A Comparison Between Evolutionary and Rule-Based Level Generation

En jämförelse mellan evolutionär och regelbaserad generering av spelbanor

Peter Nilsson
Oliver Nyholm

Examen: Kandidatexamen 180 hp
Huvudområde: Datavetenskap
Program: Spelutveckling
Datum för slutseminarium: 2017-06-01

Handledare: Steve Dahlskog
Examinator: José Font

Springer Format

1 http://www.emena.org/files/springerformat.doc
A Comparison Between Evolutionary and Rule-Based Level Generation

Peter Nilsson¹ and Oliver Nyholm¹

¹ Malmö University, Faculty of Technology and Society, Nordenskiöldsgatan 1, 211 19 Malmö, Sweden

Abstract. Creating digital games and hiring developers is a costly process. By utilizing procedural content generation, game companies can reduce the time and cost of production. This thesis evaluates and compares the different strengths and weaknesses of generating levels for a 2D puzzle platformer game using an evolutionary and a rule-based generator. The evolutionary approach adopts the use of grammatical evolution, whereas the rule-based generator is a strictly controlled system using generative grammars. The levels produced by each method are evaluated by user studies and combined with an analysis of the time and complexity of developing each generator. The data presented by the evaluations shows that the rule-based approach was slightly more appreciated by users, and was also less complex to implement. However, the evolutionary generator would scale better if the game would expand, and has the potential of performing better in the long term.

Keywords: Procedural, content, generation, PCG, grammars, grammatical, evolution, evolutionary, generative, level, puzzle, 2D, platformer

1 Introduction

The digital game industry is a large and continually expanding business. It is common that games today are developed by hundreds of people at a time and that it takes a year or more for them to be completed [4]. For larger as well as smaller companies, and even for independent game developers, human labour is costly; both in financial and time aspects [4][20]. As the industry keeps growing, the costs, development time and amount of developers needed for a game keeps growing as well. One of the reasons for this is that technology is advancing, and companies require increasingly
advanced hardware and software systems to be able to meet customers’ expectations [20]. This means that it is possible that fewer games will be profitable to develop, and that not as many developers can afford to develop games at all [4].

Among the potential hundreds of developers working to create games, several different roles can be found. There are programmers, artists, designers, audio and video engineers, et cetera. Out of all these roles, the more expensive ones for a company usually are the artists and designers, meaning that the content created by these developers usually is the more expensive.

Indubitably, companies want to reduce the cost of development, and by using Procedural Content Generation (PCG), developers can assign manual labour to the computer instead of, for example, artists and designers. PCG is a method used to algorithmically create different kinds of game content, and is mostly based on artificial intelligence (AI) methods. In games like Diablo [5], Minecraft [6] and No Man’s Sky [7] PCG methods have been utilized to create some, or most of their game content. In the aforementioned examples, and a lot of other games, the content generated is mostly landscape and terrain.

When it comes to research within the field of PCG, other areas of content generation, such as level and puzzle design, are less researched upon. Especially the combination of the two. One could argue that one of the reasons for this is that level and puzzle design usually is more intricate.

However scarce, research has been made on generating levels for 2D platformers [3] and physics-based puzzle games [2] separately, but how would the results be when combining the two genres based on earlier studies?

1.1 Purpose

In this thesis we therefore present two approaches for generating levels with PCG for a 2D platformer puzzle game, Madame Légume, which is a game created by us. See section 2.5 for a more detailed description of the game. We will also present an evaluation of the two level generation methods based on actual user input. The outcome of this evaluation will determine the strengths and weaknesses of the methods and their usefulness in creating levels for games similar to Madame Légume.

One level generator will be based on a concept where game objects are randomly placed in the levels and by using evolutionary search [3][4], the generator will weed out the worst and keep the best levels. The foundation for the second approach is determined by a set of rules, arranging game
objects according to them. These two generators will create several levels each and be tested against each other, evaluated by actual players to determine their worth.

The questions that we want to address in this thesis are therefore:

1. What are the strengths and weaknesses of the two generators?

2. Which generator is more suitable when creating levels for Madame Légume?

2 Background

2.1 Procedural Content Generation

Procedural Content Generation (PCG) is a way to automatically create content, e.g. for games or movies. By using algorithms [12], the computer can create content on its own or by using Mixed Initiative [4], a method where the software collaborates with a human to create content. In games, there is an array of diverse content that can be generated. Items, game rules, quests, terrain and complete levels are some of the various possibilities available. PCG can be used to generated game content either offline or online [4]. Creating content in an offline environment, this refers to using PCG during the development of the game, when large structures like maps and environment are assembled [4]. The counterpart, online generation, the algorithms are used during gameplay, allowing the game to adapt on the fly according to the player's playstyle or to generate a new level after each has been completed, resulting in a game with infinite levels [4].

There are several techniques viable for generating content, and knowing which method suits best for your own game can be complicated. Blizzard North used a method called Fractal Terrain to randomly generate dungeons for Diablo [4][5]. A way this technique can be applied to is by generating a straight path between the start and end, iteratively deforming the path a few times and randomly branching paths into new directions until the game area is filled [19].

Open world games require a large area for the players to explore, and to design and create every tree or bush by hand is time consuming. L-systems are a class of grammars which use an alphabet, and a set of production rules, to create strings that can be displayed as self-similar patterns [4]. They are
inspired by how plants grow and are therefore a suitable way to generate vegetation instead of doing it manually [4].

Another example of how content can be generated is by using a search-based approach, where the method uses an evolutionary algorithm in search for adequate or perhaps even the most optimal content [2][3][4].

