Procedural Content Generation for a Web-Based Motion Game to Increase the Variation and Progression of the Game

by

Dennis Persson
Examensarbete

Procedural Content Generation for a Web-Based Motion Game to Increase the Variation and Progression of the Game

by

Dennis Persson

LIU-IDA/LITH-EX-A--16/004--SE

2016-02-09

Handledare: Erik Berglund, Aseel Berglund
Examinator: Johan Åberg
Abstract

Computer games have always become more and more advanced. One of the biggest reasons to its rapid evolution is the use of procedural content generation (PCG), which is used to generate game content automatically. However, there is one type of games that is more unexplored when it comes to PCG, namely motion games. Motion games are games where the player interacts with the game by moving his own body rather than using a gamepad, mouse or keyboard. Thanks to that, motion games are a healthier alternative to regular games, and this thesis therefore explores the possibilities to use PCG to develop more exciting motion games. The focus lies on achieving variation and progression since both of those are important concepts closely related to PCG. An exploratory case study is also conducted to examine how the derived guidelines work in a real game. The result concluded is that the guidelines seem to work well, but that all of them are not easily adapted to every game. Different game genres therefore call for different guidelines to be used.
Acknowledgments

I would like to thank my supervisors Erik and Aseel who made it possible for me to do this thesis and continuously helped me with both positive and negative feedback. I would also like to thank all the people who have worked with me during this semester and that have made my days very enjoyable. I would of course also want to give a final thanks to my friends and family for supporting me throughout my whole education. You are all very wonderful people.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
</tbody>
</table>

1 Introduction 1
   1.1 Motivation ........................................ 1
   1.2 Aim .................................................. 1
   1.3 Research Question .................................. 2
   1.4 Delimitations ...................................... 2
   1.5 Background ........................................ 2

2 Theory 3
   2.1 Procedural Content Generation ...................... 3
      2.1.1 What is Procedural Content Generation? .......... 3
      2.1.2 Where and Why is PCG Used? ........................ 4
      2.1.3 Different Approaches to PCG ...................... 4
         2.1.3.1 Generation as Optimization .................. 4
         2.1.3.2 Generation as Constraint Satisfaction .......... 5
         2.1.3.3 Generation with Grammars .................... 5
         2.1.3.4 Generation as Content Selection ............ 5
         2.1.3.5 Generation as a Constructive Process ........ 6
   2.2 Game Entertainment .................................. 6
      2.2.1 GameFlow Model ................................ 6
      2.2.2 Rhythm Groups .................................. 8
   2.3 Motion Games ....................................... 10
      2.3.1 Entertainment in Motion Games .................. 10

3 Method 12
   3.1 Case Study .......................................... 12
   3.2 Iterations .......................................... 12
      3.2.1 Implementation .................................. 13
      3.2.2 Testing .......................................... 13

4 Results 15
   4.1 Iteration 1 .......................................... 15
      4.1.1 Implementation .................................. 15
      4.1.2 Testing .......................................... 17
4.2 Iteration 2

4.2.1 Implementation

4.2.2 Testing

4.3 Iteration 3

4.3.1 Implementation

4.3.2 Testing

5 Discussion

5.1 Results

5.1.1 Implementation

5.1.1.1 Game Design

5.1.1.2 Procedural Content Generators

5.1.2 Testing

5.1.2.1 Test 1

5.1.2.2 Test 2

5.1.2.3 Test 3

5.2 Method

5.3 The Work in a Wider Context

6 Conclusion

Bibliography

Appendix: Post-Study System Usability Questionnaire
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The bug game</td>
<td>2</td>
</tr>
<tr>
<td>4.1</td>
<td>Generated map with the constructive PCG algorithm</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Initial solution with the search-based PCG algorithm</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>Final solution with the search-based PCG algorithm</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Generated map with the search-based PCG algorithm</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Constant pattern solution</td>
<td>21</td>
</tr>
<tr>
<td>4.6</td>
<td>Pattern group solution</td>
<td>22</td>
</tr>
<tr>
<td>4.7</td>
<td>Seasons</td>
<td>23</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Design decisions ................................................................. 9
3.1 Task properties ................................................................. 13
4.1 Evaluation iteration 1 ........................................................... 15
4.2 Evaluation iteration 2 ........................................................... 17
4.3 Game modes ................................................................. 24
1 Introduction

