCriterion($M$) then
4:             add $M$ to $validSubgraphs$
5:         end if
6:     end if
7:     for $col = 0$ to $|cols(M)|$ where $M[row, col] == 1$ do
8:         $isValid \leftarrow true$
9:         if there is a $y \neq row$ where $M[y, col] == 1$ then
10:            if there is no $x \neq col$ where $M[y, x] == 1$ then
11:                $isValid \leftarrow false$    ▷ The same node cannot be a candidate for two different nodes,
12:                                              ▷ and the second node does not have any other candidates
13:            end if
14:        end if
15:        if $isValid$ then
16:            $copyM \leftarrow copy(M)$
17:            for every $x \neq col$ set $copyM[row, x] = 0$
18:            permutationMatrices($copyM, row + 1$)                 ▷ Recursion call
19:        end if
20:    end for
21: end procedure
```

As mentioned above, the purpose of this function is to check for every permutation on the correspondence matrix to find valid subgraphs. This is a recursive function that implements backtracking to construct a tree of all possible arrangements of the correspondence matrix. Each branch in the tree corresponds to the construction of a permutation matrix, this type of matrix is one where: $\forall i \in 1, 2, \ldots, m, \sum_{j=1}^{n} M[i, j] = 1$ and $\forall j \in 1, 2, \ldots, n, \sum_{i=1}^{m} M[i, j] = 1$, where m and n are the dimensions of the matrix M. Any branches that do not result in a leaf node with a valid permutation matrix are rejected and not considered. Using the function *satisfiesIsomorphicCriterion()*, each created leaf permutation matrix M' is checked to see if satisfies the isomorphic criterion $P = M'AM'^{-1}$, where

P is the adjacency matrix of $patternGraph$. If it does, the matrix is added to the list of valid subgraphs. The algorithm continues checking every permutation of the matrix until all possible permutations have been checked, and the list of valid subgraphs has been built up.

To speed up the application of rules, a modified version of Ullman's algorithm was used. Instead of finding all possible subgraphs, this modification returns the first valid subgraph it finds and then stops. This way, the process of creating a permutation tree stops as soon as the first valid subgraph permutation matrix leaf is found. To avoid repeatedly selecting the same first subgraph that is found on the tree, the correspondence matrix can be created by randomly shuffling the node order in both graphs. This scrambles the information, causing the tree to construct the permutation matrixes in a different order. In addition to randomizing the node order, an alternative approach to finding different subgraphs is to begin building the tree from different columns in the first row. This reduces the likelihood of repeating the same subgraph selection as well, leading to a more diverse set of results.

Morphing graphs

In addition to the previous interfaces, the grammar engine layer also provides the *IGraphMutator* interface, which enables the system to morph graphs based on the identified grammar rules. When a rule is found by the grammar engine layer, the *IGraphMutator* interface is used to perform the required morphing of the graph. This morphing process involves substituting one subgraph pattern in the given input graph for another pattern. The *AdaptiveGraphMutator* is a class implementation of the *IGraphMutator* interface, which utilizes one of the four algorithms outlined in section 3.2.2 (Graph Morphing) to perform graph morphing. As described in the previous section, this strategy involves analyzing the subgraph to be replaced and the graph replacing it to determine the type of substitution that is required. This analysis takes into account the number of nodes in each graph, to ensure that the appropriate morphing action is taken.

The simplest case is to replace a single node with another node, the algorithm below represents this case:

Algorithm 4 One-to-one graph morphing

```
1: procedure OneToOneMorph($host, matchNode, mutationNode$)
2:     add $mutationNode$ to $host$
3:     for all $edge \in E(matchNode)$ do        ▷ $E()$ return all adjacent edges of the input node
4:         $newEdge \leftarrow$ copyEdgeTo($edge, mutationNode$)
5:         add $newEdge$ to $host$
6:         remove $edge$ from $host$
7:     end for
8:     remove $matchNode$ from $host$
9: end procedure
```

This is a simple substitution algorithm. Given a host graph, a matching node from the host graph, and a mutation node, the algorithm adds the mutation node to the host graph, copies all the edges from the matching node to the mutation node, and removes the matching node from the host graph. More specifically, the algorithm iterates through all adjacent edges of the matching node in the host graph and creates new edges between the mutation node and the other nodes in the host graph that the matching node was connected to, taking into consideration the direction of each edge. It then removes the original edges between the matching node and the other nodes in the host graph. Finally, it removes the matching node itself from the host graph. This effectively ”morphs” the host graph by replacing the matching node with the mutation node, while maintaining the connectivity of the graph.

Another case is to replace a single node with a graph, for this, the following algorithm is used:

Algorithm 5 One-to-many graph morphing

```
1: procedure ONETOMANYMORPH($host, matchNode, mutationGraph$)
2:     $first \leftarrow$ start($mutationGraph$)    ▷ returns the node marked as first from the input graph
3:     $last \leftarrow$ end($mutationGraph$)    ▷ returns the node marked as last from the input graph
4:     add $first$ to $host$
5:     add $last$ to $host$
6:     for all $edge \in E(matchNode)$ do
7:         if $edge$ is an inbound edge then
8:             $newEdge \leftarrow$ copyEdgeTo($edge, first$)
9:         end if
10:        if $edge$ is an outbound edge then
11:            $newEdge \leftarrow$ copyEdgeTo($edge, last$)
12:        end if
13:        add $newEdge$ to $host$
14:        remove $edge$ from $host$
15:    end for
16:    add all other $mutationGraph$ nodes to $host$
17:    add all $mutationGraph$ edges to $host$
18:    remove $matchNode$ from $host$
19: end procedure
```

The main concept of this algorithm is to categorize the adjacent edges of the matching node into inbound edges and outbound edges. This way, the first node in the mutation graph can have a copy of the inbound edges, and the last node can have a copy of the outbound edges. The first and last nodes are labelled as such in the creation of the rule. More specifically, the algorithm takes in a host graph, a matching node from the host graph, and a mutation graph as input, and starts by adding the first and last nodes of the mutation graph to the host graph, then using the same method as the last algorithm it copies all inbound edges from the matching node to the first node of the mutation graph and all outbound edges from the matching node to the last node of the mutation graph. Finally, it adds all other nodes and edges of the mutation graph to the host graph and removes the matching node from the host graph. Effectively replacing the match node in the host graph with the mutation graph and its edges.

On the other hand, if the morphing process involves replacing a subgraph with a single node, then the following algorithm is applied:

Algorithm 6 many-to-one graph morphing

```
1: procedure MANYTOONEMORPH($host, matchGraph, mutationNode$)
2:     remove all edges between nodes of $matchGraph$ from $host$
3:     add $mutationNode$ to $host$
4:     for all other edges in $matchGraph$ do
5:         $newEdge \leftarrow$ copyEdgeTo$(edge, mutationNode)$
6:         add $newEdge$ to $host$
7:         remove $edge$ from $host$
8:     end for
9:     remove all $matchGraph$ nodes from $host$
10: end procedure
```

The goal of the algorithm is to transform the host graph by replacing the match subgraph with the mutation node. It takes in a host graph, a matching subgraph of the host graph, and a mutation node as input, and it starts by removing all edges between the nodes of the match subgraph from the host graph. It then adds the mutation node to the host graph. Next, it iterates over all the remaining edges in the match subgraph, and for each edge, it copies the edge to the mutation node, again respecting the edge's direction. It adds these new edges to the host graph and removes the original edges from the host graph. Lastly, it removes all nodes of the match subgraph from the host graph, effectively replacing the match subgraph in the host graph with the mutation node.

Finally, in case the replacement includes substituting a subgraph with another graph, then the following algorithm is applied:

Algorithm 7 many-to-many graph morphing

```
1: procedure MANYTOMANYMORPH($host, matchGraph, mutationGraph$)
2:     remove all edges between nodes of $matchGraph$ from $host$
3:     add all $mutationGraph$ nodes to $host$
4:     for all $matchNode \in matchGraph$ do
5:         for all $mutationNode \in mutationGraph$ do
6:             if alias$(matchNode)$ = alias$(mutationNode)$ then
7:                 for all $edges \in matchNode$ do
8:                     $newEdge \leftarrow$ copyEdgeTo$(edge, mutationNode)$
9:                     add $newEdge$ to $host$
10:                    remove $edge$ from $host$
11:                end for
12:            end if
13:        end for
14:    end for
15:    remove all remaining $matchGraph$ edges connecting to $host$
16:    remove all $matchGraph$ nodes from $host$
17:    add all $mutationGraph$ edges to $host$
18: end procedure
```

As mentioned in section 3.2.2 (Graph Morphing) the algebraic SPO graph rewriting approach requires context elements between the two sides of a rule, as such when a rule is created the nodes of both sides of the rule are labelled numerically and context elements are defined by giving a node on the RHS graph the same number as the one on the LHS, these nodes are also known as pivot or anchor nodes.

With this labelling, the algorithm takes in a host graph, a matching subgraph of the host graph, and a mutation graph as input, and begins by removing all edges between the nodes of the match subgraph from the host graph. Then, it adds all nodes of the mutation graph to the host graph. Next, the algorithm checks if there are any pivot nodes by iterating each node in the match subgraph and trying to find a node on the mutation graph with the same label or alias. If so, it copies all edges in the match node to the mutation node using the same method as the previous algorithms, removing the original edges from the host graph. Lastly, the algorithm removes all nodes of the match subgraph from the host graph as well as all remaining edges connecting the match subgraph to the rest of the host graph and adds all edges of the mutation graph to the host graph. Completing the process and successfully morphing the hot graph.

In summary, the grammar layer provides the means of specifying graph transformation rules and applying them to graph structures. The interfaces mentioned, provide developers with the means to modify and refine these algorithms, providing the flexibility needed to create rules and transformations that are optimized for particular use cases and requirements. In essence, this allows for a more customizable and adaptable approach to graph rewriting, as developers can tailor the rules to suit the specific needs of their applications.

4.3 Generation engine

In this section, we will delve into the details of the engine responsible for creating both mission maps and space maps. On a higher level, this layer provides two interfaces, namely *IMissionGenerator* and *ISpaceGenerator*, both of which contain a *generate()* method that generates a mission map or space map accordingly, based on a set of parameters. These interfaces can be implemented by game developers to define their own *generate()* method, allowing developers to tailor the map generation to their specific needs. The rest of this section will specifically focus on how these interfaces were implemented for the prototype game developed for this study, we will dive deeper into the inner workings of this engine to understand how it can create these maps.

4.3.1 Mission map generation

The *IMissionGenerator* implementation developed for this study's prototype, uses a grammar-based approach to generate mission maps. The implementation class is called *MissionGenerator* and uses the grammar engine and the validation engine to create and validate mission maps. These maps are represented by a class called *Mission* that contains a graph. In Fig. 4.5, we can see the class diagram representing how the mission generation system is implemented.

The *MissionGenerator* class implementation of the *generate()* method involves initializing a host graph with a seed node, and then applying grammar rules until no more rules can be applied or the maximum allowed number of iterations is reached. Depending on the mode, the rules can be picked randomly or via a hill-climbing approach that searches for the best-fitting rule to apply. After the maximum number of iterations is reached, the non-terminal symbols in the mission map are transformed into terminal ones using a closing grammar. The resulting mission map is then returned as an object of the *Mission* class.

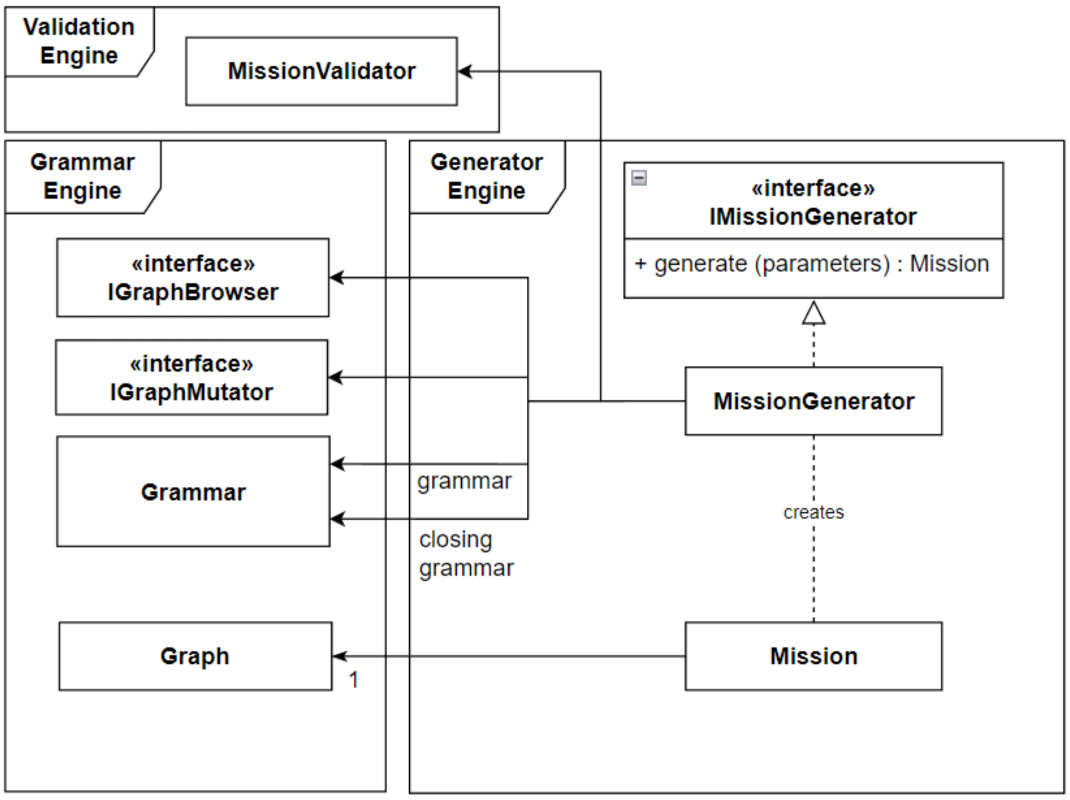


Figure 4.5: A class diagram of how the *IMissionGenerator* is implemented

Mission generator initialization and parameters

In order to generate missions using the *MissionGenerator*, at least one Grammar object, an *IGraphMutator* object, and an *IGraphBrowser* object are required. For this prototype, we utilized an *AdaptiveGraphMutator* and a *GraphBrowser*, as well as two grammars - a regular one utilized for the evolution process, and a closing one utilized to conclude the generation process. The mission map validations are done with a *MissionValidator* object. Also, it is important to note that the *generate()* method implemented takes in the following set of parameters that affect the mission map generation process:

- ***maxGenIterations*** specifies the maximum number of iterations allowed for the mission map generation process. If this limit is reached and the mission map still contains non-terminal symbols, the closing grammar is used to transform those non-terminal symbols into terminal ones.

- ***generationMode*** determines the mode of generation used, which can be either random or hill-climbing.

- ***useRulesPriority*** is a boolean value that indicates whether or not the generation process should prioritize picking the rule's RHS that has a higher priority.

- ***validationParameters*** contains a set of parameters (see section 3.2.2 - Validating mission maps) that are given to the *MissionValidator* in order to validate the generated mission map.

Generation process

In further detail, the generation process commences by initializing a host graph with a single seed node, which serves as the starting point for map generation. The process then proceeds by executing a loop that searches the host graph for applicable rules to expand and construct the mission map, until either no more rules can be applied or the maximum number of iterations allowed has been reached. During this loop, rules are picked based on the **generationMode** parameter.

If this parameter is set to ”Random”, the map generation process randomly selects a rule to apply from the set of applicable rules in the regular grammar. This is achieved through the use of the *IGraph-Browser* which searches for rules whose LHS is applicable to the current host graph. Once a suitable rule is found, the method selects a random subgraph match from the set of all subgraph matches in the host graph that can be mutated with the rule. If the selected rule has more than one RHS mutation, the **useRulesPriority** parameter determines how the mutation is picked. If this parameter is false, a mutation is picked at random. However, if this parameter is true, the mutation is selected based on the probability it was given. After the mutation is chosen, the *AdaptiveGraphMutator* is used to apply the mutation on the host graph replacing the selected match subgraph. This results in the expansion of the graph, contributing to the generation of the mission map.