As 2D platformer puzzle games is a relatively unexplored genre in procedural generation, it would be wise to compare two methods that are contrary to one another, measuring if one approach is better than the other. Two techniques that are considerably distinct are Generative Grammars [13] and Grammatical Evolution [9]. They are both based on grammars but differ in fundamental ways, as the former is more constructive, generating its results directly, and the latter uses evolutionary methods to find an appropriate solution.

2.2 Generative Grammars

Generative grammar is a method that hails from the field of linguistics and describes grammar as a system of rules that uses words to form grammatical sentences [13]. Applied to procedural content generation, it uses an alphabet that is made up of a series of arbitrary symbols that represent certain elements or concepts of the software being created, for instance a game. It also uses a set of rules, that dictate in what way these elements and concepts can be combined and connected to create wanted content. Specifically in games, this alphabet and its rules is therefore a way to describe a representation of the different pieces of the game, e.g. obstacles, enemies, treasure, start and end positions, etc., and how these connect to one another in order to create a functional level.

2.3 Grammatical Evolution

Grammatical Evolution (GE) is a combination of an evolutionary algorithm and a grammatical presentation [3][9]. The GE represents its content with genotypes and phenotypes [4]. Genotypes is the data in the code created by the grammar, stored in a string or in a vector of objects. The genotype can be seen as the blueprint for what is going to be constructed. The phenotype is what is later created from the genotype, resulting in the actual game objects presented in the game. This process is also referred to as a genotype-to-phenotype process. As done in genetic algorithms [8], a population of individuals are created. In this case, the entity created from the genotype-to-phenotype process is a unique individual to the population. Each individual in the population is
assigned a fitness value, determined by their attributes, during creation. Possessing a high fitness score increases the chance of passing one’s genes to the next generation during the genetic operation [4][8]. The genes are the different characteristics of an individual, e.g. world position, objects in the genotype, specific attributes etc.

2.4 Related Work

Narrowing down the field, we consider how others have implemented automatic level generation in similar game genres. N. Shaker et al. [3] and M. Shaker et al. [2] have both used Grammatical Evolution (GE) when developing levels for the game Super Mario Bros and Cut the Rope respectively. Super Mario Bros is Nintendo’s famous 2D platform game, released in 1985. Cut the Rope is a physics-based mobile puzzle game released by Zeptolab in 2010. The goal of the game is to feed Om Nom, a hungry frog-like character, a piece of candy by using physics-based components, transporting the candy in the right direction.

In both the aforementioned papers, they start off by generating the genotype using their design grammar. During the genotype-to-phenotype process, they differ to one another. In N. Shaker et al’s generator [3], the objects are created along the x-axis at random position, but still within the maximum distance the player can jump. In the generator created by M. Shaker et al, each object is given a random world position within the given parameters. After the phenotype is generated, both generators calculate a fitness score to the level and add it to the population for the evolutionary process.

When creating action adventure games, Dormans proposes a two step approach when generating levels [1]. In his paper, Dormans looks into how a temple can be created for game: The Legend of Zelda: The Twilight Princess. The first step is to create missions for the level using graph grammar, a specialized version of generative grammars, but instead of creating strings, graph grammar evolves into graphs with edges and nodes. This allows for sub quests and different solutions for completing dungeons. The second part of the generation is using shape grammar, a technique following rewrite rules as other grammars, but instead generating space. The shape grammar generates the world space for the temple based on the graphs of missions developed in the first step.
2.5 Madame Légume

The game on which the generators will be created and tested upon is a 2D platformer puzzle game, Madame Légume. The game is a project developed by us, created in Unity Engine. The player takes the role of Madame Légume, a lady made out of vegetables. The levels of the game consists of different platforms that the player can move around on and jump between. Some platforms might have blades sticking out of them. The blades are deadly, meaning if the player collides with a blade, the player needs to begin from the starting point again. Every level has an organic waste bin placed on one of the platforms, which acts as the end of the level, meaning that the goal of the game is to reach the organic waste bin. Reaching the end platform is not possible by just jumping from platform to platform, as a platform might be too high up for the player to reach or a gap is too wide to jump over, resulting in the player falling to its death upon trying. For this reason, the player needs to use the various edibles (explained in section 2.5.1) that can be placed in the world space to aid the player.

At each level, the player is given a set amount of edibles which the player can and must use to reach the end of the level. The puzzle aspect is therefore how the player should place the components so that it creates a path to the end point.

In figure 1, the player receives a banana and a tomato which has been arranged in a way that creates a path to the organic waste bin.

![Figure 1](image.png)

**Fig. 1.** Madame Légume, using the fruits to advance through the level towards the end point.
2.5.1 Game Components

Figure 2, shows the different components which the player can use in the game. Madame Légume (a) is the character which the player controls. She can place three different edibles. The carrot (b) acts as an extra platform which the player can stand on. The tomato (c) acts like a springboard, flinging the player into the opposite direction of where Madam Légume collided with the tomato (upwards if collision on top side of tomato, downwards if collision on bottom, etc.). The last fruit, the banana (d) is like a slippery dip, increasing the player's velocity along its shape.

![Fig. 2 The different game components.](image)

Displayed in Figure 3 are the different platforms generated for the levels. Each levels contains a start (b) and end platform (a). Platform 1 and 2 (c, d) can be generated for the level. Upon creation of platform 1 and 2, one or multiple blades (e) can be generated around the platform.

The goal of our generators is to be able to place all of these components in such a manner that the levels are playable, meaning that the player should be able to reach the end point using the platforms and edibles provided by the procedural generation.

![Fig. 3. The different platforms](image)
3. Method

3.1 Design Science

We have based our research process on the Design Science (DS) research methodology [10]. DS is a methodology, with a process including six different steps, used to create and evaluate IT artefacts with the purpose of solving certain identified problems [11]. The process involves, i.e., designing these artefacts, evaluating the designs, and finally communicating the results to appropriate audiences [10].