1.1 Motivation

Game development has turned into a really tough industry in the last decades. Nowadays it is possible for ordinary people to design and release their own game. This has caused an explosive surge in the number of games available which forces developers to design their games more carefully to improve the user experience they convey. One way to do this is by ensuring that the games are varying and progressive. It can be very hard and time consuming to attain such qualities, wherefore procedural content generation (PCG) is a convenient method to use. PCG refer to the idea to generate game content automatically through suitable algorithms. In this context, content can be any content in the game, such as maps, levels, characters, terrain, stories or even AIs. The concept PCG is not something new, yet there are many more subdivisions to investigate. In this thesis, we delve into one of them, namely how to design PCG algorithms to function in a web-based motion game. Motion games, which in fact are a quite new and growing phenomenon, are games where the player has to move his own body in order to play the game. This way to interact with the game does imply many restrictions on the game design, due to the limited and unfamiliar user input possibilities. Additionally, a prototype game is developed as a part of this thesis, to make it possible to examine the design guidelines. The game is web-based and is similar to a tower defense (TD) game.

1.2 Aim

The main objective of this thesis is to investigate how to design PCG algorithms to acquire variation and progression in a web-based motion game. Moreover, a web-based motion game similar to a Tower Defense game will be developed as a case study, in order to validate that the chosen design solutions work.

\[\text{[1] Tower Defense is a genre that aims at preventing creeps to reach a certain point. The difference between a regular TD game and the game built in this thesis is how the player wipes out the creeps. Usually, this is done by purchasing towers that shoot at the creeps. In this game however, it is done by inducing a motion over the creeps.}\]
1.3 Research Question

The main question that this report will answer is

- How should procedural content generation be used in a web-based motion game to increase the variation and progression of the game?

1.4 Delimitations

This thesis focuses on PCG for web-based motion games where the player controls the game without any other devices (e.g., Nintendo Wii remote) except himself. Also, since PCG is a very wide concept, this report will mostly consider PCG for maps and enemy spawning. This is not a big delimitation though, since many other types of content (e.g., terrain and character) do not have a direct impact on the game mechanics, and may therefore not need any modifications when they are used in a motion game. Anyhow, the design decisions made in this thesis, to achieve variation and progression with PCG, are therefore adapted for games that match those two criteria. They could possibly also work for PCG construction generally, especially for TD games, motion games with devices and for other content than maps and enemies.

1.5 Background

The game that has been developed during this thesis is a further development of a game that already existed. The goal of the game is to prevent bugs from reaching and eating a tree. Meanwhile, the player must avoid splashing water drops in order to let them reach the tree. There are two types of bugs in the game: regular bugs that must be taken in a specific order and special bugs that can be squished anytime. The latter ones also have some additional effects on the game. If the player succeeds to let enough of water through to the tree, he will save the tree and the game will continue with a new tree to save. The saved trees will eventually work as a measurement of how much of the game the player has passed.

The initial game was completely playable. The game uses a webcam to perceive the motions of the player. As can be seen in Figure 1.1, it detects every motion and illustrates it on the screen. Every movement that sweeps over a creep (bug or a water drop) will splash them. At the beginning of the thesis, there was only one map in the game.

The enhancements that have been implemented in this thesis are two PCG algorithms, one to generate maps and one to generate the creep flow. Furthermore, there were only a few bug types available in the existing game. A few more bugs have been tried out in this thesis to be able to achieve more variation and progression in the game.

Figure 1.1: The bug game as it was when the thesis began. The human shaped shadow illustrates the movement that the player has performed.
This section contains necessary information and guidelines for games that satisfies the aim in this thesis. The subjects that will be addressed are PCG, motion games and game entertainment.

2.1 Procedural Content Generation

This section describes what procedural content generation really is and why it is so widely used. We will also look at some different approaches of how to design PCG algorithms, and discuss how and when they should be used.

2.1.1 What is Procedural Content Generation?

Procedural content generation is a way to automatically generate game content with the help of algorithms. The exact definition of it varies depending on who you ask. Togelius et al. reasoned about it and settled with the definition “the algorithmical creation of game content with limited or indirect user input” [7]. The word “limited” assures that user input is allowed, although in a moderate amount. “Indirect” broadens the definition to actually allow unlimited user input, as long as the user does not construct it directly. The reason to do that is to allow algorithms that are entirely based on user input, but are created in a way that makes it impossible for the user to foresee what the generator will produce. As an example of that, Togelius et al. created a procedural level generator to the Infinite Mario Bros game that is entirely dependent on how the user plays the previous level [7].