On the other hand, when the *generationMode* parameter is set to ”HillClimbing”, the process employs a heuristic to select the most promising rule to apply based on the current state of the graph. Initially, for every applicable rule in the regular grammar, a corresponding list of the rule's RHS mutations is created. Subsequently, for that same rule, the *IGraphBrowser* finds all possible subgraph matches that are isomorphic to the rule's LHS pattern. Next, the process iterates through each combination of mutation and subgraph match, and using the *AdaptiveGraphMutator* applies the mutation in the subgraph match location on a copy of the host graph. The resulting graph is then scored using the *MissionValidator* according to the parameter values specified in the input parameters. This process is repeated for every applicable rule collecting the scores of all resulting graphs in a list. In the end, the graph with the highest score is selected and continues to the next step of the mission map generation.

This rule selection and application process is repeated until either no more rules can be applied, or the maximum number of iterations has been reached. At this point, if the host graph contains only terminal symbols the mission map is completed, otherwise the host graph is passed through a concluding step where the closing grammar rules are used to transform those non-terminal symbols into terminal ones. An example of a mission map can be observed in Fig. 4.6, additionally, appendix A.5 is an example of a mission map created with grammar A (see appendix A.1), the grammar used to generate mission levels in the prototype developed.

As it is possible to observe in Fig. 4.6, in the prototype developed every mission map starts with a node of type *start* and ends with a node of type *goal* that marks the end of the level. Every terminal node type in the mission alphabet is associated with one or more prototype structures, as explained in the last chapter these structures are templates for the actual game-level pieces that make up the level, so the $start\_1$ node would be associated with one or more structures that represented the start of a level like a door on top of a platform for example, the $platform\_2$ node would be associated with an empty platform or terrain formation, and so on or and so forth. Moreover, the slope of an edge between two nodes determines whether the prototype structure associated with that edge's target node is going to

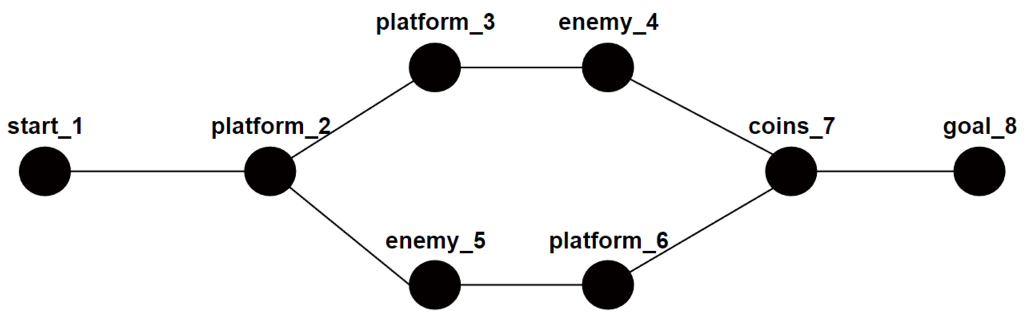


Figure 4.6: An example of a mission map generated

appear below, in front or above the previous structure in the level. This means that the mission map dictates what kind of game-level pieces will appear on the level as well as have an impact on the overall shape of a level. Therefore, the mission map is a high-level representation of the level being generated and plays a crucial role in the generation process. It ensures that the generated level follows the desired structure set by the game developer.

4.3.2 Space map generation

The *ISpaceGenerator* implementation for this study's prototype, takes in a mission map and generates a corresponding space map based on the mission requirements. The implementation is done with a class named *SpaceGenerator* which uses the services provided by the grammar engine and the validation engine to create and validate the space maps. These maps are represented with a class named *Space* which contains a graph called *SpaceGraph*. The decision to create a separate *SpaceGraph* class distinct from the normal *Graph* class used in mission maps was made to better organize and manage the information needed for the nodes in a space map, as they require more parameters than those in a mission map. While it is technically possible to redesign the *Graph* class to accommodate these differences, it would require additional time and effort that may not be practical or interesting for this study. Furthermore, separating the *SpaceGraph* class from the *Graph* class can help clarify the conceptual differences between the two types of maps and make the code easier to understand, maintain or modify as needed. The class diagram in Fig. 4.7 shows how the space map generation system is implemented.

The *SpaceGenerator* class implementation of the *generate()* method, involves translating a mission map into prototype structures and positioning them in the level based on the jumping capabilities of a player AI simulation and the slope of the adjacent edges of the mission node being translated, these prototype structures are then placed in space nodes on the space graph, the edges between the space nodes serve as the possible paths a player can take from node to node. Once all mission nodes have been translated and the structures have been given a location on the map, the space map is complete and is returned as a *Space* object.

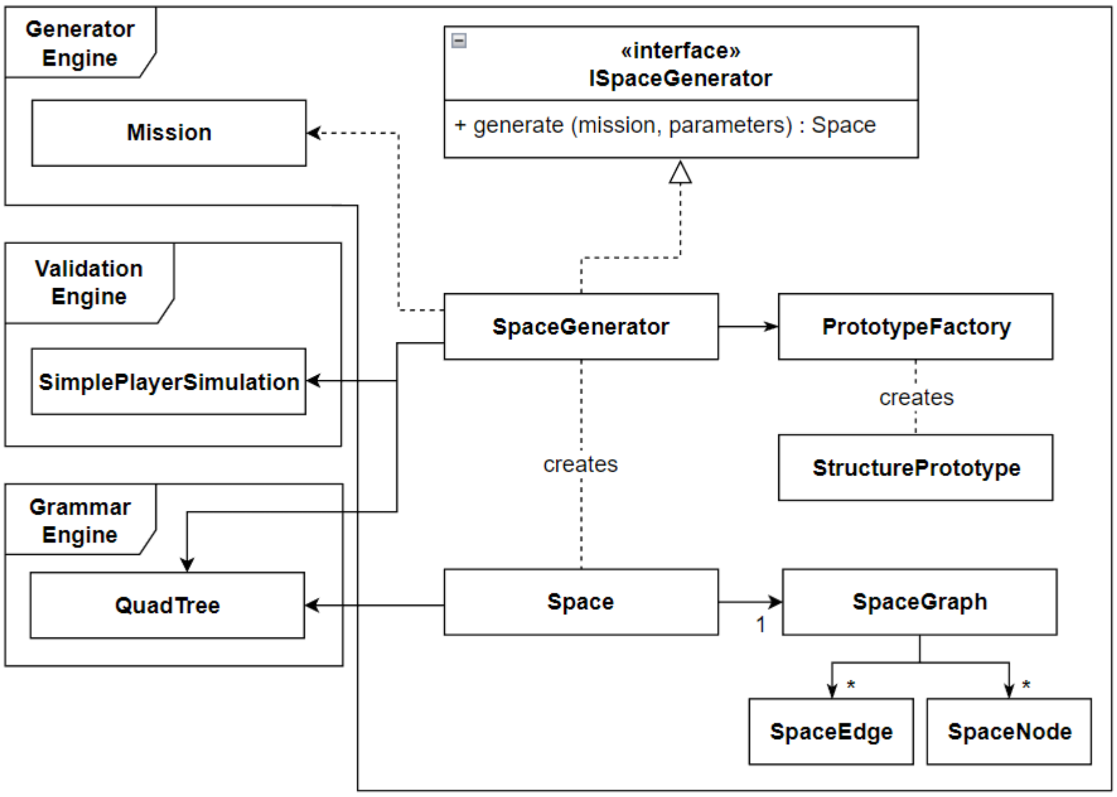


Figure 4.7: A class diagram of how the *ISpaceGenerator* is implemented

Space generator initialization and parameters

Before the generation process begins, there are certain dependencies that need to be set for the *SpaceMap-Generator* to work properly.

Firstly, a *SimplePlayerSimulation* object needs to be initialized, which is an object that represents the player in the generated space map. This player is used to determine the distance that prototype structures like platforms can be placed from each other horizontally or vertically without the need to add a connection structure between the platforms, the validation engine provides three predefined types of player simulation, *EasyPlayer*, *MediumPlayer* and *HardPlayer*, the difference between them being the distance the player simulation can jump. Increasing the jumping distance of the player simulation affects the maximum distance that a structure can be placed from the current one. So as the jumping distance increases, the player is required to make longer jumps to traverse the level, thereby increasing the level of difficulty.

Secondly, a *QuadTree* object needs to be initialized. A *QuadTree* is a data structure used to partition the space into smaller sub-spaces, which allows for efficient collision detection between prototype structures when trying to find a position to place them. In the prototype game created for this study, prototype structures can be placeable or non-placeable, if a structure is non-placeable then it should not overlap another non-placeable structure, the *QuadTree* helps detect these overlaps more efficiently when finding a location of a structure. To maintain a simpler level design, the prototype game was deliberately designed without any overlapping structures, such as platforms. However, in a more intricate game with multi-layered level layouts, checking for overlapping platforms may either not be desired or only done on a layer-by-layer basis.

Finally, the *SpaceMapGenerator generate()* method also takes in four generator parameters that influence the generation:

- ***maxDistanceBetweenPlatformsX*** and ***maxDistanceBetweenPlatformsY*** determine the maximum distance that can exist between two platforms in the horizontal and vertical directions respectively.

- ***minDistanceBetweenPlatformsX*** determines the minimum distance that can exist between two platforms in the horizontal direction, this property is useful to allow a gap to exist between platforms and allow the player to fall and access the platforms that exist below. In a more elaborate game with layered levels, this problem could be solved by creating platforms that allow the player to either fall through them or jump through them to reach the top.

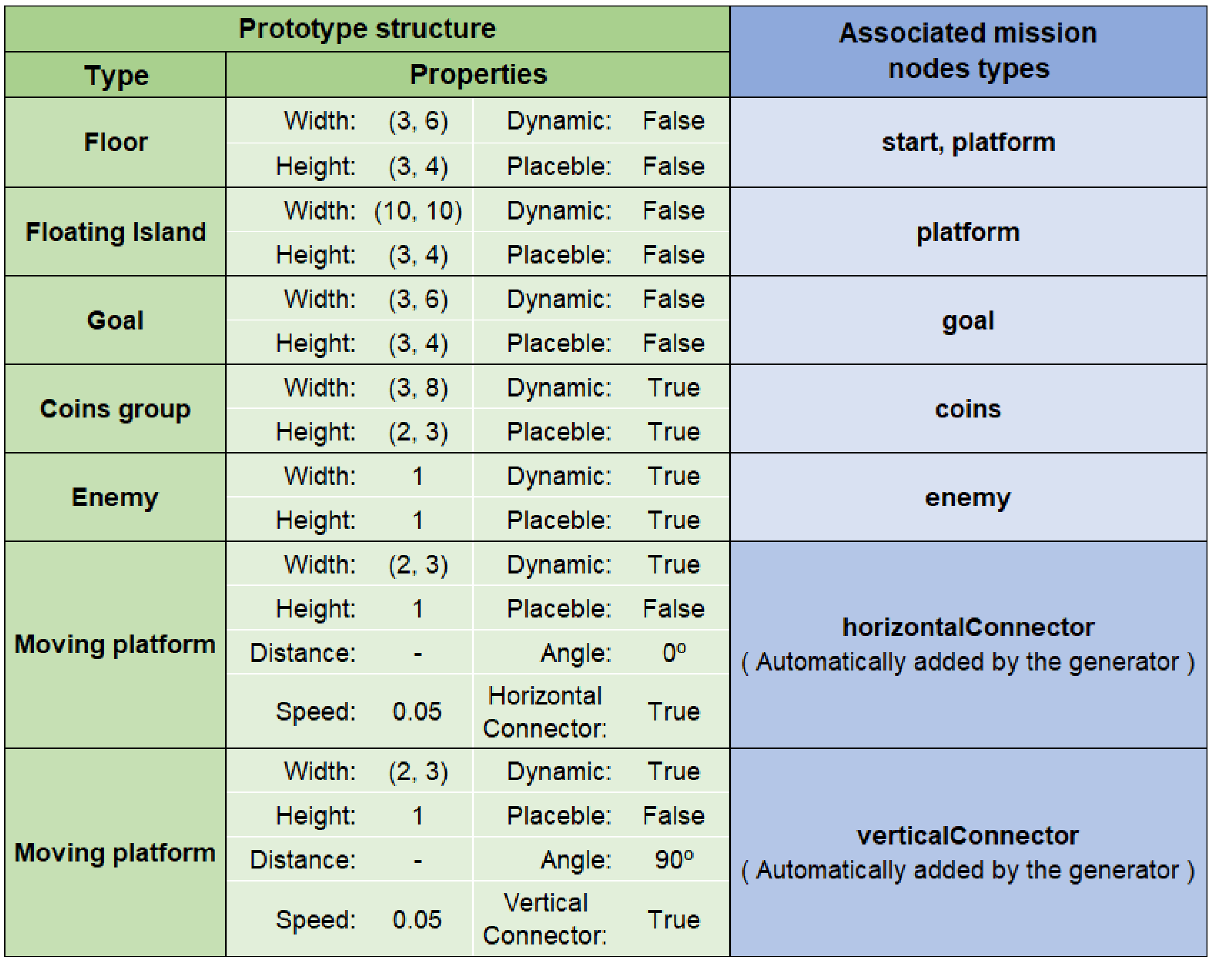
- ***addConnectionProbability*** parameter is used to determine the probability of placing a structure farther away from the player than the maximum distance he can jump. If the structure is placed outside the reach of the player, another prototype structure marked as a ”connector” is placed as well, to help the player reach the new structure. This adds a level of randomness to the generation process and can create more interesting maps.

Prototypes structures

As mentioned before prototype structures are templates used to hold desired properties and behaviours of actual game-level objects or pieces, they are an abstraction created so that the space generator does not need to know how the objects are implemented in the game engine. Each prototype structure is characterized by a specific set of primary properties. These include a type identifier, a list of key mission node types (i.e., mission grammar terminal symbols) that are associated with this type of structure, the dimensions the structure should have, whether or not this structure is dynamic, meaning they have one or more behaviour associated with it, as well as whether or not this structure is placeable, meaning that it can overlap other non-placeable structures. In addition to these primary properties, each individual structure prototype may possess its own unique set of specific properties. For example, a prototype for a moving platform may include an additional property to define the speed at which the platform moves.

To create and maintain a list of prototype structures, a prototype structure factory class called *PrototypeFactory* is utilized. Initially, this list of structures was populated by creating prototype structures through a *C#* constructor, but in the future, they could be imported using a JSON file or a scripting language. Incorporating a scripting language such as *Lua*[1] could not only facilitate loading parameter values for prototype structures but also allow for the integration of new behavioural logic for these structures without requiring the game to be recompiled. Fig. 4.8 shows a list of all the prototype structures used to create levels in the prototype game created for this project, on the left in green one can see separated by type the prototype structures and their properties, and on the right, in blue the mission nodes each structure is associated with.

[1]*Lua* is a lightweight, high-level scripting language that can be embedded into applications, and is used in the game development industry.



Prototype structure					Associated mission nodes types
Type	Properties				
Floor	Width:	(3, 6)	Dynamic:	False	start, platform
	Height:	(3, 4)	Placeble:	False	
Floating Island	Width:	(10, 10)	Dynamic:	False	platform
	Height:	(3, 4)	Placeble:	False	
Goal	Width:	(3, 6)	Dynamic:	False	goal
	Height:	(3, 4)	Placeble:	False	
Coins group	Width:	(3, 8)	Dynamic:	True	coins
	Height:	(2, 3)	Placeble:	True	
Enemy	Width:	1	Dynamic:	True	enemy
	Height:	1	Placeble:	True	
Moving platform	Width:	(2, 3)	Dynamic:	True	horizontalConnector (Automatically added by the generator)
	Height:	1	Placeble:	False	
	Distance:	-	Angle:	0^o	
	Speed:	0.05	Horizontal Connector:	True	
Moving platform	Width:	(2, 3)	Dynamic:	True	verticalConnector (Automatically added by the generator)
	Height:	1	Placeble:	False	
	Distance:	-	Angle:	90^o	
	Speed:	0.05	Vertical Connector:	True	

Figure 4.8: A list of all the prototype structures used in this project's prototype game

Generation process

In practice, the space map is generated by going through every node in the mission map graph in a breadth-first search manner, starting on the mission node marked as "*startNode*". On every step of the graph traversal, the current mission node, referred to as the parent, and its adjacent nodes connected by outgoing edges, known as children, are translated to prototype structures. This is done using a method from the *PrototypeFactoty* class, which takes in a mission node and picks at random a prototype structure from a list of structures associated with that mission node's type. For future studies, the choice of what structure to use should also depend on the neighbouring structures that are already placed on the level as well as any associated adjacent child mission node, this would allow a level designer to have more control over what structures should appear next to each other. To keep track of translated nodes and their associated prototype structures, they are stored in a dictionary data structure. This enables efficient access to the data and prevents the re-translation of nodes.