In more detail, the six steps of the DS process are:

1. **Problem Identification and Motivation**
   This step involves defining a specific research problem and justifying a solution for this problem [10]. The justification accomplishes both motivating the researcher and audience to pursue the problem, and understanding the researcher’s reasoning associated with the problem [10].

2. **Objectives of the Solution**
   This step involves rationally deriving the objectives of a solution from the problem definition and knowledge of what is possible and feasible [10]. These objectives can be either quantitative or qualitative.

3. **Design and Development**
   During this step, the artefact is developed, which includes: determining the artefact’s desired functionality, its architecture and finally creating the actual artefact [10]. There are different categories of artefacts, defined by Hevner et. al. [11]: constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices), and instantiations (implemented and prototype systems).

4. **Demonstration**
   In this step the artefacts ability to solve one or more instances of the problem is demonstrated, using an appropriate activity, e.g., experimentation, simulation or case study [10].

5. **Evaluation**
   In this step relevant metrics and analysis techniques are used to compare the solution objectives (step 2) to actual observed results from the artefacts use in the demonstration (step 4) [10].

6. **Communication**
   In this final step, the problem and its motivation, as well as the artefact
itself and its effectiveness, is communicated to a relevant audience, such as researchers or practicing professionals [10].

By following this approach, we aim to design two different solutions to the problem of procedurally generating levels for Madame Légume and similar physics-based platformer games, evaluate these designs to determine their usefulness and finally communicate our conclusions based on these evaluations.

3.2 Research Process

The purpose of this study is to evaluate both the generator created by the evolutionary approach and the generator created by the rule-based approach, when it comes to creating levels for a physics-based puzzle game such as Madame Legumé.

Our design research is done by creating two generators; one GE-generator and one heavily rule-based grammar generator. The generators will create playable levels for our game, Madame Légume, and will then be evaluated through player testing. This is because our belief is that the use of PCG should be a tool to create more and better experiences for the actual consumers of games: the players. Therefore we have decided to approach the evaluation of our content generators based on the opinion of actual players.

The GE-generator will be based on how M. Shaker et al. [2] have generated levels for Super Mario Bros and Cut the Rope. The rule-based generator is inspired by Dormans [1] approach of generating levels in two steps.

As our research process is based on DS research methodology, our work adhere to the six step inherent to the DS process, as described in section 3.1. Following is a short description of how we utilize these steps:

1. Problem Identification and Motivation
   With content creation being a costly and time consuming part of game development, we decided to further research within the realm of PCG. Narrowing the problem area to a simple game of our own creation, we figure that we can quickly and easily make contributions to the area of concern.

2. Objectives of the Solution
   Conducting a literature research, we look to related work and different PCG techniques that might help us better understand how to approach a solution to our problems. Further description of how we conducted this research and of the work we have based our own research upon can be found in section 3.2.1 and section 1.4 respectively.
3. **Design and Development**  
During this step we determine what functionality our generators require, what algorithms to use and how to implement them, as well as balancing the generators to make sure that one generator do not have any functionality that the other one does not. With development finished, we have created two instantiation artefacts [11] in the form of prototype systems, that will serve as the platforms for the true purpose of this study, namely the evaluation of said artefacts.

4. **Demonstration**  
In this step we demonstrate the value of our artefacts by presenting them to testers, allowing them to choose their preference between one level created from each generator at a time. By conducting our testing in this way, we are able to gather actual metrics that could be evaluated statistically to determine which generator produced the most preferable levels. See section 3.2.2 for a more detailed description of our data collection approach.

5. **Evaluation**  
The metrics obtained from our testers serve as the base of our evaluation, providing a direct insight into which generator is more preferable. However, on top of this, we conduct an evaluation between the metrics and the labour and complexity of the implementation of each generator as well, giving us a more detailed analysis of the functionality of the generators versus the cost of implementing them. See section 3.2.3 for a more detailed description of our data analysis approach.

6. **Communication**  
Finally, the results of our work and our evaluations has been brought together and is communicated through this thesis.

### 3.2.1 Literature Research

Understanding the problem with procedural level generation, we can get a better grasp of how to create successful generators. Therefore, it is necessary that we study earlier research on the topic.

To find wanted articles, we use the keywords “Procedural Content Generation”, “Graph Grammar”, “Grammatical Evolution”, “PCG”, and combining them with “Platform Games” and “puzzle games” to search in Google scholar’s database [15].

We also use the snowball sampling methodology [16] to find relevant related work, since several papers published in the realm of procedural
content generation are strongly connected to one another, both by the specific field and by the authors.

3.2.2 Data Collection

For the purpose of this study, and in concordance with the demonstration and evaluation principles of design science, the two generators will be put beside each other, and will be evaluated based on actual players’ choices. We will also knit these choices together with our own metrics, which involve implementation time and ease of implementation of the generators.

Combining usability and lab testing [17], the players will play the levels and rank them based on personal preferences. This will be done by using a rank-based questionnaire scheme [18], allowing players to test one level created by the rule-based generator and one level created by the evolutionary generator, and make a choice as to which one they prefer. Some of the test sessions will be held in the form of lab-testing, which means having each participant evaluate the levels one-on-one together with a moderator. The rest of the tests will be distributed digitally, together with clear instructions and the evaluations to be answered. Combining these two approaches ensures that we are able to cover a lot of ground, reaching a higher number of participants, as well as ensuring that the tests are understandable and relevant by supervising the play testers.