Assume that we agree with Togelius et al.’s definition. Then we still do not know what exactly “content” is. According to most people, content can be anything that appears in the game. Some examples could be characters, maps, terrain and the characters’ AI. In another report written by Togelius et al. they even included the story and rule sets as content [8]. So basically, one could say that there is a lot of content that can be generated with a PCG algorithm.
2.1. Where and Why is PCG Used?

The application possibilities for PCG is undoubtedly quite many. As mention earlier PCG can be applied to nearly every component of a game. One of the most common usages of it may be to generate maps though. Well known game titles like Diablo and Far Cry use it to do so. Even early games like Rogue (released in early 1980s) used PCG for that purpose.\[6\] \[1\]

Even if some generators can be tricky to implement, the benefits acquired from doing that tend to overweight the burden. The profits are so many more than just saving time by not having to manually program the content. Risi et al. developed a Facebook game called Petalz to profit from the variety that PCG is able to produce \[10\]. By automatically generating flowers they allow for an endless amount of different flower models to be created. Without PCG, this would not only be impossible, it would also require endless of memory. This is something that also David Braben and Ian Bell thought of when they developed Elite. Back in early 1980s, when Elite first came, memory consumption was a big problem. Despite that, Braben and Bell managed to store hundreds of stars with names, dimensions, locations etc. by generating them from a sequence of digits \[2\] \[6\]. In this way, they could ignore the memory problem and still have a massive game world.

In more recent games, PCG is used to create content that is much more concealed to the player. For instance, it may be obvious for some people that Diablo 2 automatically generates the map, but what is less clear is that they also generate scenarios \[6\]. Another example of the same thing is Left4Dead, where the spawning of enemies is dependent on how stressed the player is, or at least how stressed the game thinks he is \[6\].

2.1.3 Different Approaches to PCG

There are several different approaches to construct PCG algorithms. The three most basic ones are constructive, generate-and-test and search-based (SBPCG) algorithms \[8\] \[9\]. Constructive algorithms are those who construct a solution directly without testing it. For this to work, the algorithms has to ensure that the solution is valid while creating it, e.g. by following construction rules. This is something that is not done in generate-and-test algorithms. Instead, generate-and-test algorithms test if the solution is legal after having created it. If it is not, the algorithm will just discard the solution and create a new one from scratch.

SBPCG is a group of algorithms that uses different heuristics to search for a “good” solution. To know what a “good” solution is, SBPCG algorithms use a utility (or fitness) function to evaluate the solution. They can be seen as an extension of generate-and-test algorithms that alters a deficient solution instead of dismissing it. Even though the exact approach is not specified, most SBPCG algorithms are Evolutionary Algorithms (EA) \[8\]. EA use a set of candidate solutions and perform mutations inspired from biological evolution on them.

All three basic approaches to PCG are very general notions and not very specific to PCG only. A few approaches that are a bit more specific to the subject are the five approaches that G. Smith provides \[11\]. These five approaches are described in in the following sections. An imaginary nature game will be used as an example scenario in order to serve examples of how the approaches may be used. The choice of approach is dependent on many parameters. What content that should be generated, what the requirements are and if the content should be generated online (at runtime) or offline (in advance) are just a few examples. These approaches do not have to be mutually exclusive either. They are possible to combine to suit the demands on the game they will be used in.

2.1.3.1 Generation as Optimization

Generation as optimization means the same as SBPCG. The concept is to use search algorithms to find content that is valuable according to a criterion, i.e. the fitness function. The criterion can either be a human who is deciding what is good or bad, or a mathematical formula that results in a value or a vector with values. The purpose of the criterion is to favor
solutions that are rated higher by the developers. Generation as optimization is one of the most computationally expensive approaches. At the same time, it has the potential to produce very good solutions. Due to the heavy calculations, this method is not suitable when the generation has to be done in runtime. Since the criterion makes the algorithm to converge towards a specific solution it may in some cases have to be altered over time to achieve variety. Although, it can work well when it comes to progression, since the criterion can be seen as a goal that we slowly (or quickly) will converge towards. [11]