After doing the necessary translations, the parent structure is placed in its pre-calculated location. This calculation was done either in a previous step of the mission graph traversal or this is the first structure in the level, in which case, it is placed randomly at the start of the level. Next, the location coordinates for each adjacent child structure are determined, with the calculation being based on three factors. Firstly, it takes into account whether the structure is placeable or not. In the event that it is, a new non-placeable structure is generated to serve as its foundation and the dimensions of this new structure are used for the rest of the calculations. The specific type of non-placeable structure created is either selected from a list parameter of the placeable structure or by using a default structure if no such list exists. Secondly, the slope of the edge that links the child node to its parent node determines the height at which the structure should be placed in relation to that parent structure. If the slope is equal to 0 then it is placed in front of the parent, if it is equal to 1 it will be placed diagonally above the parent, and lastly, if it is equal to -1 the new structures will be placed diagonally below the parent. Thirdly, is the combination of the player AI jumping distance and the likelihood of adding a connector structure as determined by the value of the *addConnectionProbability* parameter, these will influence the vertical and horizontal distances the new structure is placed from the parent neighbour. To determine if a connection structure should be used, a random number between 0 and 1 is generated and if that number falls under the value set for the *addConnectionProbability* parameter then a connection structure is used.

With these factors in mind, the location-finding algorithm works as follows, once the dimensions of the prototype structure (or added foundation prototype structure) have been established, the algorithm proceeds to determine the height at which the structure should be placed in relation to its parent. Then depending on the height, different checks are made to calculate a location:

When placing a structure in front of its parent (see Fig. 4.9), the vertical position is set to be the same as its parent with a slight offset to ensure that the top of the child node aligns with the top of the parent node. The horizontal position is then determined by selecting a random value between the x coordinate of the top rightmost corner of the parent structure and the distance the simulated player can jump horizontally minus their width, or until the maximum horizontal distance if a connection structure is to be placed.

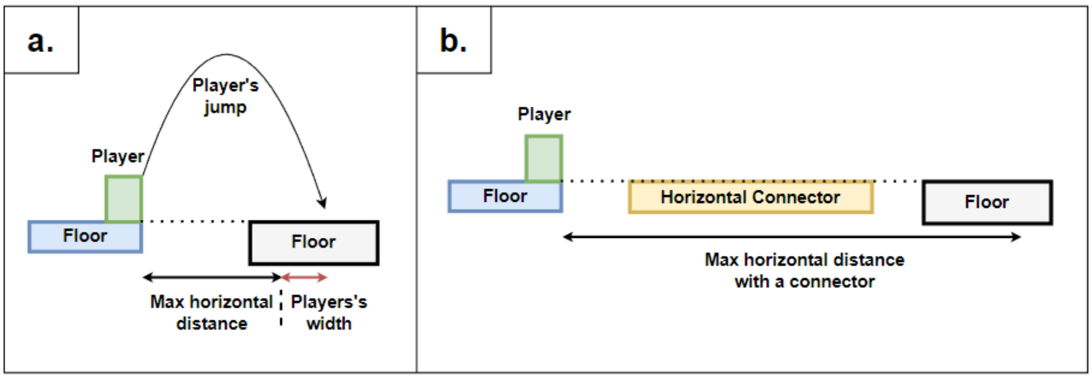


Figure 4.9: The max horizontal distance a child structure can be placed in front of its parent

If instead, the structure is to be placed diagonally above its parent (see Fig. 4.10), the vertical position is selected randomly from a range between the y coordinate of the top rightmost corner of the parent structure plus the simulated player's height and the value of that y coordinate plus the maximum distance the simulated player can jump vertically minus twice the player's height, or until the maximum vertical distance if a connection structure is to be placed. The horizontal position is then determined in the same way as in the previous case.

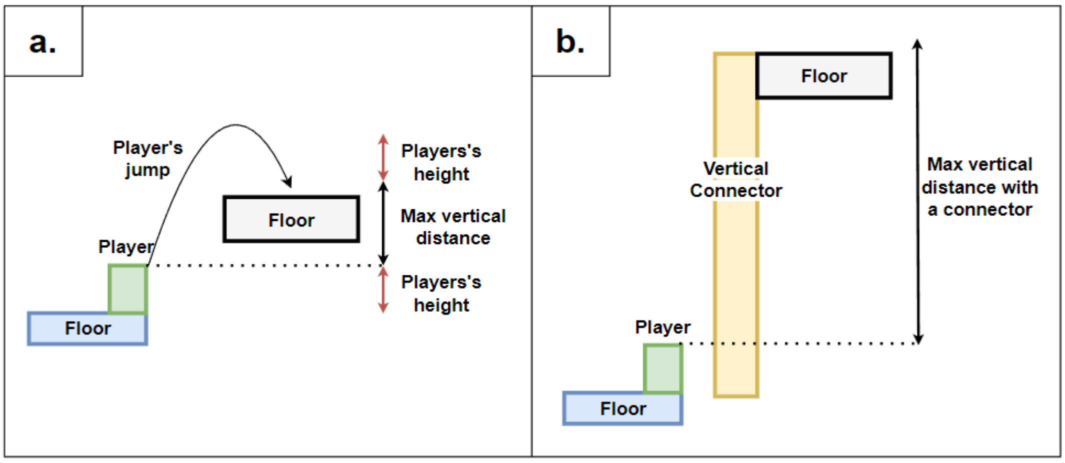


Figure 4.10: The max vertical distance a child structure can be placed above its parent

Finally, if the structure is to be placed diagonally below its parent (see Fig. 4.11), the structure's vertical position is chosen at random within a range that starts from the y coordinate of the bottom rightmost corner of the parent structure and ends at the value of that y coordinate minus the result of the sum of the maximum distance the simulated player can jump vertically and the height of the new structure being placed, and removing from that sum the player's height and the height of the parent structure. But, if a connection structure is to be placed, the range starts in the same y coordinate but ends at the value of that y coordinate minus the maximum vertical distance. The horizontal position is then determined in the same way as in the first case, where the structure was placed in front of the parent.

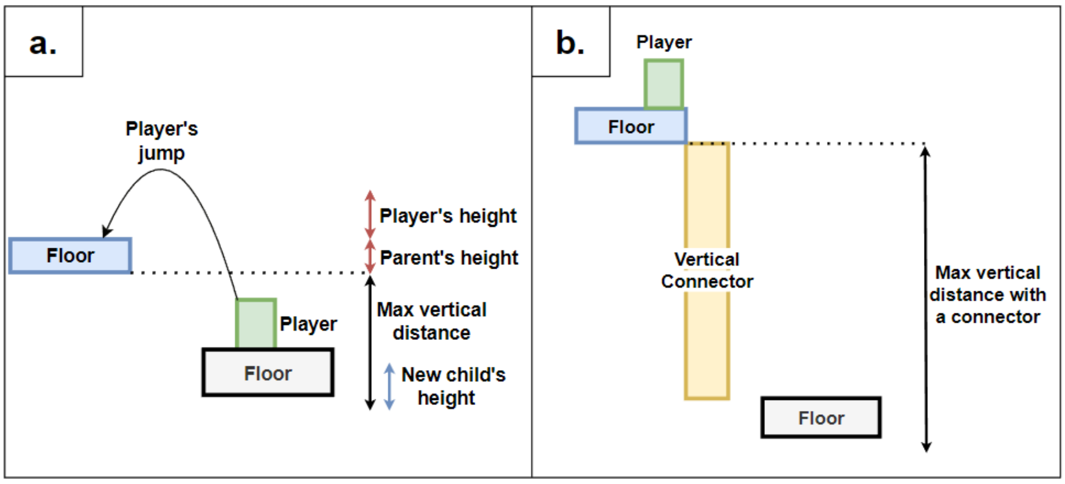


Figure 4.11: The max vertical distance a child structure can be placed below its parent

During this algorithm of positioning the children in relation to the parent node, if one of the children is going to be placed diagonally below and another in front of the parent, then when positioning the one in front of the parent a horizontal gap has to be left between the parent and the child structure being placed in front, to let the player reach the child structure below. This placement algorithm ensures that the vertical and horizontal jumping limits of the player are taken into account by subtracting the height and width of the player, respectively. This ensures that the player has enough clearance to make the jump safely. Additionally, the algorithm prevents any overlap between structures by checking the placement area of the child prototype structure using the *quadtree* to determine if any existing structures already occupy that space. If collisions are detected then the position of the prototype structure being placed is recalculated. Fig. 4.12 illustrates the two cases that derive from this recalculation. In the normal push-forward case (4.12a.), if there is not enough space to fit the structure A in front of its original parent structure, the algorithm selects the rightmost structure that occupies the area where A was intended to be placed as the new parent. The reason for selecting the rightmost structure in the collision area is that, in the game prototype developed, levels are constructed from left to right. Therefore, this particular structure has a higher probability of not having other structures in front of it at this point of the level creation. Structures in the original placement area are considered first as parents, in order to try to respect the mission map layout. A check is then made to see if A fits in front of this new parent structure. If another collision occurs, this process is repeated by selecting another new parent structure until a suitable location is found that can fit A without causing any collisions or until a limit of tries is reached. If this limit is reached then the second case (4.12b.) is applied. In this case, A is discarded and its child nodes are instead added as child nodes of the mission node that is associated with the rightmost structure that occupies the area where A was intended to be placed originally. This iterative process ensures that all structures are placed in a way that avoids overlap with other structures.

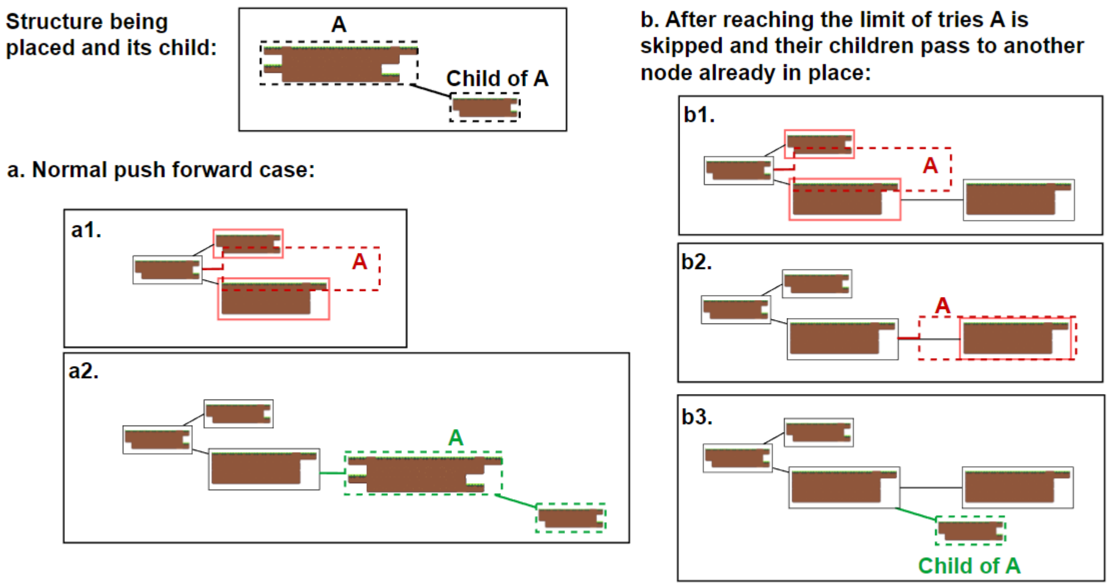

Structure being placed and its child:
A
Child of A
a. Normal push forward case:
a1.
A
a2.
A
b. After reaching the limit of tries A is skipped and their children pass to another node already in place:
b1.
A
b2.
A
b3.
Child of A

Figure 4.12: Both cases of structure placement collision and how they are handled

After all nodes of the mission map are visited once, and all possible structures have been placed, then the space map is complete and is returned as *Space* object. A space map example generated from the mission map shown in Fig. 4.6 can be observed in Fig. 4.13.

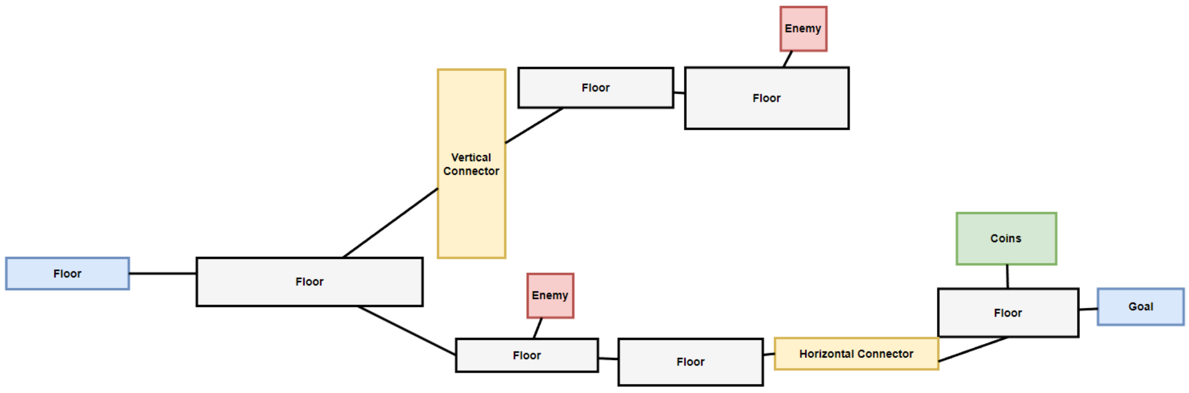


Figure 4.13: A partial space map

In the illustration above, the start and goal space nodes are painted in blue, the connectors are painted in yellow, the red and green ones are placeable structures, and the rest are non-placeable structures. The lines between the structures are the edges between the space nodes, in this example, all structures are separated by an empty space so that the edges between nodes are more easily seen, in a real space map generated, structures can be positioned right next to each other. The validation engine uses these edges to find a path from start to end that the player simulation can take to test if a level generated with this space map is possible to be completed. Using this space map as a blueprint the game engine can now replace each prototype structure with an actual game object and create the level. An extra example of a more complete space map, created by transforming the mission map in appendix A.5, can be seen in appendix A.6.

4.4 Validation engine

The validation engine, in this project, functions as a critical component of the level-generation process. Its primary purpose is to ensure that the generated levels are playable and consistent with the level design goals and constraints. The validation process involves defining project-specified parameters that are used to score and rank the generated mission and space maps. To accomplish this task, the engine contains two validation classes, namely a *MissionValidator* and a *SpaceValidator*, both of which receive a set of predefined parameters that are utilized to evaluate and score their respective maps. Since each game has its unique requirements and validation criteria, the engine allows developers to create new parameters. To achieve this, a developer needs to extend the *Parameter* class and provide an implementation for the *score()* function, which returns an object of type *Score*. This engine also provides a player AI simulator that allows for the validation of space maps. The simulator is responsible for simulating the behaviour of the player as they traverse through the space map.

4.4.1 Mission validation

The *MissionValidator* is responsible for validating mission maps, and for this particular prototype game, its main focus is to ensure that the generated map is of the appropriate size, has the correct structure, and provides a suitable challenge for the player. With this in mind, four parameters were defined, the level size, graph linearity, number of enemies, and number of coins. The level size parameter ensures that the generated map is of the appropriate size, while the graph linearity parameter validates the connectivity and structure of the map. The number of enemies and coins parameters can be used to ensure that the level has a reasonable difficulty level and rewards for the player. These parameters are described in more detail in section 3.2.2 (Validating mission maps).

For each of these four parameters, a scoring function was created. The level size scoring function checks the number of nodes the mission graph has in total, the graph linearity scoring function counts the number of edges between nodes that have a slope different than 0, meaning that they represent a change in elevation in the level, and the number of enemies and coins parameters scoring functions count the number of enemy or coin nodes in the graph, respectively. The score of each of these parameters is then summed together and that gives the final score of the mission map. Missions with bigger scores rank higher.