The test session will begin by having the player play a quick tutorial level, ensuring that the game design and mechanics do not interfere with the actual testing. The testers will include a post-task rating [17, p. 125] after each level, evaluating the level played.

After all levels have been evaluated, the tester will do a final post-study rating [17, p. 125], evaluating the levels as a whole. The post-study ratings will be collected as qualitative data, allowing the testers to freely write down their opinion on how enjoyable the levels were, if there was a challenge to solve the puzzle aspects, and if the levels felt like they were generated and what impact that had on the tests.

3.2.3 Data Analysis

To evaluate the data collected from our lab-testing, we will make a statistical analysis of the metrics from the test sessions and intertwine these results with the time and effort put into creating the generators. The metrics collected from the testers will be straight-forward and binary, allowing us to easily translate the results into statistical values. As mentioned in section 3.2.2, we will let the testers play one level by each generator at a time, and choose their preference between them. The binary questions they will be
asked to answer are: which level they prefer, which level is more difficult, and which level feels more computer generated. By collecting data by ranking choices in this way, we get answers that actually follow an underlying numerical scale and as such is better suited to be turned into statistics, as opposed to having the players rate the levels on, e.g. a Likert-type scale [18].

The first question, which level they preferred, is of most importance as we are looking for a generator that creates enjoyable content for the players. The two other questions, difficulty and look, acts as feedback on why the preference of that level might be the case. This will give us an idea of which generator that is a more appropriate approach when solving the problem of procedurally generating levels for Madame Légume, and will also provide us an insight as to why that might be the case.

4. Level Generation

4.1 Rule-based Generator

The first generator created is a strictly controlled, yet still non-deterministic, generative grammar system. It is partially based on how Dormans uses generative grammars to generate missions for adventure games [1]. However, as Dormans uses a specialized form of generative grammars called graph grammars, which produces graphs instead of strings to create more complex connections, we have chosen to use a basic generative grammar instead. This makes for a more linear representation which we feel is better suited to the type of levels we are creating for Madame Légume, as the 2D-layout, content and the fact that there is only one mission in the game makes it linear.

4.1.1 Grammar

The levels created by the generator springs from a generative grammar, that recursively adds certain symbols to a string. An algorithm is then used to read through this string, analysing the different symbols and creating game objects corresponding to these symbols.

The grammar used to create the genotype for these levels is depicted in figure 4, and can be interpreted like this: Firstly, the level will contain a start, an obstacle, and an end. Start and end, are what is called terminals, meaning they will not produce further symbols. Obstacle, on the other hand, is a non-terminal, meaning it will produce further symbols. The grammar then
proceeds to explain what Obstacle actually may produce. For instance, it might produce one horizontal platform and two more obstacles, or it might produce one vertical platform and one more obstacle. This means that, as long as the generator is not stopped, it will keep on producing symbols continuously, as every Obstacle produces at least one new Obstacle. In our generator, we run the algorithm 7 times before we add the terminal end, resulting in no more obstacles being placed and the generator going on to implement the objects based on our algorithmically produced string. By doing this we retain control over the size of our levels, and gives us greater possibilities of adapting the levels according to our needs.

\[
\text{Level } \rightarrow \text{Start } + \text{Obstacle } + \text{End}
\]

\[
\text{Obstacle } \rightarrow \text{Obstacle}
\]

\[
\text{Obstacle } \rightarrow \text{Horizontal platform } 1 + \text{Obstacle } + \text{Obstacle}
\]

\[
\text{Obstacle } \rightarrow \text{Horizontal platform } 2 + \text{Obstacle}
\]

\[
\text{Obstacle } \rightarrow \text{Vertical platform } 1 + \text{Obstacle}
\]

\[
\text{Obstacle } \rightarrow \text{Vertical platform } 2 + \text{Obstacle}
\]

\[
\text{Obstacle } \rightarrow \text{Horizontal platform } 2 + \text{Obstacle } + \text{Obstacle}
\]

\[
\text{Obstacle } \rightarrow \text{Obstacle } + \text{Obstacle}
\]

**Fig. 4.** The generative grammar used in the rule-based generator to create levels.

### 4.1.2 Rules

To ensure that the levels created will have a high chance of playability, challenge and entertainment, we apply certain rules to our generative grammar. In order to transform the grammars’ symbols into actual game objects’, we have created rules that determine in what way these objects can be instantiated, based on previous game objects. This affects at what position the objects will be placed, and also ensures that the platforms cannot go back from where they just came, that they cannot exceed the game boundaries, and that they cannot box in certain areas that would render the level unplayable, such as the starting point and the end point. On top of this, we apply a rule to every platform as it is being created, which gives it a chance of becoming inactive, thus creating a gap in the level and increasing the challenge of the level. Should a platform not become inactive however, another rule is applied which gives a chance of creating between 1 and 14 blades on the platform, adding another element of challenge to the level.

When creating this generator, we looked at how Dormans use graph and shape grammars to create levels for *The Legend of Zelda: The Twilight Princess*
[1]. Where Dormans uses graph grammars to generate missions, and then shape grammars to create levels to accommodate those missions, we have instead chosen to approach this in the opposite manner. Viewing the fruits in our game as our “missions”; the tools used to advance through the level, we have chosen to first generate the level, and then generate the fruit to create a playable path from start to end, as we find that this approach is better suited for this type of game.

After the platforms have been generated, we instantiate the fruits by going through the genotype representing the level created by the grammar, and placing it at appropriate positions based on distance and height difference to the closest upcoming platform that has not become a gap. For instance, if the next platform is at the same height as the current one, but is also far away, a carrot is placed between them, to simulate how a player would approach the problem. Similarly, bananas and tomatoes are placed if the next platform is lower or higher up than the current one, respectively.