2.1.3.2 Generation as Constraint Satisfaction

These kinds of algorithms are typically search-based. The idea is to prune the paths to all bad/invalid solutions. This is done by declaring which properties of the content that must hold and which must not. An example of a constraint in the nature game would be that no Arctic animals are allowed in desert terrain. The reason to do this is to diminish the search space and thereby be able to find solutions fast even with mediocre search algorithms. If the goal with PCG is to achieve variety it may be a good idea to use generation as constraint satisfaction. Depending on how many constraints, this method may be able to produce very varied content if it uses a (partially) random search algorithm. The algorithm may also count as fast if there are few constraints and therefore many solutions. Since we already have specified constraints it may neither be necessary to test that the solution works. But of course, that depends on the strictness of the constraints. Either way, this method is appropriate to use when there are strict requirements regarding the solution. [11]

2.1.3.3 Generation with Grammars

This method is similar to the previous one. Although, a big difference is that this often is a pure constructive algorithm. While constraints only tells whether a property is legal or not, grammars specifies what must be done. It works by designing “rules”, which will be followed when the content gets generated. An example of a rule in the nature game could be: “if there is a desert, then put a camel in it”. When the algorithm generates a solution it will only do it by following the specified rules. Since generation with grammars is suitable as a constructive algorithm, the grammars should be designed to always produce legal solutions. A big difference between generation as constraints and generation with grammars is how to implement it and how the developers want to express their rules. [11]

2.1.3.4 Generation as Content Selection

Generation as content selection builds up the content (e.g. map) by putting together custom-made chunks (e.g. map parts), just as putting a puzzle together. The complexity of the algorithm is dependent on its type: constructive, generate-and-test or search-based. Generation as content selection is not always considered to be classified as PCG. However, let us look back at the definition Togelius et al. proposed for PCG: “the algorithmical creation of game content with limited or indirect user input” [7]. According to that definition, there is nothing that says that content selection would not be a full-fledged PCG approach. Thus it will be considered as a qualified approach in this report as well. With that said, content as selection is one of the simplest approaches to PCG [11]. It is suitable to use when the generation has to be done in runtime since it is technically speaking very fast to run [14]. The only thing that has to be done is to load another chunk and fit it together with the other one(s), and maybe also run a validity check. The method can feel repetitive since the same pieces shows up over and over again. This can be more or less varied by changing the chunk size and the number of chunks used.
2.1.3.5 Generation as a Constructive Process

Constructive generation is, as earlier mentioned, a method where content is created in a way that makes it unnecessary to test if the solution works. These generators are often built in the same way as a content selection generator but with smaller chunks. That makes them a little more complex, but they are still most often very time effective and are therefore suitable to use for PCG algorithms that will run online. One of the reasons to the low running time is that the solution does not have to be validated, since constructive algorithms always should be designed in a way that ensures that the solution works. [11]

2.2 Game Entertainment

One of the most vital aspects of a game is that it has to be fun. If the game is not fun, no one will play it. But what does it really mean that a game is “fun”? This is something that has been studied for a long time and therefore there exist many models for how to make a game fun. In this report, we will look at two of them: the GameFlow model and rhythm groups.

2.2.1 GameFlow Model

The GameFlow model is a model composed by Sweetser & Wyeth [12]. The model is based on an earlier concept called Flow [12]. The model can be seen as a guide for how to design games in order to make them fun to play. According to the model, there are eight fundamental factors that determine whether a game is fun or not. These eight elements are summarized in the following paragraphs.

Concentration
Games should require concentration and the player should be able to concentrate on the game

1.1) games should provide a lot of stimuli from different sources
1.2) games must provide stimuli that are worth attending to
1.3) games should quickly grab the player’s attention and maintain their focus throughout the game
1.4) players should not be burdened with tasks that do not feel important
1.5) games should have a high workload, while still being appropriate for the players’ perceptual, cognitive, and memory limits
1.6) players should not be distracted from tasks that they want or need to concentrate on

Challenge
Games should be sufficiently challenging and match the player’s skill level

2.1) challenges in games must match the players’ skill levels
2.2) games should provide different levels of challenge for different players
2.3) the level of challenge should increase as the player progresses through the game and increases their skill level
2.4) games should provide new challenges at an appropriate pace
2.2. Game Entertainment