4.4.2 Space validation

The validation of space maps is done by the *SpaceValidator* class. For this specific prototype game, its primary purpose is to verify that the generated map is both playable and beatable, while also ensuring that it is not too short. To accomplish this task, the validator contains a player AI simulator in the form of a class called *SimplePlayerSimulation* that determines the feasibility of the map. Three other parameters were also defined for the space validator, namely ”distance to goal”, ”number of collisions”, and the ”length of the player path”. These parameters are described in more detail in section 3.2.3 (Validating space maps). The scoring of each of these parameters is done using their respective scoring functions. The ”distance to goal” scoring function is simply the Euclidean distance between the start node position and the end node position, the ”number of collisions” scoring function relies on the fact that went the space is being created and the structures are being placed the number of collisions that happen is recorder in a variable on the space map object, and finally, ”length of the player path” is the length of the shortest path from the start node to end node determined using the $A*$ algorithm. Each of these parameters has a weight associated with them that influences how they affect the final score.

The scoring algorithm begins by calculating each of the three previous parameters and multiplying them by their respective weights. Afterwards, they are all added together to create the final score of the space map. Next, the player simulation is used to traverse the path identified. This traversal involves testing two scenarios: first, whether the player can navigate the structure associated with a node from one side to another without colliding with any placeable structures, and second, whether each consecutive node in the path is positioned at a jumpable distance from the player. It is important to note that, in this space graph, connection structures have their own space node. If the player simulation collides with any placeable structures or is unable to jump to a structure from another, the space map is considered not feasible and is discarded, ignoring the previously calculated score. If the simulation passes both conditions, then the space map is given the previously calculated score. The ranking of the scores follows the same logic used in mission validation, with higher scores being ranked above lower scores.

A limitation of the validation approach used to assess the feasibility of a space map is that it is overly simplistic and may result in discarding levels that can actually be beaten. This is because the test only considers the shortest path in the space map’s graph, which does not account for all the possible paths a player can take. Players can freely jump and fall from one structure to another, even if they are not connected in the graph. Additionally, the shortest path may be blocked by structures placed too closely together, but a player can still take another path not represented on the graph to reach the goal. However, in the prototype game developed for this study, generating a level involves creating several space maps. Consequently, a swift and straightforward validation approach like the one used in this study is capable of quickly testing each of the generated spaces, identifying only those that are feasible. Making this validation process adequate enough for selecting workable maps that players can enjoy.

In future iterations of this tool, it would be interesting to develop a more advanced simulation and validation process that considers these limitations, and tests for a wider range of possible paths, while creating graphs that more accurately represent the layout and possible movements of the player in the level. Perhaps instead of finding the paths on the space map graph, a better approach would be to represent the space map as an intermediate tilemap that reflects the level layout, with each tile representing a specific width and height area. This tilemap would be a matrix where the structures would be laid out in their corresponding x and y coordinates converted to a position on the matrix. In this matrix, it is only possible to change cells with a value of 0 during the placement of structures. Any cell containing a non-placeable structure is designated as -1, while walkable areas, such as cells above non-placeable structures and connector structures, are marked as 1. The remaining structures and empty spaces would be left as a 0. Next, the $A*$ algorithm could be used on that matrix to find the shortest path from the start position to the goal position, by stepping only on cells that are marked with a 1 or cells that are marked with 0's that are in close proximity to a 1, the proximity being determined by the player's jumping distance. While finding the shortest path, the player's dimensions would be taken into account as well. This means that the algorithm would take into account the neighbouring area with the player's dimensions while selecting the next step. This proposed alternative approach, although more complex than the one used, is still relatively straightforward and does not require excessive computational power to execute. Moreover, the algorithm would consider the actual level layout in its entirety, and generate a path that closely resembles a route a real player would take. This path could then be tested using a player simulation to verify its feasibility.

4.5 Game engine

In this section, we will discuss the game portion of the prototype developed for this study. This simple game was created to help examine the performance and expressive capabilities of the generator framework constructed in the previous two sections. Dubbed the game engine, this layer contains all the game logic and acts as a stand-in for the actual logic of a game where a developer would want to use this generating framework to generate new levels. The stand-in game created consists of a simple platformer game with minimal assets. The game contains two distinct modes: a level selection mode and a sandbox mode. Within the sandbox mode, players are able to adjust various properties before generating a level, and can freely generate or refresh levels as desired. On the other hand, the level selection mode generates ten levels that can be played in sequence. As each level is successfully completed, the subsequent one starts. Progress is automatically saved, enabling players to return and continue playing at a later time. Each level is generated and consists of a player moving from left to right while jumping from platform to platform, gathering coins, and avoiding or defeating enemies to reach an end goal marked by a red flag.

Inside this engine, a *LevelController* object is used to interact with the Unity engine and set up level-building dependencies, in order to prepare the level to be played. It calls upon a *LevelGenerator* object to generate the level, which produces a *Level* class object. The *LevelGenerator* contains an implementation of *IMissionGenerator* and the *ISpaceGenerator* provided by the generator engine, it also contains a *MissionValidator* and *SpaceValidator* provided by the validation engine to aid in generating valid levels. The class diagram in Fig. 4.14 shows how this system is connected.

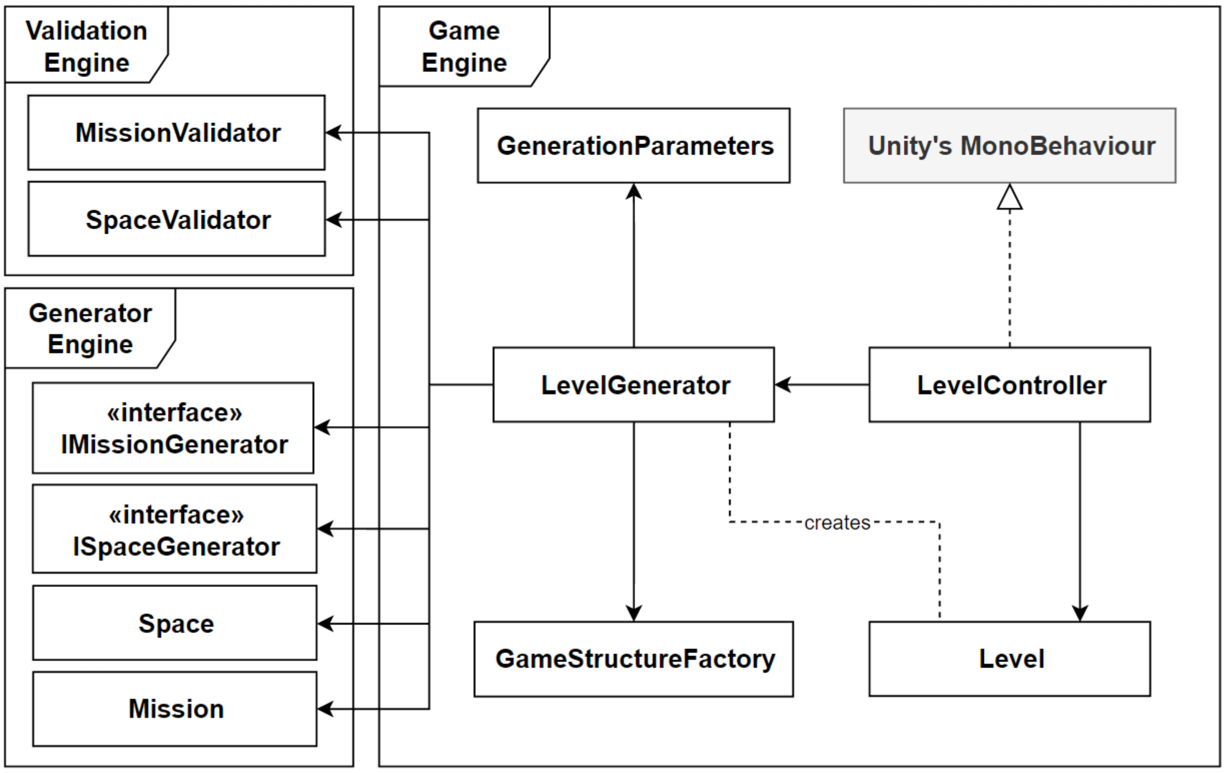


Figure 4.14: A class diagram of how the level generation is implemented

The *LevelGenerator*'s *generate()* method employs these dependencies to generate the level. It begins by generating multiple mission maps with the mission generator, and then these missions are scored and ranked using the *MissionValidator*. When the scoring is done, only the top-rated missions are retained. Subsequently, the list of top missions is traversed, and a space map is generated from each mission map using the space generator. Afterwards, the *SpaceValidator* also scores and ranks the resulting space maps, but this time only the best one is chosen to be converted into a level.

Converting a space map into a level is done by translating each prototype structure into the actual game objects they represent, this is done using the *GameStructureFactory* which converts prototype structures into the respective game objects they represent. During this transformation process, all of the necessary parameters and connections of the game object, are set according to the specifications defined in the prototype structure, including logic parameters and the object's placement within the level. Each game object is visually represented on the level by one or more image sprites. In the end, the *LevelControl* obtains a fully constructed level, containing a list of game objects, and all the necessary parameters and logic to render the level on the screen. Below, Fig. 4.15 shows a level created from the space map in Fig. 4.13.

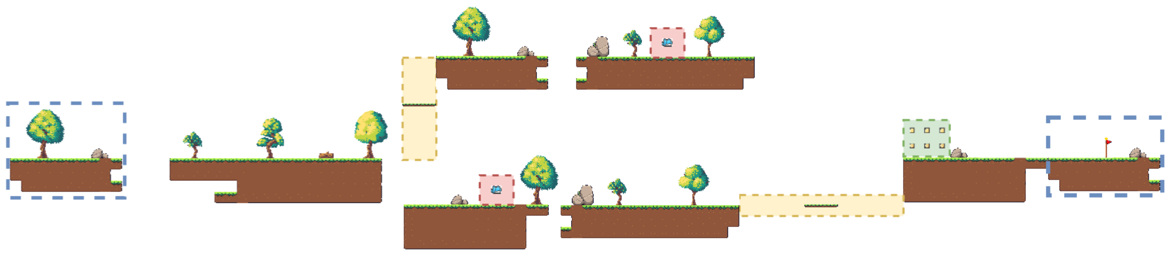


Figure 4.15: An example of a finished level

The terrain depicted above represents a finished generated level, where the prototype structures of floor type were transformed accordingly to create the terrain illustrated. Similar to the previous space map example, the colours in the illustration indicate the various types of structures present. The blue-lined terrains mark the starting and ending points of the level. Notably, the vertical and horizontal connectors were transformed into dynamic, moving platforms that traverse within the yellow area, represented in the illustration. Furthermore, the enemies selected for this level are highlighted in red, while the coins are marked in green. Additionally, the trees and decorative objects in the background are generated in conjunction with the terrain structures and selected randomly from a predefined list of decorations. Annexed at the end of this document in appendix A.7 is a more complete and longer level example.

4.6 Summary

In summary, the implementation chapter has presented a detailed account of the technical aspects involved in building the generator framework as well as the game prototype used to test the framework. An outline was made of the various tools and methods used to create the project. This chapter also described the project's architecture and how it is designed to generate and validate levels using various dependencies, which are divided into layers. At the bottom layer the grammar engine function as a custom-made graph rewriting system that can load and apply grammar rules. Then the generation engine uses the functionality provided by the grammar engine to generate mission maps and space maps. And finally, the game engine transforms the space maps into proper levels that can be played. Alongside these layers, the validation engine offers methods to validate each step of the generation process, particularly the created mission and space maps. The next chapter will explain and show the results of how this implementation was tested and analysed.

Chapter 5

Analyze and Results

To explore the design and expressivity of the generator, several data points were taken and analyzed from levels generated by the ”*2D platformer generator*” prototype game, using three different mission grammars. Additionally, the prototype game was made publicly available on the website *itch.io* [26] along with a small data-collecting survey [27] for people to play-test the prototype and then answer the survey. This chapter will present how the data was collected and the subsequent analyses made with that data.

5.1 Generator expressivity evaluation

The expressivity of the created generator is directly linked to the design of the mission grammar, the type of prototype structures the game needs and the parametrization in the generation. Since the main goal of this project was to investigate the generation of levels using rules, an evaluation test was conducted to analyse how the generator would perform with different grammars. As a result, three different mission grammars were used to generate levels with the same generation parameters and the same prototype structures. For each of the three grammars, 100 levels were generated and evaluated using four data points ”linearity”, ”leniency/difficulty”, ”density”, and ”candidate feasibility”. These data points were selected because they represent key aspects of level design that can affect the overall player experience. As a matter of fact, the data points of linearity, leniency, and density were used by Britton Horn et al. in their study on comparing different Super Mario Bros. (SMB) level generators [28]. They argued that by analyzing these three characteristics, they could gain insight into the effectiveness of each generator at producing levels that are similar to human-designed levels in terms of their structure, difficulty, and pacing. The last data point, ”candidate feasibility”, was also analyzed since it is closely linked to the performance of the generator. All data values collected were normalized to the range [0,1] using a min-max normalization.

5.1.1 Evaluation mission grammars

As stated, in order to better analyse the generator, three mission grammars were used. Grammar A (see appendix A.1) is the main grammar used by the prototype game developed for this study, it is a simple grammar, designed to create linear platform segments with the occasional choice to go up or down, but both paths eventually connect to the main path again. Grammar B (see appendix A.2) exhibits a subtle variation from the previous one. It is also primarily tailored to generate linear platform segments while introducing a greater number of side paths that do not intersect with the main path. Furthermore, it incorporates a customized end-cell pattern featuring two potential goals. Grammar C (see appendix A.3) is an example of a more complex rule system, designed to create levels that have sections with multiple paths. This last grammar serves as a stress test to see how the generator handles a grammar that creates mission maps with multiple crossing edges.

5.1.2 Evaluation parameters

Linearity is influenced by the presence of various altitude differences along the level created by the variances in platform height. A very nonlinear level has frequent variations in platform height. A level with such features forces the player to execute more jumps, allowing him to reach higher areas, it can also give the player more than one alternative path to the level's goal. This parameter was measured by iterating through the elements of the level, computing the absolute values of the differences between the current element height and the highest element in the level, and adding them together. After calculating the sum, the result is negated, such that, the values of nonlinear levels are closer to zero after normalization.

Leniency or **difficulty** measures the level's tolerance in terms of how easy it is for the player to complete it. A very lenient level can make the player feel not challenged enough and bored, in contrast, a very non-lenient level can make the player frustrated to play. The difficulty of a level is very subjective to the style and audience of a game, a casual player might find a level very difficult, but a *Kaizo Mario World*[1] veteran player would find the same level very easy. However, attempting to measure it can provide additional insight into what kinds of levels the generator is capable of generating. The same measurement technique employed by Britton Horn et al., was applied here by iterating through the level's elements and assigning different lenience values to certain level elements:

- Gaps: -0.5;
- Average gap length: -0.5;
- Structure connectors: -0.5;
- Enemies: -1;

Density, in the context of this study, refers to the frequency with which a structure's position had to be altered because it would have otherwise intersected with another structure already placed on the level. The generator aims to avoid such collisions in its placement of structures.

Candidate feasibility represents the number of space map candidates that during the level generation were marked as feasible. Feasibility is the measure of whether the level can be completed by a human player, it is determined by going through the space map using a player simulation and checking if the simulation can reach the goal. In this respect, for each space map candidate created during the level generation, the value of feasibility is either 0 (not feasible) or 1 (feasible). Measuring the number of feasible candidates produced during the level-generation process is a valuable way to evaluate the performance of the generating algorithm. This is because the generator employs a space validator to reject any space maps that are not feasible when selecting potential candidates. Therefore, a low candidate feasibility rating indicates that the generator is wasting resources while creating invalid options.

[1]*Kaizo Mario World* is a series of *Super Mario World* ROM hacks that features extremely difficult level designs

5.1.3 Test generator parameters

As explained, the test involved generating 100 levels using the ”*2D platformer generator*” prototype with three different mission grammars, but keeping the generator parameters and prototype structures constant. The prototype structures used for this test were the same ones created for the ”*2D platformer generator*” prototype game and the generation parameters were set to their default values:

- level size: 1 (preference for larger levels);
- enemies: 0.8 (preference for more enemies);
- coins: 0.8 (preference for more coins);
- mission graph linearity: 0.5(preferring a similar amount of mission maps with varying edge slopes).