By generating fruit in this way, we approximate how an actual player would engage the level. We also increase the likelihood of creating a playable level, as we ensure that the level contains fruit relevant to countering the obstacles created by the generator.

Fig. 5. The rules placed a banana and a carrot to overcome the gaps between the platforms
4.2 Evolutionary Generator

The second generator created is a Grammatical Evolution Generator (GE Generator) inspired by how M. Shaker et al. [2] generate levels for Cut the Rope. The phenotype is a one-dimensional list of game objects, containing platforms and edibles.

4.2.1 Design Grammar

The genotype is generated by following a set of grammar rules. A common way to represent design grammar is by using Backus-Naur form (BNF) [4, p.91], as can be seen in figure 6. BNF consists of four different types of symbols: \{ T, N, P, S \}. The symbol N stands for non-terminals. Non-terminals are symbols that will be rewritten to non-terminals or terminals, and the generation of the genotype continues until all non-terminals are replaced. \langle obstacles \rangle, \langle more_obstacles \rangle, \langle edibles \rangle and \langle more_edibles \rangle are all non-terminals, found in figure 6. Terminals are symbols which are not rewritten after generated. In figure 6, there are four terminals; \langle start \rangle, \langle platform \rangle, \langle edible \rangle and \langle end \rangle. Production Rules, P, are the different ways in which the rules will rewrite themselves, e.g. \langle more_edibles \rangle results into either a \langle edible \rangle or an \langle edible \rangle with another \langle more_edibles \rangle. Finally, the start symbol, S, is how the grammar starts, in our case the \langle level \rangle is our start symbol, containing two terminals and two non-terminals.

\[
\begin{align*}
\langle level \rangle &::= \langle start \rangle \langle obstacles \rangle \langle edibles \rangle \langle end \rangle \\
\langle start \rangle &::= \langle start \rangle(\langle edible \rangle, \langle start \rangle) \\
\langle obstacles \rangle &::= \langle obstacle \rangle \langle more_obstacles \rangle \\
\langle more_obstacles \rangle &::= \langle obstacle \rangle \langle more_obstacles \rangle | \langle obstacle \rangle \\
\langle obstacle \rangle &::= \langle platform1 \rangle | \langle platform2 \rangle \\
\langle edibles \rangle &::= \langle edible \rangle \langle more_edibles \rangle \\
\langle more_edibles \rangle &::= \langle edible \rangle | \langle more_edibles \rangle \\
\langle edible \rangle &::= \langle carrot \rangle | \langle tomato \rangle | \langle banana \rangle \\
\langle end \rangle &::= \langle end \rangle(\langle edible \rangle, \langle start \rangle) \\
\langle platform1 \rangle &::= \langle platform \rangle(\langle \langle \rangle, \langle \rangle \rangle, \langle \rangle, \langle \rangle, \langle \rangle, \langle \rangle) \\
\langle platform2 \rangle &::= \langle platform \rangle(\langle \langle \rangle, \langle \rangle \rangle, \langle \rangle, \langle \rangle, \langle \rangle, \langle \rangle) \\
\langle carrot \rangle &::= \langle carrot \rangle(\langle \langle \rangle, \langle \rangle \rangle) \\
\langle tomato \rangle &::= \langle tomato \rangle(\langle \langle \rangle, \langle \rangle \rangle) \\
\langle banana \rangle &::= \langle banana \rangle(\langle \langle \rangle, \langle \rangle \rangle) \\
\langle \rangle &::= [0, 30] \quad \langle \rangle &::= [0, 25]
\end{align*}
\]

Fig. 6. The level design shown in Backus-Naur Form (BNF)
Each level generated will contain a start platform, one or more platforms and edibles, and finally an end platform. The grammar will produce a string of rules which represents the different game objects that will be created for the level. Contrary to the rule-based generator, at creation, each object is given a random position within the x and y range without taking any other object into consideration. Only the start platform is limited to position 0 in the x-axis. Due to GEs using context-free grammar, the objects might overlap each other. Issues that might arrive due to overlapping is first taken into consideration during the fitness calculation.

![Image](image.png)

**Fig. 7.** Illustrating level generated. Genotype: <Start><Platform 1><Platform 2><Banana><Tomato><End>

### 4.2.2 Ensuring Playability

As for most other games, it is important that the game is playable. Especially for a puzzle game like Madame Légume, trying to solve an impossible level is a waste of time and not particularly fun. Having an agent simulate the level or player playing it would be the best way of ensuring playability. However, playing the level is very time costly and would result in the evolutionary process being extremely slow. For example, having a level being played would in average take around ~20 seconds. Playing every level in a population of 100 levels, during 100 generations
would result in approximately 55 hours (20 seconds * 100 levels * 100 generations).

M. Shaker et al. [2] evaluate the playability in their version of *Cut the Rope* by direct fitness, calculating the fitness of the level by a set of conditions. If all conditions are met, there is a huge chance of the level being playable. The solution for M. Shaker et al. [2] is not possible to implement in the same way for Madame Légume. In most cases of levels in Cut the Rope, the candy moves downwards towards Om Nom, the effect of components affecting the candy are constant and there are no obstacles intersecting the way (not in the generator made by M. Shaker et al. [2]). In Madame Légume, the player moves in all directions with platforms obscuring the way, and while edibles have an effect, the player doesn’t necessarily always follow the way.

The solution we therefore created was using a traversal algorithm, traversing between objects in the level, seeing if a path is formed between the start and end platform. The algorithm used was a depth-first search (DFS) algorithm [14, p. 93], where each platform and edible was seen as a node and given a projection area consisting of rays. Starting at the start platform, the algorithm checks if it is intersecting any non-visited platform or edible and delving deeper into the graph if a new object is found. If a path is formed between the start and end platform, the level has a high probability of being playable. The DFS cannot guarantee full playability, as the rays have a chance of reaching a platform and just barely missing an edge that the player will collide with.