Player Skills
Games must support player skill development and mastery

3.1) players should be able to start playing the game without reading the manual
3.2) learning the game should not be boring, but be part of the fun
3.3) games should include online help so players do not need to exit the game
3.4) players should be taught to play the game through tutorials or initial levels that feel like playing the game
3.5) games should increase the players’ skills at an appropriate pace as they progress through the game
3.6) players should be rewarded appropriately for their effort and skill development
3.7) game interfaces and mechanics should be easy to learn and use

Control
Players should feel a sense of control over their actions in the game

4.1) players should feel a sense of control over their characters or units and their movements and interactions in the game world
4.2) players should feel a sense of control over the game interface and input devices
4.3) players should feel a sense of control over the game shell (starting, stopping, saving, etc.)
4.4) players should not be able to make errors that are detrimental to the game and should be supported in recovering from errors
4.5) players should feel a sense of control and impact onto the game world (like their actions matter and they are shaping the game world)
4.6) players should feel a sense of control over the actions that they take and the strategies that they use and that they are free to play the game the way that they want (not simply discovering actions and strategies planned by the game developers)

Clear Goals
Games should provide the player with clear goals at appropriate times

5.1) overriding goals should be clear and presented early
5.2) intermediate goals should be clear and presented at appropriate times

Feedback
Players must receive appropriate feedback at appropriate times

6.1) players should receive feedback on progress toward their goals
6.2) players should receive immediate feedback on their actions
6.3) players should always know their status or score

Immersion
Players should experience deep but effortless involvement in the game

7.1) players should become less aware of their surroundings
7.2) players should become less self-aware and less worried about everyday life or self
7.3) players should experience an altered sense of time
7.4) players should feel emotionally involved in the game
7.5) players should feel viscerally involved in the game
2.2. Game Entertainment

Social Interaction
Games should support and create opportunities for social interaction

8.1) games should support competition and cooperation between players
8.2) games should support social interaction between players (chat, etc.)
8.3) games should support social communities inside and outside the game

As we can see, the GameFlow model influences the interaction between the player and the game, the interaction among the players, and also the game content. The interaction parts do not affect the design of PCG algorithms very much and will therefore be more or less ignored in this report, even if it has been kept in mind. The game content — or more particular — the first three (concentration, challenge and player skills) factors, are far more interesting to consider when designing a PCG algorithm. These three elements are more or less associated with variety and progression, which easily can be seen if we summarize some of the criteria into the following five statements:

- The game should catch the user’s interest quickly
- The game must be easy enough in the beginning
- The game should offer new experiences throughout the game
- The user should feel stimulated from different sources
- The difficulty of the game has to increase as the time goes by

However, the GameFlow model is of course about much more than just variation and progression. Some other criteria that are relevant to PCG are shown in Table 2.1 together with explanations of how to design PCG algorithms that meet the criteria requirement.

2.2.2 Rhythm Groups
A well-known term related to game development is *design patterns*. Design patterns can be described as standard solutions that can be adapted to the environment to solve frequently occurring problems [15]. Dahlskog & Togelius wanted to find a gamer specific notion of it and described it as: “a human or computer designed problem that reoccurs throughout the game”. The problem should be large enough to be able to be solved in various ways. This is the definition that will be used when referring to design patterns in this thesis. Dahlskog & Togelius also searched for patterns in Super Mario Bros and were able to identify 23 different patterns. The patterns they found were combinations of how to mix enemies, gaps, valleys, multiple paths and stairs together to form new types of problems. In this context, valleys are small rooms created by putting two vertical obstacles (such as pipes) with a fairly small distance in between them. [14]

A concept that is very closely connected to design patterns is rhythm groups. One could describe rhythm groups as design patterns without a geometrical representation. Design patterns include some type of graphical representation, even if the appearance of it may vary. Rhythm groups on the other hand, are totally geometric-free. Another way to explain it could be as patterns in the player’s input to the game. A single rhythm group would then be a certain time-dependent permutation of input [13]. A more accurate definition can be seen in the quotation below. The quotation is taken from a report about rhythm-based level generation for 2D platformers written by Smith et al. [13].
### 2.2. Game Entertainment