The generator also contains two important generation parameters, which are ”number of iterations” and ”number of candidates”. The ”number of iterations” represents the maximum number of times a grammar can be used to alter a graph in the mission map creation process and it is also a safety precaution that prevents infinite rule recursion from happening. Setting this property too low means that not many grammar production rules are applied, therefore the grammar's syntax cannot be effectively tested, setting it too high makes the number of structures and length of the level increase and it might have an impact on the feasibility of the levels generated. As for the ”number of candidates” parameter, this represents the number of mission and space map candidates created and evaluated during the generation of a level. Setting this too low means that there are fewer options to choose from when scoring and picking candidates, and setting it too high means that the generator has to spend more resources generating and evaluating more candidates. To analyse the impact that changing these two parameters has on the candidate feasibility of the generator, two tests were conducted.

On the first test, mission grammar A (see appendix A.1) was used to create levels. During this test, the ”number of candidates” parameter was kept constant at 70. While the ”number of iterations” parameter varied, and for each value of this parameter 100 levels were created. To determine the percentage of feasible levels for each number of iterations, an average of all feasible candidates was calculated from the total of candidates created during the generation of the 100 levels. This calculation was made for each of the values of the ”number of iterations” parameter. Fig. 5.1 illustrates how the percentage of feasible levels changes with different values of the ”number of iterations” parameter. We can see that the percentage of feasible levels tends to decrease as the number of iterations increases, with the tendency to stabilize at around 10% feasible levels. This means that the generator works best when a small number of rules are applied, so this property was set to 8 in this evaluation test.

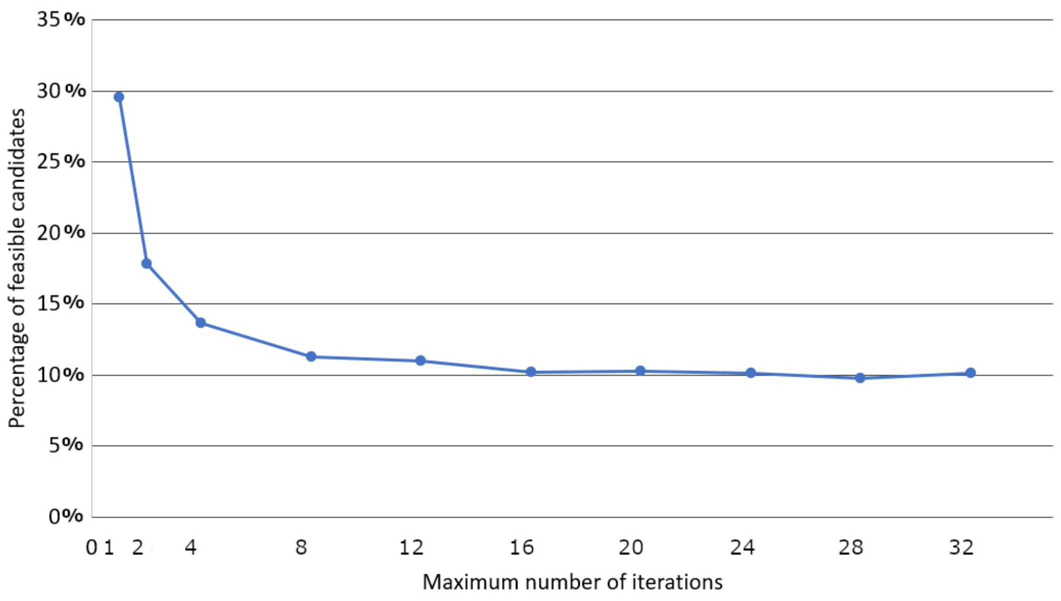


Figure 5.1: Average percentage of feasible candidates created during the generation of 100 levels with different maximum numbers of iterations to apply rules

The graph in Fig. 5.1, also shows that even at just one iteration the percentage of feasible candidates is just 30%, suggesting that either the generator is not effective at producing feasible levels or there may be issues with the feasibility validation process. Furthermore, it should be emphasized that the ”number of iterations” attribute exclusively impacts the number of times rules from the regular grammar are applied. Once the specified number of iterations is reached and the resulting graph still contains non-terminal symbols, the closing grammar rules are applied until all the nodes in the graph represent terminal symbols of the mission grammar utilized.

The second test followed a similar format as the first one, but with a variation in the ”number of candidates” parameter while keeping the ”number of iterations” parameter constant at 8. Fig. 5.2 illustrates how the percentage of feasible levels changes with different values of the ”number of candidates” parameter. We can see that increasing the number of candidates only decreases the percentage of feasible candidates created very slightly. This decrease is not very significant which indicates that the number of candidates has very little or no effect on the number of feasible candidates generated, even though intuitively more tries means more chances of generating feasible candidates. It also reveals that when using the grammar A to generate a level with the ”number of iterations” parameter set to 8, the average percentage of candidate feasibility in the generator is 14%. This implies that during the generation of a level, the generator is wasting an average of 86% of its time creating invalid level candidates, once again suggesting that either the generator is not effective at producing feasible levels or there may be issues with the feasibility validation process. Given the generator’s low success rate, it is essential to have a sufficiently large number of candidates to ensure that at least one of them is feasible. However, it is equally important to avoid setting the number of candidates too high, as this could cause the generator to waste time creating invalid levels. Therefore, the number of candidates was kept at 70.

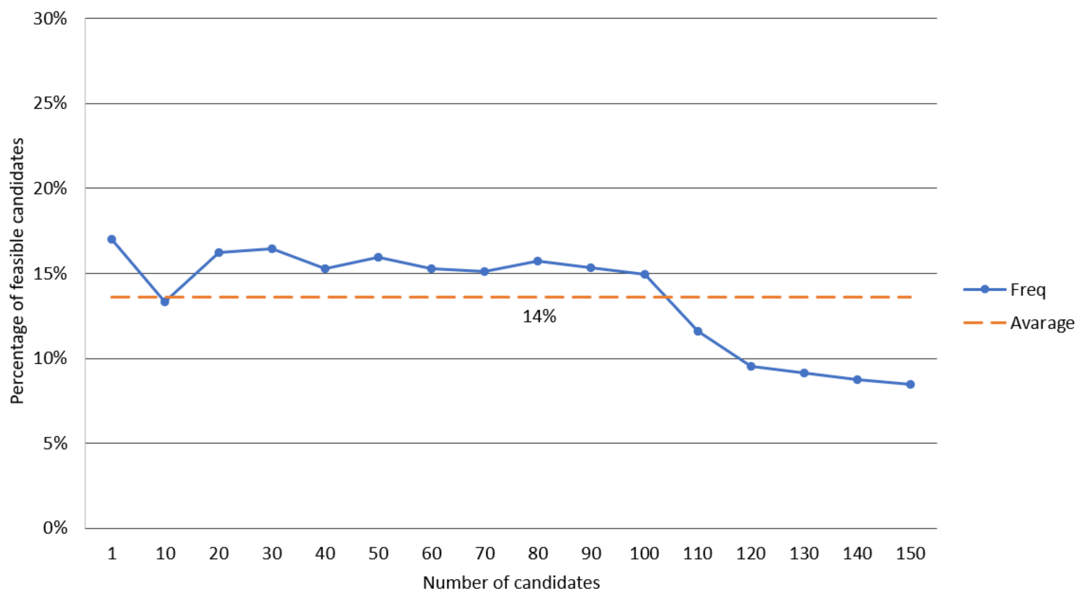


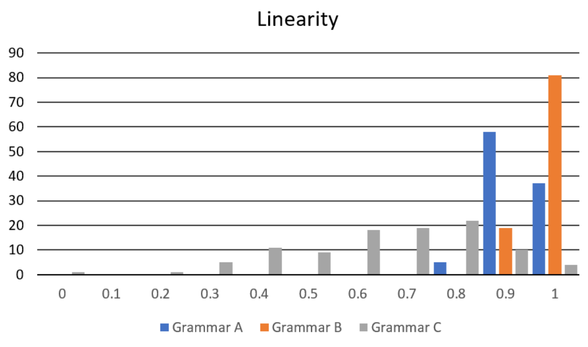
Figure 5.2: Average percentage of feasible candidates created during the generation of 100 levels with different amounts of candidates

5.1.4 Results

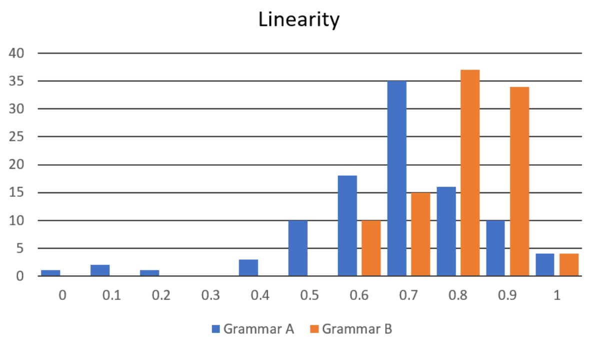
In the following section, two histogram charts will be presented side by side for each of the four data parameters to compare the frequency of the values between the grammars. The charts on the left depict values normalized between all the data collected with all three grammars, but because grammar C contains more extreme data values, the data from grammar A and B is squished together along the x-axis making it harder to see the difference between the two. So for each data parameter, a second histogram was constructed depicted on the right side, taking only the normalized data between grammar A and B into account, making the difference between the more explicit.

Linearity

From the linearity left chart 5.3a, it is possible to observe that grammars A and B generate mostly linear levels, as expected, and that the more complex grammar C generates levels with a wider range of linearity values. On the right chart 5.3b we can see that although both grammars generate linear levels, grammar B seems to be more linear even though it was designed to have more side paths. But when comparing these results with the mission maps created with both grammars 5.4 it becomes evident that even though grammar A does not generate graphs with side paths, the height of the main path can vary substantially, while in the graphs created by grammar B there is a lot of branching but the branches don't deviate much in height compared to the main path.



(a) Comparing the three grammars



(b) Comparing grammar A and grammar B

Figure 5.3: Histogram comparing the linearity of 100 levels created by different grammars

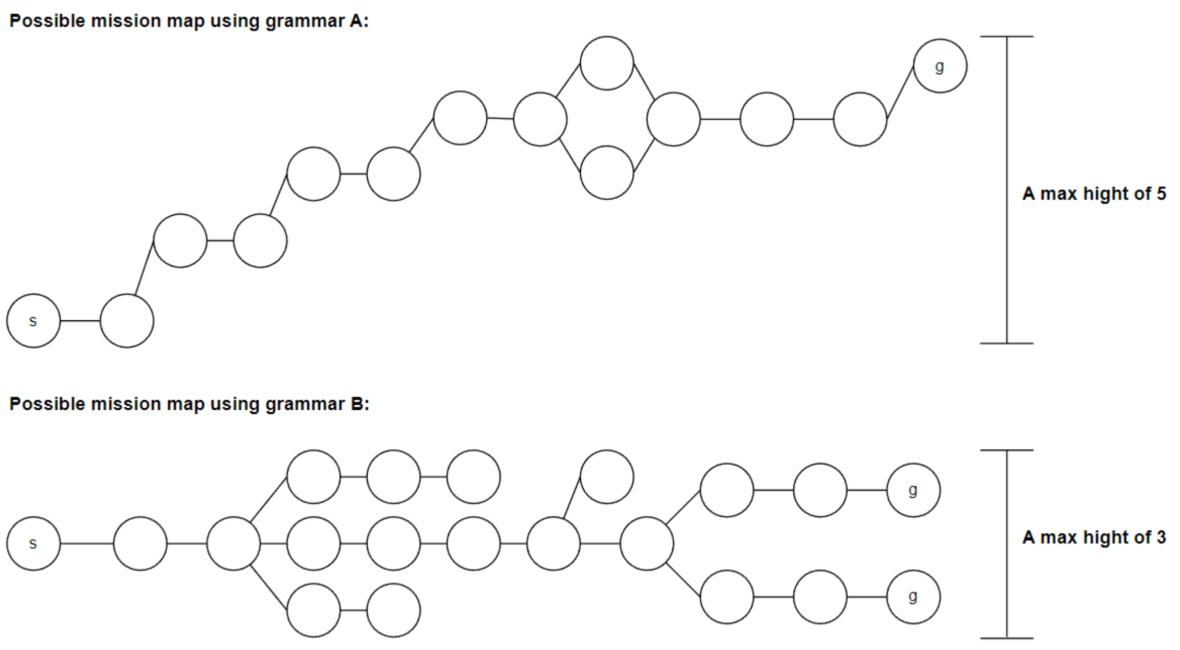
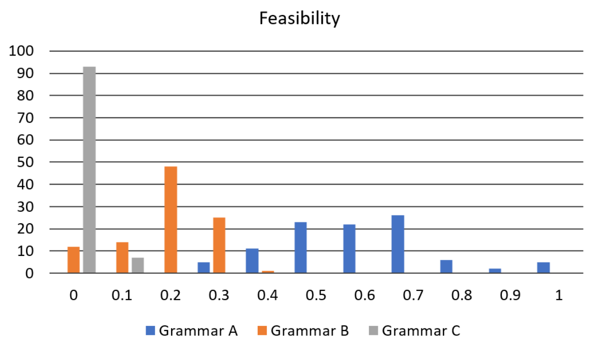

Possible mission map using grammar A:
s
g
A max hight of 5
Possible mission map using grammar B:
s
g
g
A max hight of 3

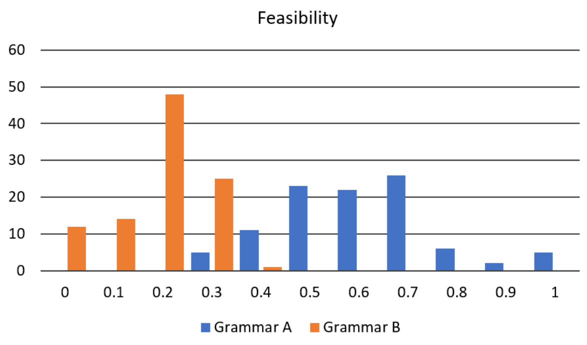
Figure 5.4: The shape and height of a mission map created with grammar A and another created with grammar B

Feasibility

The left feasibility chart 5.5a, shows that the generator has problems validating levels when grammar C is used, in 90% of the levels created with this grammar, none of the 70 candidates created could be traversed by the player simulation, meaning that the generator failed to produce a valid level. This is due to the way the grammar is designed, many production rules place platforms at different heights, meaning that nodes and edges in the mission grammar can overlap with each other. During the creation of a space map the slopes of the mission map edges are used to guide the placement of structures, this has the advantage of creating maps that have more or less the same shape as the mission map graph. But, if the mission graph is too tangled, many space structures will collide in the placement process, causing the algorithm to send them to the end of the map to try and place them in a new location, or in cases where many structures have already been placed in close proximity, it can make the algorithm skip placing the structure entirely because it can not find a place to put it. Essentially, when the algorithm is forced to relocate a structure it invalidates not only the current structure position calculation but, also the placement of all the structures linked to it that have already been processed and placed. This effect is less noticeable in grammars A and B because their production rules were designed to grow the graph in a more directed way, instead of letting the graph branch out in every direction. However, because grammar B still crates some breaching paths and tends to group structures closer together, in the feasibility chart on the right 5.5b we can see that it creates less feasible maps than grammar A . In fact, if we set the generator to ignore all non-feasible maps and analyse the output of each grammar (see chart 5.6), we see that grammar A can generate all 100 levels without ever failing to generate a valid level from the number of candidates used, while grammar B and especially C failed sometimes.