![Fig. 8. The depth-first search algorithm ray casting a path from start to end platform](image)
4.2.3 Fitness Calculation

The fitness calculation is calculated by taking six different aspects into consideration. The first and most important aspect is if the level is playable by using the DFS algorithm. The playability check is done twice. During the first DFS run, all edibles are disabled from the scene, and the level is checked if it is playable without any edibles. If the level is playable at this stage, the fitness calculation ends and the level is given a set fitness value of 30. This is because a level that is completable without any edibles loses its puzzle aspect and therefore also the main point of the game; to solve the puzzle of how to complete the level. If the level isn’t playable without fruits, the fitness calculation is done again, but this time with the edibles enabled. If the level is playable with the edibles, the playability variable \( F(\text{playability}) \), where \( F \) stands for fitness, is given a value of 1. If the level is not playable, the value is 0.

The second aspect is connected with the results of the first aspect, as the fitness calculation checks which edibles were used to complete the level. If the level is not playable, the returned value is 0. In figure 8, both the banana and the tomato would be used to complete the level, whereas in figure 7, only the banana is needed. The \( F(\text{edibles}) \) variable varies depending on which components were used. If the level is playable, \( F(\text{edibles}) \) is calculated as following:

\[
F(\text{edibles}) = 1 + \text{notCarrotCount} \times 0.2 + \text{ediblesCombination} \times 0.2.
\]

The \( \text{notCarrotCount} \) variable is the amount of used edibles that are not carrots. This is due to carrots being simple and less exciting components used to solve levels compared to the tomato and the banana. \( \text{ediblesCombination} \) is a variable that rewards the levels for combining different edibles. If the level requires two different edibles to complete, \( \text{ediblesCombination} \) is given the value 1, and if all edibles are required, the value is 2.

The third aspect is the difference in \( y \)-position between the start and end platform. The variable \( F(\text{heightPosition}) \) is given simply by subtracting the \( y \)-positions of the end platform with the start platform. This gives \( F(\text{heightPosition}) \) a positive value if the end platform is above the start platform vice versa. The reasoning behind this is that the levels become more challenging when the player must climb upwards. The fourth aspect is \( F(\text{distance}) \). This variable is simply the distance between the start and end platform.

The fifth aspect is, like the second aspect, dependant on if the level is playable. The variable \( F(\text{endInSight}) \) keeps looking at each node in the graph
constructed by the DFS algorithm if the end platform is visible. By saying visible, it means if it is possible to cast a ray from the current platform to the end. For each node obstructed by an object to the end platform, $F(\text{endInSight})$ is added plus one. The larger the number, the more the player needs to navigate around objects to reach the end platform.

The sixth and final aspect checks if the edibles are intersecting other objects. This is a solution to solving the issue with randomly placing objects during the design grammar and also necessary to check since the player is not able to place edibles on top of other objects during gameplay. If no edible is colliding with another object, $F(\text{colliding})$ is given a value of 1. If one of the edibles does, $F(\text{colliding})$ is 0.

All six aspects are finally put together, calculating the total fitness of the level accordingly:

\[ \text{fitness} = 20 \times F(\text{playability}) + 10 \times F(\text{edibles}) + F(\text{heightPosition}) + F(\text{distance}) + 5 \times F(\text{endInSight}) + 10 \times F(\text{colliding}) \]

### 4.2.4 Evolutionary Process

The generator starts off by generating a population of 500 individuals and immediately discarding the 400 worst, keeping the 100 best for the evolutionary algorithm. This is due to the nature of randomly generating levels, a big portion of the levels are unplayable. By generating a huge population at start, there is a greater chance of evolving more playable levels. After each population has been generated, the 10 best are moved straight to the next generation. The individuals that are selected for reproduction are used by Tournament Selection [8] by size 2, where the individuals who win the tournament are also removed from the old population, not eligible to enter the tournament again. The crossover function is a one-point crossover with a probability of 70% [8]. Lastly, the evolutionary process has a mutation chance of 10%, changing the position of a random game object in the scene and switching the position of two other objects. The process of tournament selection, one-point crossover and mutation is repeated until the new population has reached 100 individuals. Each run lasts for 100 generations, and after all generations has been done, the best level produced is stored. This was done for 100 runs, generating a total of 100 levels, to ensure a wide selection of levels to choose from for the testing.
4.3 Generate-and-test

As the purpose of this study is to evaluate the two different approaches used to create levels for Madame Légume, we require sample levels produced by both generators to run tests on. Due to the rather wide range of quality of the levels produced by our generators, we need a way to ensure that the levels used for testing holds a relatively high standard and fairly equal quality. This is to reduce potential bias during testing, which could adversely affect the test results. We will generate and pick out 10 levels from both generators that we find successfully generated. This is a process called generate-and-test [4, p. 9]. The initial criteria for a successful level is merely that it is playable, followed by a test run to ensure that it is not trivial and that it contains some kind of puzzle aspect.

4.3.1 Levels Selected

In figure 9 are examples of four levels, out of total twenty levels, selected from the generate-and-test. Level a) and b) were compared against each other and c) and d) likewise. There were two different versions of the playtest. One where you first played level a) and then b), and another that was inverted, starting with b) and then a) instead.

When creating levels, both generators place the edibles directly on the scene. This is for us to make an easier, final evaluation about the level when choosing levels during the generate-and-test, as we can see the puzzle of how the edibles should be located to reach the end platform. But when the levels are saved for playtesting, the edibles are removed from the scene and given to the player instead, allowing them to figure out where they should be placed.