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Players should not be distracted from tasks that they want or need to concentrate on</td>
<td>Content that does not contribute to the game itself (graphics and unreachable environment) should be designed in such a way that it does not steal the players focus. Even though this is a design issue it is also something to think about when designing a PCG algorithm. Paths, areas and enemies those are not reachable for the player should not be generated at all. If they are, it may distract the player from whatever he tries to concentrate on.</td>
</tr>
<tr>
<td>Players should not be able to make errors that are detrimental to the game and should be supported in recovering from errors</td>
<td>An important thing to remember is that a PCG algorithm must ensure that the player cannot do anything that ruins the game. In the case of a map generator, it is vital to ensure that all levels are passable and that the player cannot reach places where the character gets stuck. Note that this topic is not only specific to map or enemy generation; it also requires them both to work together. If both the map and enemies are generated and they both work individually, they can still cause impossible levels if they run together.</td>
</tr>
<tr>
<td>Players should feel a sense of control and impact onto the game world (like their actions matter and they are shaping the game world)</td>
<td>This criterion is much about what the player is able to do in the game and less about the generated content. However, even if it is not necessary, it can be a good design choice to let the PCG algorithm to be dependent on how the gamer plays the game. One example of this could be that a player can block a path and thereby preventing enemies from being spawned, or on the contrary, throwing baits to bring forth more enemies. When it comes to map generation, one could design the next area/level dependent on choices made by the player in the current area/level. These dependencies must of course be noticed by the player if he should feel that his actions matters. It could be as simple as that a door broken by the player continues to be broken in the next level.</td>
</tr>
<tr>
<td>Players should feel a sense of control over the actions that they take and the strategies that they use and that they are free to play the game the way that they want (not simply discovering actions and strategies planned by the game developers)</td>
<td>PCG algorithms should generate many optional solutions and features. At its very simplest form, this criterion was considered in Shattered Planet [3]. What they did, was to create several paths that lead to the same places. The reason is simply to give the player more options to choose among even if it does not really matters which action that gets chosen. A more evolved solution suggestion to this topic is to let enemy spawning be dependent on the player’s strategy. A concept that is used in many games is that less enemies spawns if the player hides the corpse after having killed an enemy.</td>
</tr>
</tbody>
</table>

Table 2.1: Design decisions
“Rhythm groups are small, non-overlapping sections of a level that encapsulate a sense of rhythm for the player. For example, a rhythm group could be three short hops with minimal movement in between, or a repeated section of long runs with small jumps in between. To capture this sense of rhythm, rhythm and geometry are separated through a two-tiered generation system. All rhythm groups built off of the same rhythm may have a different geometric representation, but ultimately “feel” the same to the player. Rhythms maintain a length, density, beat pattern, and the ability to reflect or repeat that pattern.”

Rhythms can be an effective way to establish both variety and progression [13]. By alternating different rhythm groups and increasingly introduce new ones, it is possible to regulate progression and achieve a more varied game. Generally, map and enemy PCG algorithms are both obvious choices to introduce rhythms into, since both of them have a direct impact on how the player must play the game. The PCG algorithm can then include rhythms by alternating the distance and/or direction to obstacles and enemies.

2.3 Motion Games

There are two very important things to decide before starting to develop a game. The first thing is to decide the game idea and the game genre. The other thing is to decide how the player should interact with the game. These two decisions are very dependent on each other. Different game genres call for different controllers, and vice versa. Klochek & MacKenzie proved that keyboard and mouse were more suitable for first person shooter (FPS) games than a gamepad [16]. There has also been many attempts to change the standard controls for real time strategy (RTS) games [17]. None of those has resulted in success.

A relatively new way to control a game is through motion. These games are called motion games and work by letting the player move his own body to control the game. A bunch of sensor technologies have been developed to be able to do that. Some examples are Nintendo Wii, Playstation Eyetoy and Microsoft’s Kinect sensor [20]. Other devices as regular webcams and smart phones can also be used. All of these equipments are suitable for different game types. For example, Kinect and webcams support full body interaction, while Wii can get much better precision, thanks to the small size of the Wii remote aim point in compared to human limbs [20]. However, this thesis is delimited to only consider PCG for technologies that do not need any external tools to control the game. That means that Nintendo Wii gets excluded because it uses a Wii remote control.

Motion games have been proven to have many health benefits compared to regular games. When playing a regular game, the player most often sits still the whole time he plays the game. Physical inactivity like that has been shown to be associated with many diseases like diabetes and cardiovascular disease [18]. Gerling et al. also concluded that motion games can have a positive impact of the player’s mood [19]. However, too much playing can still be harmful and cause injuries like musculoskeletal disorder [27]. Even if a player does not play too much, it is still a good idea to include natural breaks in the game so the player can rest sometimes. It can be really exhausting to hold up your arms for a long amount of time.