(a) Comparing the three grammars



(b) Comparing grammar A and grammar B

Figure 5.5: Histogram comparing the feasibility of 100 levels created by different grammars

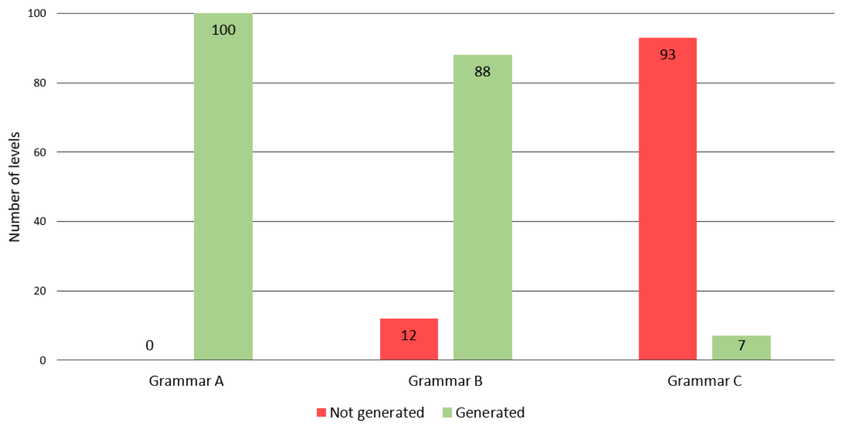
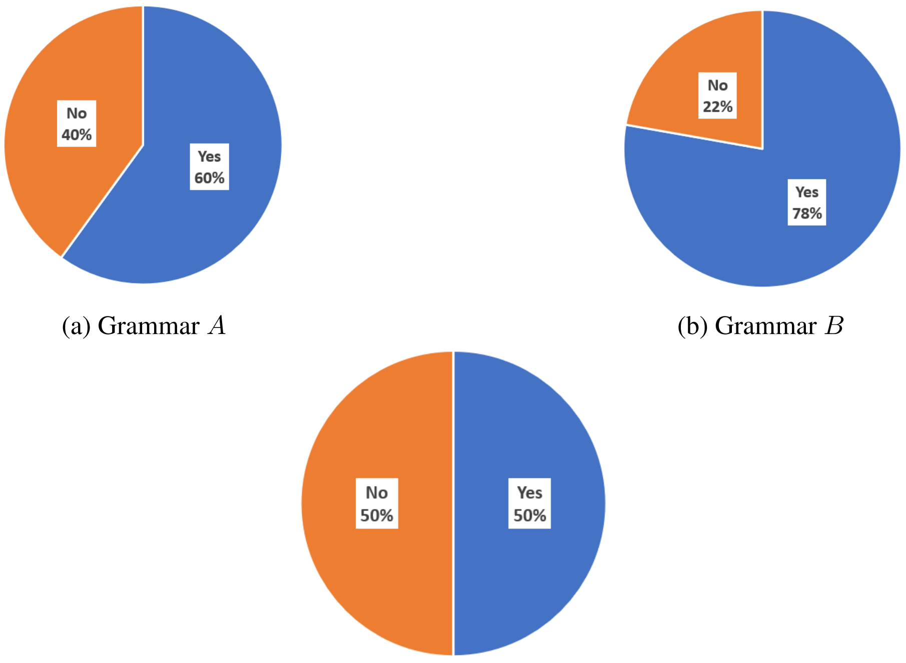


Figure 5.6: The number of levels each grammar could actually generate out of 100

This data, also indicates that the validation system, to determine if a level is feasible, might be too restrictive since the grammar A has an average of 86% non-feasible candidates and both grammar B and C show an even bigger number of non-feasible candidates. As such, another test was made to analyse how many of these non-feasible levels were actually impossible to complete by a human player and determine the error associated with the player simulation test used in this feasibility validation of a candidate. The test was done by setting the generator to output the best-fitting non-feasible candidate as the generated level. Ten levels were created using each of the grammars and then they were play-tested by a human. Fig. 5.7 shows three pie charts depicting the percentage of levels that could be beaten by a human in blue versus those that could not in orange. We can see that for all three grammar, the percentage of levels that could be completed by a human is bigger or equal to those that could not, which confirms the suspicion that the feasibility validation is too restrictive or has problems. This is most likely caused by the fact that the player AI only checks whether it can take the shortest path from the start to the end in the space map graph, so if any node in the shortest path ends up having its structure blocked by some other node's structure by being too near it, the player AI will not be able to cross that node structure and be blocked from reaching the end making the level as non-feasible, but a human player can just find another path.



(a) Grammar A

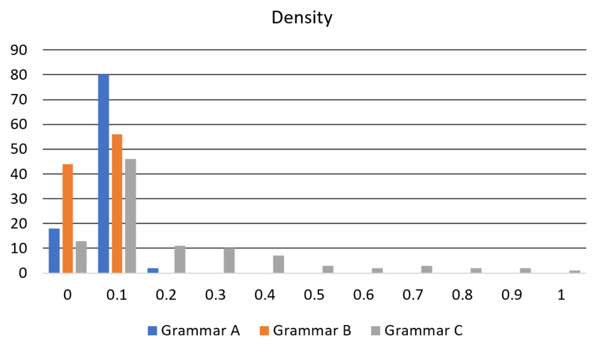
(b) Grammar B

(c) Grammar C

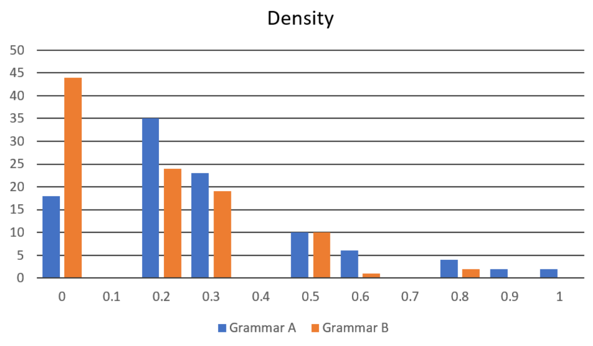
Figure 5.7: Shows for each grammar the percentage of levels marked as non-feasible that could be completed by a human (Yes) versus those that could not (No)

Density

Looking at the density chart comparing all the grammar 5.8a, it becomes even more clear why grammar C has such a poor performance in the creation of levels. Although the density values are scattered across the graph, grammar C has the biggest number of collisions overall which means that a lot of corrections were made by the generator while trying to place all the structures, this has two consequences: First, it can create levels without platforms or connecting structures that make the level impossible to complete. Second, it can create a space graph with misplaced edges connecting the structures, this cuts one section of the graph from another section making it impossible to find a path from the start node to the goal node for the simulation player to test. Both of these consequences make the level be classified as not feasible. When comparing the design of grammar A with B, it is surprising to see that B creates levels with fewer collisions in general, but still, its performance is worse than grammar A. This may be explained by the fact that grammar B generates levels with very clustered pathways, making structure repositioning more difficult, and forcing the algorithm to hit the limit of replacement tries and skip the structure which might make the level not feasible.



(a) Comparing the three grammars

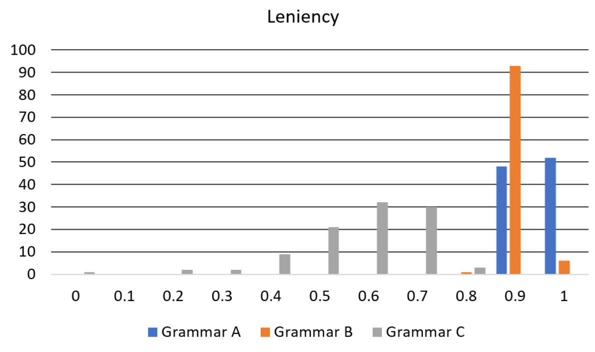


(b) Comparing grammar A and grammar B

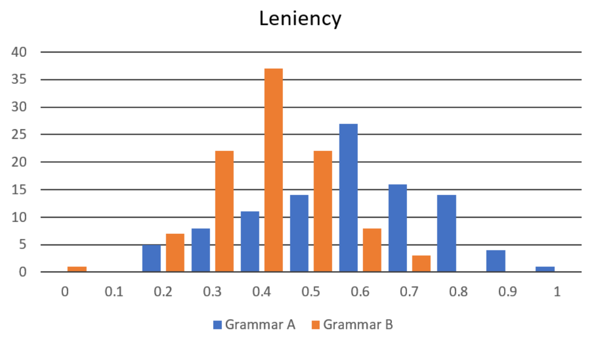
Figure 5.8: Histogram comparing the density of 100 levels created by different grammars

Leniency

As for the leniency of the levels created by the three grammars, it is not surprising to see that once again levels generated using grammar C have more diversity in difficulty due to the variety of mission maps it can create and the fact that it produces mission grammars with more nodes. It is also interesting to observe grammar A tends to create more lenient levels than grammar B, this is most likely because grammar B rules create more enemy nodes.



(a) Comparing the three grammars



(b) Comparing grammar A and grammar B

Figure 5.9: Histogram comparing the leniency of 100 levels created by different grammars

5.1.5 Review

The data analyzed shows that the generator offers some flexibility in handling different mission grammar designs, being able to create valid levels with two of the grammars used, but it has trouble dealing with grammars containing production rules that create graphs with multiple overlapping branching paths. Since the space map is created purely by following the mission map the edges between mission nodes tell the space map generator where to place the next prototype structure in relation to the position of the current structure, and a validation is made to see if the new position will collide with any structure already placed. If so, the structure is relocated to try a place it elsewhere in the map. The data shows this happens more often in mission graphs with multiple side paths especially if the pathways are clustered together. The data also indicates that this repositioning behaviour is incomplete and flawed because it can leave a level without platforms or connecting structures, which might make it impossible to complete. Or, it can mess up the creation of the space graph in such a way that a path from the start to the end of the level does not exist. To improve this validation, when a structure is repositioned because it would clash with other structures already in place, it should first be examined to see if the player can move from the current structure to any of the structures implicated in the collision. This dependence on the shape of the mission map for positioning puts a lot of responsibility in the design of one grammar, to not design rules that create too much branching or overlapping. A better solution would have been to follow Dorman's [6] idea of creating a separate grammar for the shape of the level. In fact, this may be taken a step further by developing additional specialized grammar targeted to deal with certain aspects of the level, such as a grammar for platform position, another for enemy types, another for enemy positions, and so on.

5.2 Analyzing participants data

To further analyze the capabilities of the "*2D platformer generator*" prototype, it was essential to gather feedback on its effectiveness. For this reason, the prototype was made publicly available for playtesting. Allowing a broad range of participants to provide feedback on their experience playing the generated levels. The study intended to evaluate the generator's ability to produce game levels that were both feasible and engaging. As part of the study, participants played a variety of levels generated by the "*2D platformer generator*" and completed a survey to rate their experience. By collecting feedback from participants on their experience with the generated levels, the study aimed to gain insights into the strengths and weaknesses of the generator and identify areas where improvements may be required.

5.2.1 Test procedure

As stated, during the study, participants were asked to play a set of ten levels generated by the "*2D platformer generator*". The levels varied in terms of difficulty and size, providing a range of challenges for the participants to complete. Upon finishing all ten levels, participants were then asked to complete a survey consisting of 26 questions that evaluated their experience with the game.

Levels generated

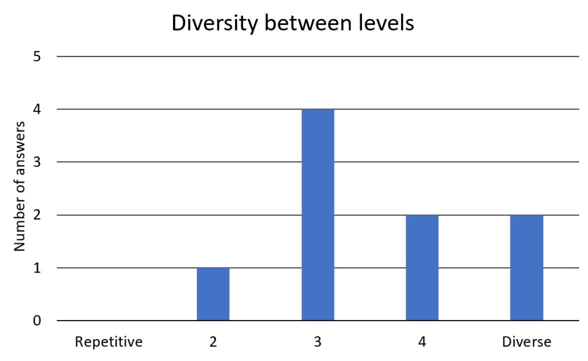
All of the levels were generated using mission grammar A, which has better feasibility, but with four different settings: easy, medium, hard and very hard. The first three levels were generated with the easy setting, this setting only allows two iterations to transform the mission map with the grammar, the number of enemies and coins are set to a low value, and it prefers mission maps with more edge slopes equal to 0 and the player simulator is set to easy. This means that it generates smaller horizontal levels with small gaps and few enemies and coins. Levels 4, 5 and 6 were generated with the medium setting, this setting allows four iterations to transform the mission map with the grammar, and the number of enemies and coins are set to a medium value, it also prefers mission maps with a small number of edge slopes different than 0 and the player simulator is set to medium. This means that the levels generated are a bit more vertical and longer than the easy ones, they also have wider gaps and more coins and enemies. Levels 7, 8 and 9 were generated with the hard setting, in this setting, the number of iterations is set to five, and the number of enemies and coins are set to the maximum possible, it still prefers mission maps with a small number of edge slopes different than 0 and the player simulator is set to hard. Another parameter that was changed is the percentage of a gap containing a moving platform, this percentage was made smaller to make the player have to jump instead of waiting for the platform. This made the generated levels as vertical as the medium ones but longer in size, they also have even wider gaps with fewer moving platforms and more coins and enemies. Finally, level 10 was generated with the very hard setting, this setting is very similar to the hard setting, the only changes are the number of iterations is set to 15 and it prefers mission maps with as many edge slopes different than 0 as possible. This generates levels similar to the hard ones, but a lot longer and vertical with a bit less linearity. An interesting side effect of this setting is that sometimes the end goal is positioned in the middle of the last quarter of the map.

Survey

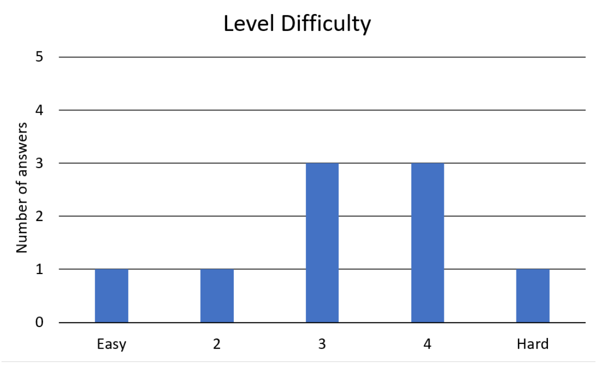
The survey consisted of 26 questions divided into 6 sections. The first section serves as an introduction and it contains a question to see how often the participant plays video games. The second section focuses on the participant's general experience when playing the prototype's levels, it questions the participant about the level's size, variety, difficulty and enjoyment while playing. The third section is only for participants who have played other 2D platformer games before and focuses on the participant's opinions of the prototype when compared to these other games. The fourth section focuses on the generated level's layouts and the participant's experience navigating through them, it inquires how frequently participants become lost and had to backtrack, as well as whether they could backtrack. The fifth section is a less important section about the creation of new grammar rules, unfortunately, no participant tried to change the rules. And finally, the last section is a closing section asking for extra user suggestions and comments.

5.2.2 Results

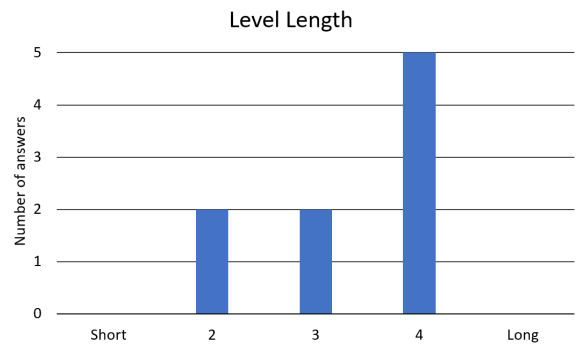
Unfortunately, only nine participants filled out the survey, so the results presented here don't have the diversity needed to do a proper evaluation of the prototype, still, this small feedback provides us with the opportunity to see how people react to playing the levels in the prototype. All of the participants play video games, with seven of them playing frequently. In addition, all participants had previously played a 2D platformer. When analysing the participant's answers, we see that in terms of diversity between levels, 44% of the participants thought that the ten generated levels were different from one another (see chart in Fig. 5.10a). While another 44% were unsure whether the levels were different or not, even so, all participants thought the levels were different enough to be enjoyable. This uncertainty is predictable because only the location of elements changes from level to level, while the aesthetics of the levels remain constant. If the background and style of platforms changed as well it would be more noticeable, however, this could lead participants to believe that the level has changed when, in fact, just the aesthetics were altered. In terms of difficulty, the majority of the participants thought the levels were challenging enough to be entertaining with only one thinking otherwise, from the chart in Fig. 5.10b we see that this challenge was not too difficult, with 67% of the answer being split between the middle of the scale and value above it. In terms of level length, the chart in Fig. 5.10c shows that 56% participants thought that most of the levels were more on the long side, but also thought that the quality was about the same as the shorter levels, with only one person saying otherwise. In terms of enjoyment, it was pleasant to see that the majority of participants believed playing the levels was enjoyable, with only two participants answering in the middle of the scale (see the chart in Fig. 5.10d). With the exception of two participants, the majority of users believed that there was more than one way to reach the goal and no participant believed it was tedious to search the map for coins or the goal.