For level a), the player receives one banana to complete the level. For level b) the player receives one of each edible. Level c) receives one banana and one carrot, and finally, level d) includes one carrot and two tomatoes.
5. Evaluating the Generators

5.1 Rule-based Generator

The rule-based generator was easier to develop compared to the evolutionary generator, as it did not require equally complex algorithms, meaning it was not as hard to implement and debug. Since we worked on the game in such a small scale with only a few game components, there were only a few rules that were required to be implemented. As we adopted Dormans two step method of dungeon generation [1], the generator was able to ensure playability single-handedly by placing edibles at the correct location as a result of knowing how the level was generated in the first step.

The generator proved to be efficient and fast at creating new levels. By simply pushing a button, the generator quickly created a new level, making it fast and easy to use, which also suited our generate-and-test approach perfectly. In a lot of cases, the levels were dull and easy to solve, and recurring patterns of the platforms and edibles could often be seen. But this is not to say that the generator is unable to produce interesting content. Given the strict nature of the rule-based approach, the amount of fruits and spaces...
can easily be controlled, which translates to a higher probability of levels requiring more fruits and clever thinking in order to be completed. Adding this together with the fast and easy production of levels yielded a lot of interesting levels in the long run.

The generator does not scale well when including more components, however. As the rules governing the generator is quite strict and heavily adapted to the content currently available in Madame Légume, adding new components would involve modifying the algorithm and its rules for the generator to keep functioning properly. One would probably need to add completely new rules as well, depending on what kind of content that is added to the game. Also, as stated above, a lot of levels generated are quite alike, resulting in less diversity, and narrowing the space of content that can be generated.

5.2 Evolutionary Generator

Generating the content with design grammar and randomly arranging the objects on the game scene was not difficult and easily implemented. The biggest task was to solve the fitness calculation, since there are many characteristics to a level that could be taken into consideration. The playability check was obviously the biggest hurdle to overcome, and creating an efficient algorithm to solve the problem was a huge task alone. Since we chose the DFS algorithm with ray casting, we had to modify each game object to support the ray casting. Solving how the rays could cast and getting the DFS to work properly was difficult and time costly, especially when it does not fully guarantee playability.

The base for genetic algorithms are simple to create, but difficult to optimize to give the wanted result. Finding the balance where good levels are rewarded to repopulate to the next generation without ending up with heavy elitism was challenging. To allow the lesser levels to have a chance at breeding, we invested a considerable amount of time into testing different selection and crossover methods, and also trying out other fitness calculations.

Another weakness that the generator had was that it was not very efficient. In average, it took around 4 ½ minutes to complete one run of the generator, resulting in approximately 13 levels per hour.

The strengths of the generator is that it scales well with more components. The only thing needed when including more edibles or platforms is adding the support for the DFS algorithm to the component and including it to the
design grammar. Since the placement of objects is completely random as well, the generator has a broad spectrum of creating diverse levels.

Like the rule-based generator, most levels created were not exciting. Since playability was the aspect that rewarded the level with most fitness score, the generator more often created levels that required the player to descend down to the end platform; an easier task compared to traversing up since the player needs edibles and platforms to aid on the ascent. But from a run of generating 100 levels, there were enough good ones to suit our demand for testing.

5.3 Results of Testing

There were a total of 36 game testers, where the players were asked to compare two levels against each other for a total of 20 levels. The tests were done in a lab environment and also sent out for people to download and fill out the forms from home. Most testers completed all 20 levels, but some only did a few comparisons. In the end, there were 342 comparisons between the evolutionary generator and the rule-based generator.

As seen in figure 10, there was not a clear winner between which generator produced more fun levels. In the post-study-ratings, most players expressed that the levels were enjoyable, although some were too easy to complete.

![Pie Chart](image)

**Fig. 10.** The players enjoyed both levels, but the rule-based generator received slightly more votes.
Levels that included more edibles and those that required the banana were more fun and required more thinking of how to solve the level. In figure 13, the chart shows that 64% of the levels that were the most fun were also the level that were the most difficult ones between the two. However, in figure 11, the evolutionary generator was the one that in general created more difficult levels but was still not the most preferred one. Studying the results from the post-study ratings, several play testers found that some levels were more difficult due to timing when jumping and avoiding the blades, and not challenging in a puzzle aspect, which characterizes more towards the evolutionary generator. Also in figure 13, only 35% of the preferred levels were also the ones that felt more computer generated, which the play testers overall voted more on the evolutionary generator. In the post-study form, most testers did not find the levels feeling as if they were computer generated, and if they were, they did not find it having an impact on the level. The reasoning behind why some levels felt more generated than the other were blades sticking up from random locations where the player never needed to pass by and platforms that were at weird locations and inaccessible. This description is also more applicable on the evolutionary generator, as the chart in figure 12 below shows.

![Pie chart showing difficulty comparison between evolutionary and rule-based generators.](image)

**Fig. 11.** Evolutionary generator was more difficult.
**Fig. 12.** Evolutionary generator felt more computer generated.

**Fig. 13.** ~64 % of the levels where the player chose the favourite level, that level was also the most difficult one, whereas only 35 % found the level that was most fun also was the one that felt more computer generated.

In conclusion from the test data, a majority of players preferred the rule-based generator, because it more often generates levels that include more edibles and feels less computer generated. When the rule-based generator was the more difficult level between the two, a huge part of the play testers also voted on it as the favourite.
6. Discussion

Given the findings presented in the previous chapter, the two methods we chose to use for the study proved to be successful, as both generators were capable of generating levels enjoyable for the players. Thus, our initial question of the possibility of combining the two genres when procedurally generating levels, which had previously been studied upon separately [2][3], has been affirmed.