2.3.1 Entertainment in Motion Games

Motion games have the potential to become ridiculously popular. Studies has shown that motion games increase the player’s engagement to the game [21]. According to the GameFlow model we saw earlier, we could see that players get more entertained if they pay attention to the game (criterion 7.4). The model also teaches us the importance of adjusting the difficulty to the player’s skills, especially early in the game (criterion 2.1). This may be extra important to think about, since some people are not as accustomed to controlling the game with motions as they are with gamepads and keyboard/mouse.
Motion games can also very easily adapt rhythms into the games. More parts of the body can then be involved in the rhythms than just a few fingers as is the case when using a gamepad. According to Isbister & Mueller rhythms are a very good thing to include in motion games [20]. Lately, there have also been many successful games in the rhythm genre. An obvious example of that is the Guitar Hero series [4]. Another very successful rhythm game that also is a fulfilled motion game is Just Dance [5]. These two are examples of true rhythm games where the rhythm is the main concept of the game. Even though all motion games do not build upon that, it still can be a good idea to embed rhythms into them.
3 Method

The approach that has been used in this thesis in order to come up with and evaluate PCG solutions for web-based motion games was to perform a case study. This thesis was dependent on constructing PCG algorithms that work very well in the meaning of variation and progression. The algorithms had to be as effective as possible to make them feasible to use in all lightweight web-based games like the one in the case study. Because of those reasons, it would have been insufficient to only test them at the end of the thesis. Therefore tests have been run and evaluated continuously during the whole thesis. An agile software development method was used to assure that the game would be tested and altered throughout the whole thesis. This chapter brings up more information about the case study and the iterations.

3.1 Case Study

A case study is a study that focuses on a specific case in order to find out how theory works in practice [22]. The purpose can either be exploratory, descriptive, explanatory or improving, or a combination of them [22]. In this thesis, the purpose was to explore what happens when applying the design guidelines from chapter 2 on a motion game. For obvious reasons it is impossible to cover all kind of motion games in this report, and therefore, this case study used the bug game that was described in the Background section in chapter 1. The derived results are therefore not general for all games that classify as web-based motion games. The results should therefore be seen as an illustration of how the guidelines may be applied.

3.2 Iterations

This thesis was divided into three iterations. Each iteration consisted of two phases: Implementation and Testing. This section describes both of them.
3.2. Iterations

3.2.1 Implementation

The implementation phase was the phase where all the development was done. That also involved choosing what would be done in the iteration. To do that, we have to know what must be done in the thesis. That can pretty much be summarized into three main tasks to solve:

- Map generator - PCG algorithm to generate the map
- Creep generator - PCG algorithm to generate creeps and tuning game parameters
- Game design - Improving the game concept and general design. This includes tasks like redesigning the map representation, adding more bug types and inventing game rules

These main tasks were divided into smaller tasks that were conquered during the three iterations. All tasks where rated in the beginning of each iteration. This was done to ensure that important tasks were solved early. The rating was done by grading three properties of the tasks: Importance, Status and Goal. Those properties are explained in Table 3.1. As can be understood from the table, tasks can either be a completely new task or an improvement of a task done in an earlier iteration. After the tasks had been rated, it was time to start working on them, one at a time, until the iteration was over.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Grading</th>
</tr>
</thead>
</table>
| Importance | How important the task is to fulfill in this iteration | 1 = vital  
| Status | How well the existing implementation works | 2 = important  
| Goal | The desired Status after the iteration is finished | 3 = desired  
| | | 0 = not implemented  
| | | 1 = works poorly  
| | | 2 = works  
| | | 3 = works good |

Table 3.1: Task properties

3.2.2 Testing

Three user tests were arranged in this thesis; one per iteration in order to evaluate the iteration and plan what to do in the next one. The tests took place in public areas. The author of this report was in place and observed the playing and asked questions. Afterwards the players were offered to fill in a survey. The first iteration used a standard SUS (System Usability Scale) survey. SUS is a well-known survey to measure usability. It is a Likert scale (0-4) and consists of ten questions, where even questions are agreement questions and the odd ones are disagreement question. The result of the survey is calculated by adding together each question’s score and multiply the sum with 2.5 to receive a score between 0 and 100. The disagreement questions are inverted so a 4 counts as 0 in the calculations. A score above 68 is considered to be above average. The answer for each individual question is considered unimportant. [23] [24]
The last two iterations used a modified PSSUQ survey instead. PSSUQ is just like SUS a high reliability survey that is used to test usability. It uses 19 questions and a seven points Likert scale. The survey used in this thesis is not the original survey, but instead a customized version that is translated into Swedish and only has twelve questions. The translation and removal of questions were done by Mathias Nordvall at Linköping University in order to suit a game that he created. The survey can be seen in the Appendix. The result of the PSSUQ that will be used in this thesis is the overall score, which is the average value of all questions.