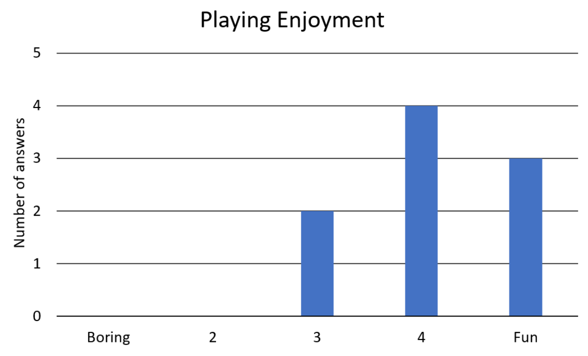
(a) Level diversity answers



(b) Level difficulty answers



(c) Level length answers



(d) Level enjoyment answers

Figure 5.10: Charts showing the participant's general experience while playing the generated levels in terms of diversity, difficulty, enjoyment and level length

The participant's experience while navigating the level was examined, by asking the participants how often they experienced the following scenarios: getting lost, backtracking, not being able to backtrack, and having to skip the level because they could not finish it. From the chart in Fig. 5.11, we can see that the majority of participants either never got lost or got lost rarely. Looking at the "backtracked" answers, we see that even if people did not get lost they had to backtrack sometimes, and the majority of participants could backtrack most of the time without problems with only two participants reporting problems rarely and an outlier having problems often. We can also see that only two participants had to skip levels.

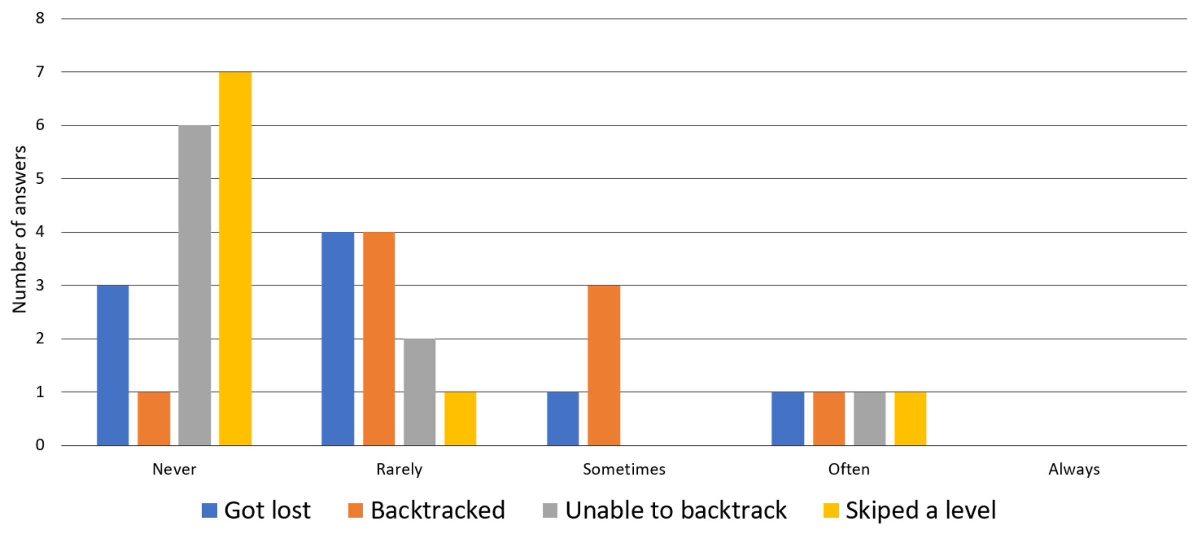
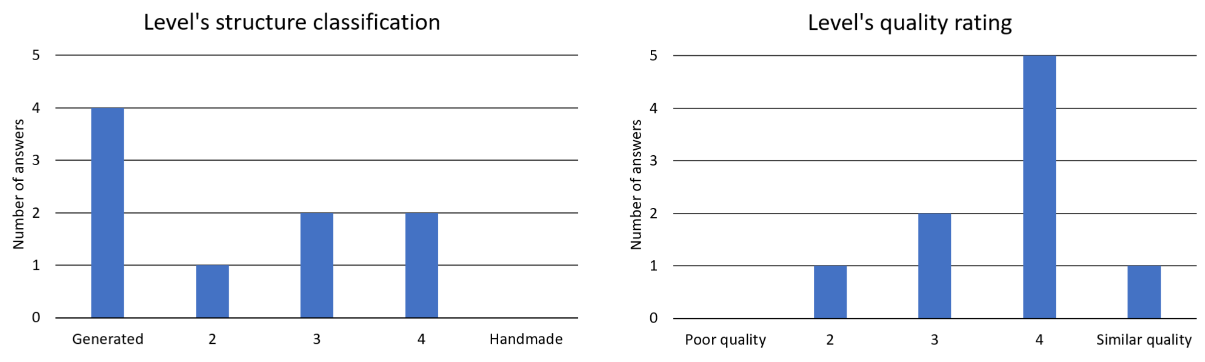


Figure 5.11: An histogram showing the frequency of participants' answers in regards to four scenarios while navigating the levels

When comparing the generated levels with levels from other 2D platformer games, the participant had played before. The participants were asked to rate the quality of the levels and classify the layout of the levels to see if they looked generated or handmade. From the charts in Fig. 5.12, we see that 44% of the participants classify the prototype levels as generated and only 22% think that the level layouts look handmade. In terms of quality, most of the participants said it had about the same quality as the levels from the other games, this was surprising, but not very helpful or meaningful information with such a small sample of participants.



(a) Level classification answers (b) Level quality answers

Figure 5.12: Two charts showing participants' opinions of the generated levels when compared to levels from other games they played

Among the comments left by the participants, some suggested improvement on certain aspects of the gameplay. Suggestions, such as increasing the collision box size of coins to make them more easily collected, or a way to pan the camera down to obtain a clearer view of whether or not there is a platform below while exploring the level. One participant, was interested in seeing the level regenerate after the player dies in the level, saying quote: *"Regenerating the map on a death is an interesting idea that makes the player unable to memorize the level, but instead have to actually play through it."*. Finally when asked all the participants were interested to see more games created this way.

Chapter 6

Conclusion

This work represents the begging stages of developing a generator framework that uses graph grammars to generate levels for a platformer game, a prototype game called "*2D Platformer Generator*" was created using Joris Dorman's methodology of creating action-adventure levels by separating the generation in two domains the mission and the space, but altering it to work with platformer games. To test the performance of the generator, three distinct mission grammars were designed to generate levels, which were then evaluated according to four characteristics linearity, feasibility, density and leniency. The grammar that showed the best feasibility was then play-tested by nine participants. The results showed that the generator can generate playable and engaging levels, with most participants saying that they enjoyed playing them, but the results also showed that, in its simplicity, the generator has trouble dealing with more convoluted rule designs and that the validation system needs to be better tuned, especially the use of the player AI simulator to determine whether a level is feasible or not. This chapter will conclude this work by reflecting upon the methodology used and what could have been done better, it will also mention the interesting future work that can still be done.

6.1 Methodology Reflections and future improvements

Upon reflection, if a future study were to be conducted there are many key points that should be considered and learned from this study. In other to have more refined control over how the grammar is applied to create the mission map, a custom graph rewriting system was created, this brought more control but also came with the cost of time. Creating this system took some time that could have been put into analyzing a better rule system, better space layout validation, better game design, or even testing the generator itself. The custom rewrite system also comes with the price of not being session tested, many problems were found and resolved, but many more still probably remain. So for future research using an external better tested library could save a lot of time and headaches at the cost of control. That being said, the custom system created could likewise serve as a foundation to be improved upon.

Another issue to consider is that, even though the level generation was separated into the creation of player steps with the mission map followed by the creation of the level's layout with the space map, only the mission page is moulded by a grammar. This created two problems, first, the layout of a level is very attached to the syntax of a mission map, even if through the use of non-deterministic grammars the level layout can look different but it will still be bound by the mission map syntax, this attachment constraints the use of a single mission map to creating level layouts with the same feel instead of being used to create a multitude of thoroughly different space maps that have the same goals. The second problem created is that the space layout has to rely on little information to decide where to place a level structure and to validate its placement, in the case of the prototype this was the mission edge slopes. This limits the solutions available and leads the space map into making decisions, such as repositioning a structure, without the context of how that could affect the structures around them. A better approach to try and minimize these problems would be to use multiple refined grammars with finer responsibilities, by for example creating a grammar that is responsible for the layout of platforms and another that is responsible for placing doors and locks. These grammars could then be connected by using the terminal symbols of one grammar as the pattern required to apply rules from another grammar. Or link the use of grammars by stipulating rules in another grammar that serves as a coordinating grammar. This affords the flexibility to select the most suitable type of grammar and formulate a functional syntax for a specific aspect of the level design. In the given example, this approach allows for the validation of the platform layout by selecting a more suitable shape grammar and relying on its syntax for validation. Similarly, when adding doors and locks using another grammar, only production rules that adhere to the connection condition between the grammars are applied, and the position of the doors is validated by the grammar's syntax if the platform map is grammatically correct. Although, creating such a system can be a challenging task, separating the responsibilities into multiple grammars gives the developers more refined control over the generator behaviour and output and makes the debugging process more manageable.

A different point to think about and improve, is the validation of whether a level is feasible or not. In this study, this is validated with the help of a player AI simulation, where each node of the shortest path from start to goal in the space map is visited by the player AI. During this process tests are made to see if any structure blocks the path of the player AI or if the player AI is capable of jumping to the structure of the next node in the path. The simplicity of this validation makes it run very fast, but as shown by the results it has a big error margin. There are several reasons for this error, first, only the shortest path is tested, some of the structures in this path might have been placed too near to other structures of other paths, meaning that the way is blocked to the player AI, but in reality, a real human player would just take another path to the goal. So for better validation, there should be a number of paths tested before labelling the level as not feasible. Another reason for the error in determining level feasibility is that in order to make sure that the levels were definitely playable, during the level generation the player AI simulation jump was used to determine the distance between structures, this was done by only calculating the arc of the player jump to see how far the player can reach if he was standing at the edge of a structure and jumped. The same AI and calculations are then used to see if the player can traverse a level candidate, but of course, only simulating the arc of the jump does not take into account interferences from external sources that might occur nor does it take into account the speed the player is travelling at. So for future research using a proper platformer player AI and actually simulating a whole section of the path being considered would yield better results.

Another point to reflect upon is the gameplay of the levels, the prototype created for this study offers a very simple gameplay, it only contains one type of enemy, one type of collectable, and the platforming challenge of going from one side of the level to another. Such a simple game worked to test the basic capabilities of the generator, but a much larger game that features a multitude of enemies, collectables, and other parts of the game that take place in more complex situations, could stress test the generator even further, probably providing more insight into ways of improving the generator. It would also offer the human players a more interesting and familiar experience while giving them more content to comment on or find flaws in. A familiar experience would also allow the human player to more easily compare with other games they have played and give more interesting feedback. For example, one simple change that could improve the gameplay, would be to let the player walk in front of some types of platforms, instead of only above the platforms. This would allow platforms to be closer together and create more interesting landscapes and level layouts. Another change that should be considered to improve gameplay, is solving pointless dead-ends in the level. This could be done by removing some of these dead-end paths completely, or by connecting them back with the main path with a connector structure, or even, making the trip down these dead-ends rewarding by placing key items or rewards in them.

Finally, the sample of human participants who tested the prototype was quite small, which means there is no guarantee that the results collected represent the thoughts of a larger sample or the population. More participants would be highly recommended in future research.

6.2 Future work

Besides improving the current generator, there is still a lot more to explore and do. One of the most tedious aspects during this study was having to manually write the grammar rules in *Json* format. An interesting project would be to create some form of graphical user interface (GUI) that would help the developer create graph rules more easily, and keep track of the grammars created. In an even more ambitious version, the GUI could allow the preview of how the new production rule would most likely affect the levels created by showing the effect on an example level.

But depending on the game requirements, creating grammars from scratch can be a time-consuming and complex process. So another interesting research project would be to turn to the field of artificial intelligence and use machine learning to automate or help create grammars. Machine learning algorithms could be trained to learn the structure of a graph and the grammar that defines it. The algorithm could then be used to generate new graphs with similar structures. The first step in generating graph grammar with machine learning is to collect a large number of existing game levels represented as graphs from a variety of sources, such as user-generated data, research papers or online databases. The graph representation could be based on the level structural layout or even the rhythms of actions performed by a player when playing through a level. The collected graphs could then be used to train a machine learning algorithm, such as a deep neural network, to identify the structure of the graph and the rules that govern it. Once the algorithm has been trained, it can be used to generate new graphs with similar structures. This can be done by feeding the trained algorithm a set of level graph patterns the developer wants to design and asking it to generate new graphs based on those patterns. A better-trained algorithm could even help the developer by giving him feedback on the effects a new production rule could have or even suggesting production rules.

6.3 Final thoughts

In conclusion, this project showed that even in a simple form, the use of grammars as a rule system to generate levels is very feasible and flexible, allowing a developer to easily change the layout of the levels generated, it also revealed that Joris Dormans's approach of separating the creation of an action-adventure level into a mission domain and a space domain, not only works when applied to 2D platformer games but also facilitates the process. In fact, it indicated that the use of multiple targeted grammars applied in small amounts could yield better results and be more easily manageable. This is the small contribution made by this project to the PCG field of study. PCG is a vast field of study, as more research is done in it, the more fascinating and intriguing it becomes, surprising us with what it can accomplish and the wonders we can create with it.

Bibliography

[1] Will Wright. The future of content. Talk by Will Wright at the 2005 Game Developers Conference.

[2] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[3] Derek Yu and A Hull. Spelunky. A PC game published by Mossmouth, LLC, Xbox Game Studios and Microsoft Studios, 2009.

[4] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*, chapter 3, pages 49–51. Springer, 2016.

[5] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[6] Joris Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, pages 1–8, 2010.

[7] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation. In *European Conference on the Applications of Evolutionary Computation*, pages 141–150. Springer, 2010.

[8] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[9] Aristid Lindenmayer. Developmental algorithms for multicellular organisms: A survey of l-systems. *Journal of Theoretical Biology*, 54(1):3–22, 1975.

[10] Bernd Lintermann and Oliver Deussen. Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, 1999.

[11] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311. IEEE, 2012.

[12] James Gips. *Shape grammars and their uses: artificial perception, shape generation and computer aesthetics*. Springer, 1975.

[13] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[14] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[15] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[16] Ciaran McCreesh, Patrick Prosser, and James Trimble. The glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In *International Conference on Graph Transformation*, pages 316–324. Springer, 2020.

[17] David Adams et al. Automatic generation of dungeons for computer games. *Bachelor thesis, University of Sheffield, UK. DOI= http://www. dcs. shef. ac. uk/intranet/teaching/projects/archive/ug2002/pdf/u9da. pdf*, 2002.

[18] Kate Compton and Michael Mateas. Procedural level design for platform games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 2, pages 109–111, 2006.

[19] Unity Technologies. unity.com. https://unity.com/, 2005. [Online; accessed 2022].

[20] Unity Technologies. assetstore.unity.com. https://assetstore.unity.com/, 2010. [Online; accessed 2022].

[21] Peter Heimann, Gregor Joeris, Carl-Arndt Krapp, and Bernhard Westfechtel. A programmed graph rewriting system for software process management. *Electronic Notes in Theoretical Computer Science*, 2:127–136, 1995.

[22] TU Berlin. AGG. https://www.user.tu-berlin.de/o.runge/agg/, 1997. [Online; accessed 2022-11-16].

[23] UT Austin. Graphsynth. http://designengrlab.github.io/GraphSynth/, 2006. [Online; accessed 2022-11-17].

[24] Martin Mann, Heinz Ekker, and Christoph Flamm. GGL. https://github.com/BackofenLab/GGL, 2013. [Online; accessed 2022-11-17].

[25] Alexandre Rabérin. unity.com. https://kernelith.github.io/QuikGraph/documentation/history.html, 2019. [Online; accessed 2022].

[26] Diogo Soares. itch.io. https://jackgoggles.itch.io/2d-platformer-generator, 2022. [Online; accessed 2022-11-29].