The study’s major finding was not that it was possible to combine the genres, however, but that the levels created by each generator also proved to be highly equal in terms of quality and enjoyability. The evolutionary generator was expected to deliver, as it had earlier been proven to work for the different genres [2][3] combined into Madame Légume. We were therefore more excited to see if Dormans approach [1] would work. The importance of the discovery from the latter concept shows that with a smaller amount of work, it is possible to create a generator that produces adequate levels when working on a small scaled game. With more tuning and work put into the design, the generator could potentially be used for online generation, allowing it to create levels on its own without the need of a generate-and-test approach.

From our assumptions ahead of development and user testing, we anticipated that the evolutionary generator would produce superior levels from the two. This belief was because the evolutionary generator would be capable of creating a wider variety of content, whereas the rule-based would have a set amount and in most cases generate levels that would be alike. Studying the levels created during development and generate-and-test, the rule-based generator created a familiar pattern more often compared to the evolutionary, but did not necessarily result in less superior levels. In fact, more often it created better levels than its counterpart shown by the playtesting. As previously mentioned in section 5.3, the levels that required more edibles and included the banana were seen as more fun. This is a strong argument to why the rule-based generator received the majority of votes when choosing the favourite level, as it is more generous when handing out edibles and bananas. In this part the evolutionary generator struggled, since the bananas peculiar use and size was more complicated to randomly place and create a path to the end platform. The levels created by the evolutionary generator would also be satisfied with only using one fruit, removing all the unused. Knowing this, the fitness function should reward the levels that included the use of more edibles and more score for bananas.
Our data also showed that levels with a more computer generated look more often was not chosen as the favourite level. Because of this, some sort of clean up function for unused and oddly placed objects could have proven to be valuable for improving the levels created by the generators. Had this been implemented, the results may have shifted in favour of the evolutionary generator, as it, according to our data, was more frequently perceived as computer generated.

The use of Madame Légume as a game for automated level generation can both be seen as a benefit and as a flaw. The benefit for us and possibly a flaw for the study is that Madame Légume is a game created by us, and is unknown to the players testing and others reading this thesis. This can be seen as biased evaluation from our part, since we are more reluctant to say that we produced inadequate levels, and we are the ones who know what type of levels Madame Légume should have. As it can be a flaw, it can also be a benefit for the testers that it is an unknown game, since they come into the tests without expectations. Compared to M. Shaker et al’s and N. Shaker et al’s studies on level generation for Super Mario Bros and Cut the Rope [2][3], if they conducted tests on them, the play testers might have experience on said games and come into the tests with expectations and preferences to what they find as preferred levels. Now, the users tested a new game and the assessment also resulted in being an evaluation of Madame Légume as well, and not just the levels.

As Madame Légume also is an unfinished and unpublished game, this also puts limitations on what was possibly to do during the time span of the study. It would have been interesting to evaluate the limitation of the generators further by having more edibles and components to generate.

Addressing our research questions, our results have given quite clear answers to the first one: both generators have merits although they both lack in their own specific areas. The rule-based generator for instance is easier to implement, has high controllability and is efficient, but would be harder to scale and often presents similar level patterns. The evolutionary generator on the other hand would scale a lot better and presents more diverse content, but is also more complex in its implementation, is less efficient and often gives a more computer-generated feel to the levels. The second question however, is harder to give a straight answer to. The rule-based generator was the more successful solution and could thus be viewed as the more appropriate approach when generating levels for Madame Légume. However, because the evolutionary generator scales better and provides more diverse content it could probably perform better in the long run. This means that the rule-based approach might only be appropriate within the confines of this study, and
that the evolutionary generator might be a better approach if applied to similar games or if Madame Légume expands.

For further studies, we are curious how complex the rule-based generator evolves to when including more game components. The study should compare what it is like to create the generator with a few components, and later including more when the initial generator is finished, measuring the time and effort for refactoring the code. The test could also compare how less complex it is when creating the generator with all components planned ahead. A second paper could be combining the two generators we present in this thesis into one generator, evolving the first population with the rule-based technique and later evolving them with the evolutionary method. The final suggestion we propose for future studies would be to combine one of the two generators with a mixed-initiative approach. This could prove beneficial, as human intervention could save a lot of levels that might only have needed a small change in order to become acceptable. It could also be a viable approach for reducing the computer-generated look, by manually ensuring that objects do not seem to be out of place.

7. Conclusion

Platform and physics-based puzzle games have previously generated levels successfully by using methods from the field of procedural content generation. In this thesis, we combine these genres into one game and research the possibility of applying PCG to generate levels for this game. We created two generators with different methods, and we were able to create levels with both techniques. We then performed user-based testing combined with an analysis of the development cost and complexity, to evaluate the strengths and weaknesses of the two. The rule-based generator created levels that were slightly more enjoyable for the play testers, showing that when creating a generator, it is achievable with only a limited amount of work when working on a small-scale game. However, when scaling to a larger game with more game components, the work becomes more complex. The generator utilizing grammatical evolution is capable of delivering a wider variety of levels compared to the rule-based, but developing the generator is more complicated to create, especially solving playability. Contrary to the rule-based, the evolutionary generator scales well with more components, suited for a larger game.

For future research, it would be interesting to examine what heights the complexity of the rule-based generator reaches when adding several game
components. We also believe that a combination of the two generators into one could work out well, creating the levels using the rule-based method and using an evolutionary method to find the best possible levels. Lastly, we suggest a study of generating levels with a mixed-initiative approach, which could further improve the content generated.

References