[27] Diogo Soares. Generation and Evaluation of 2d Platform Games. https://docs.google.com/forms/d/e/1FAIpQLSdtXiSLktSAIDTalu_h89MZXuswdNSJ8vqIzwkgeXsz6w_obw/viewform?usp=embed_facebook, 2022. [Online; accessed 2022-11-29].

[28] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. A comparative evaluation of procedural level generators in the mario ai framework. In *Foundations of Digital Games 2014, Ft. Lauderdale, Florida, USA (2014)*, pages 1–8. Society for the Advancement of the Science of Digital Games, 2014.

[29] Martin Mann, Heinz Ekker, and Christoph Flamm. The graph grammar library-a generic framework for chemical graph rewrite systems. In *International Conference on Theory and Practice of Model Transformations*, pages 52–53. Springer, 2013.

[30] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation*, volume 1. World scientific, 1997.

[31] Wolfgang Kramer. What makes a game good. *Game & Puzzle Design*, 1(2):84–86, 2000.

Appendix A

Extra figures

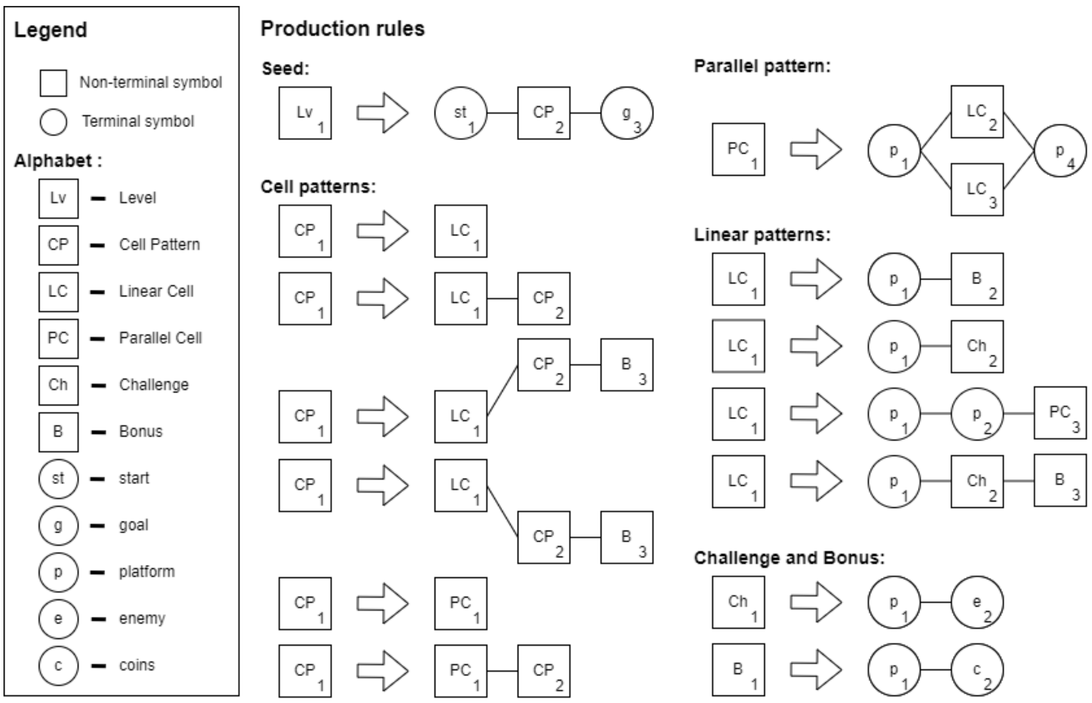


Figure A.1: Mission grammar A, used in this project's prototype game

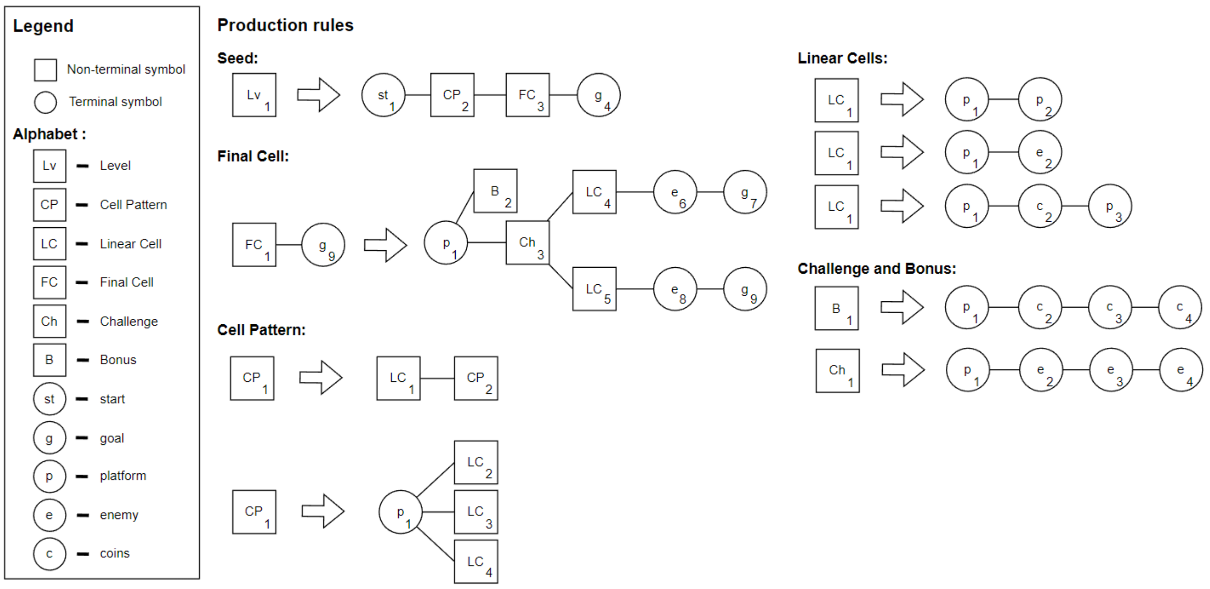


Figure A.2: Mission grammar B

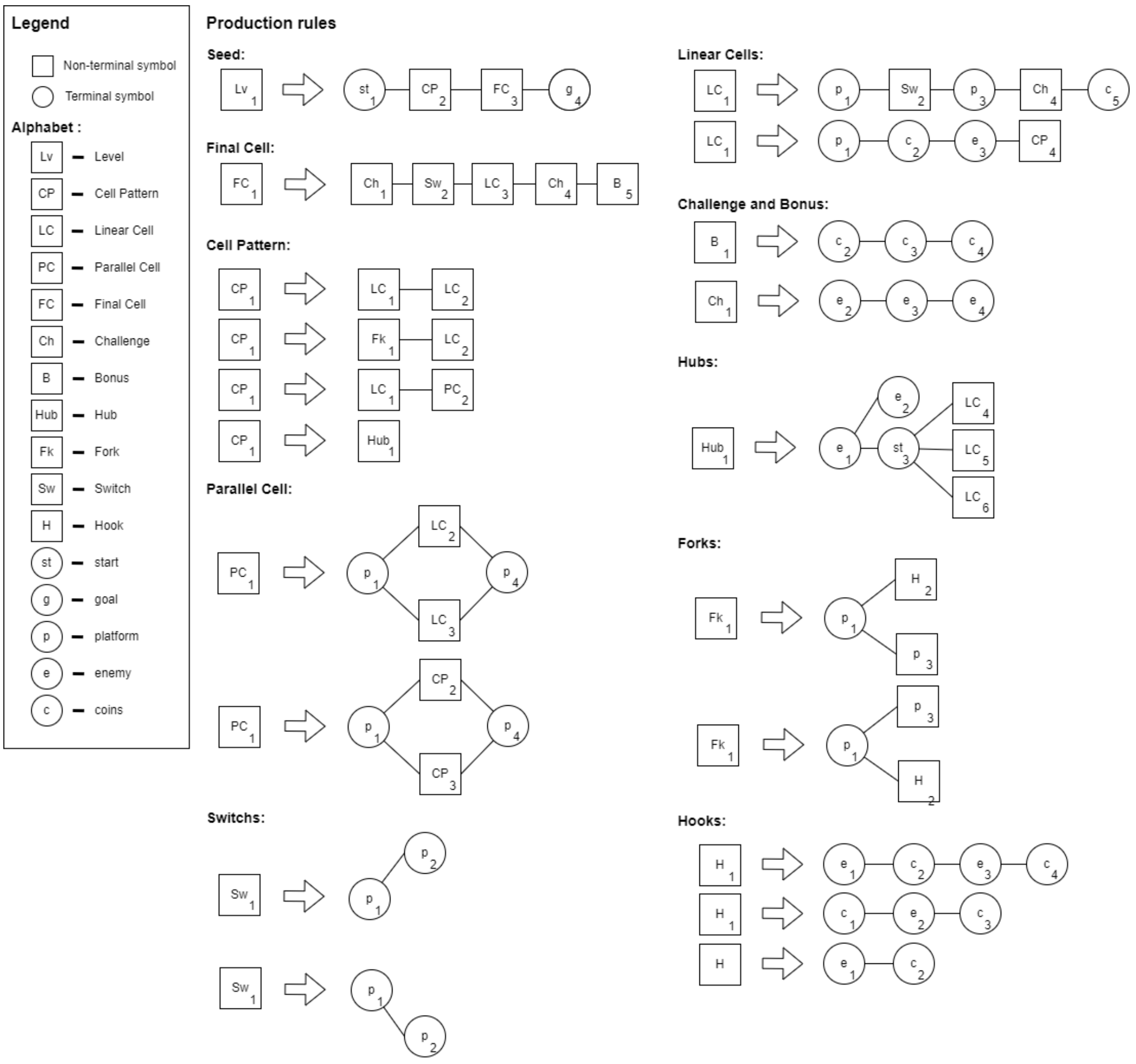
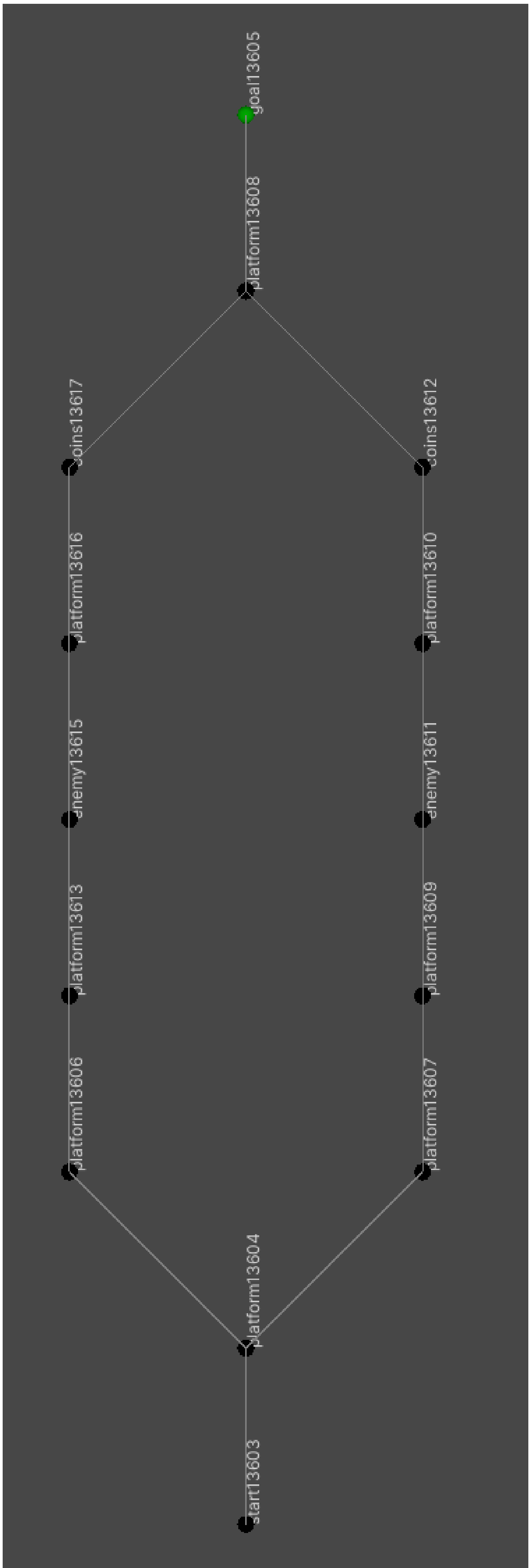
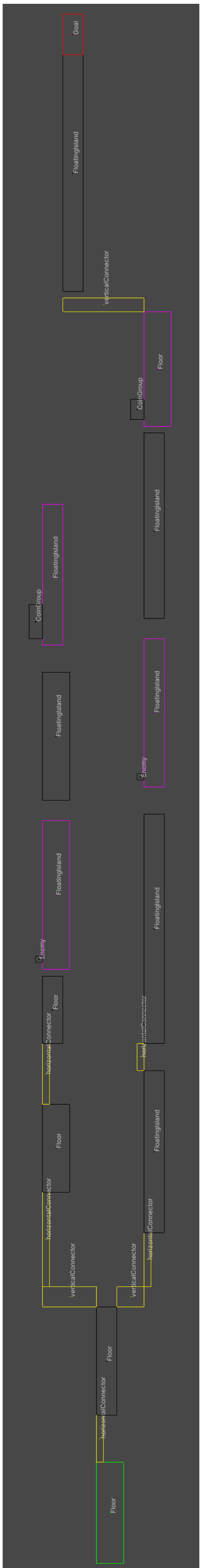


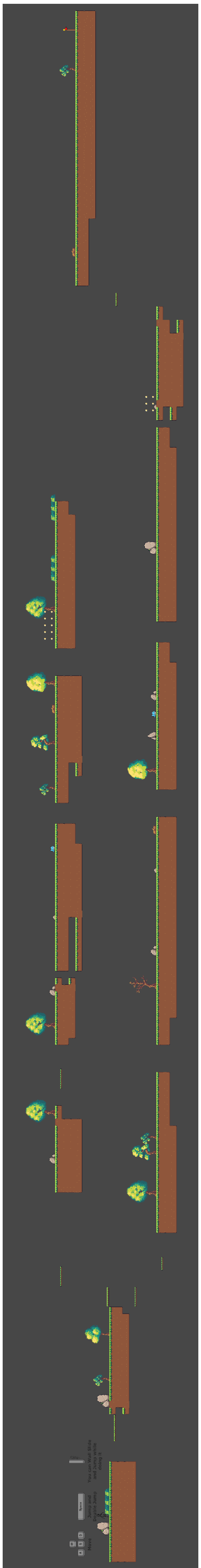
Figure A.3: Mission grammar C

```
{
  "Id": "example_rule",
  "Left": {
    "Vertices": [
      { "Type": "platform", "Alias": 1 },
      { "Type": "platform", "Alias": 2 }
    ],
    "Edges": [
        {
          "Source": 1, "Target": 2,
          "Type": "normal", "Slope": -1
        },
    ]
  },
  "Rights": [
    {
      "Priority": 0.5,
      "AssociatedRules": ["other_rule"],
      "Vertices": [
        { "Type": "enemy", "Alias": 1 },
        { "Type": "Other_Cell", "Alias": 2 },
      ],
      "Edges": [
        {
          "Source": 1, "Target": 2,
          "Type": "normal", "Slope": 0
        },
      ]
    },
    {
      "Priority": 0.5,
      "AssociatedRules": [],
      "Vertices": [
        { "Type": "enemy", "Alias": 1 },
        { "Type": "coins", "Alias": 2 },
      ],
      "Edges": [
        {
          "Source": 1, "Target": 2,
          "Type": "normal", "Slope": 1
        },
      ]
    }
  ]
}
```

Figure A.4: A composite grammar production rule written in *Json* format


goal13605
platform13608
coins13617
coins13612
platform13616
platform13610
enemy13615
enemy13611
platform13613
platform13609
platform13606
platform13607
platform13604
start13603


Goal
FloatingIsland
verticalConnector
Floor
CoinGroup
FloatingIsland
FloatingIsland
CoinGroup
FloatingIsland
FloatingIsland
Enemy
FloatingIsland
FloatingIsland
Enemy
Floor
horizontalConnector
horizontalConnector
FloatingIsland
Floor
horizontalConnector
horizontalConnector
verticalConnector
verticalConnector
Floor
horizontalConnector
Floor


Move
W
A
S
D
Space
Jump and Double Jump
You can Wall Slide and Jump while doing it