The results of the user tests are evaluated in chapter 4. However, those tests are not the only tests that have been run. A few people (e.g. my supervisors) have throughout the thesis tested the game and have come up with both criticism and ideas. The game has also run continuously 24 hours a day on a public machine on one of Linköping University’s campus. This has been used to detect long term bugs and crashes and to find out if the game is easy enough to play.
This chapter describes how the bug game in the case study has been implemented. The guidelines from the Theory chapter are used to motive the design decisions. Since the case study was divided into three iterations, this chapter will describe one and each of them individually.

4.1 Iteration 1

The main purpose of the first iteration was to redesign the game implementation to make it possible to add PCG algorithms for the map and the creep flow. In addition to that, some new bug types and basic PCG algorithms for both the map and the creep flow were planned to be implemented. At the end of the iteration all of these things were completed. Table 4.1 gives an overview of the most important tasks that were done in this iteration.

<table>
<thead>
<tr>
<th>Task</th>
<th>Importance</th>
<th>Status</th>
<th>Goal</th>
<th>New Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change map representation</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Add predesigned maps</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Implement flying bugs</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Implement twinkle bugs</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Build a map PCG</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Build a creep PCG</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Evaluation iteration 1

4.1.1 Implementation

The most crucial task in this iteration was to change the map representation. This had to be done since the already existing game only included one predesigned map, which had been created with the map editor Tiled. The map consisted of a large image (1248 x 960 px) and a JSON file describing where the bug and drop paths went. These files were loaded into the cache before the game started.
In the new solution, the map is represented by a grid layout of tiles. Each tile can hold one or more parts of a path to make it possible to include crossovers. In this case a path part consists of a type (string associated with an image) and a direction to the next tile in the path. The size of the grid system is calculated dependent on the size of the images that are loaded into the game. The current image size is 96 x 96 px which creates a 13 x 10 grid system. Two predesigned maps are created for tutorial purposes. A map equal to the previous existing map is also designed to be used as an alternative to the generated maps.

The map PCG developed runs online and generates a new map in between every two consecutive levels. The current algorithm is stochastic and uses a typical constructive process. It can easily be described by two functions: `generateMap` and `findPath`. `generateMap` is the main function which decides how many bugs and water drop paths that should be generated based on the current level. Afterwards, it randomly distributes start positions for the paths over a set of possible start tiles along the top side and the edges of the map. Subsequently, `findPath` performs a modified depth-first search (DFS) to find paths from the chosen start tiles to the tree. An Euclidean distance heuristic is used to find the goal, meaning that the paths go straight to the tree, if possible (see Figure 4.1).

Figure 4.1: A generated map. One can see how the Euclidean distance heuristic makes the paths go straight to the tree if there are no other paths in the way. Attentive people may notice that the first tiles in the paths does not necessary choose the shortest path. The reason for that is that start tiles are assigned a direction before the search begins.

If there are many paths that should be generated, it may happen that it is not possible to reach the tree from a given start tile. Therefore, `generateMap` continues to call `findPath` with new start tiles until it finds a path or until all start tiles have been tried out. In the latter case, the PCG disregards the planned path and settles with the paths that already have been created. To ensure that there exists at least one bug and one water drop path `generateMap` alternates between constructing water and bug paths.

Unlike the map PCG, the creep PCG is active during gameplay and not only in between levels. What it does is to spawn bugs and drops repeatedly and control which bugs that should be squishable. The delay between two subsequently spawns decreases throughout the level to increase the difficulty as the time goes by. Furthermore, the delay receives new start values for each level, to increase the difficulty as the levels goes by. A simple modification was also done to the choice of which bug to squish next. Instead of using a regular queue (according to spawning time) a list is used which alternates between working as a queue and stack for each bug taken. The reason for that is to make the game harder and to make the
player move more (criterion 2.1)(section 2.3.1). The stack prevents the player from always squishing the bug that has been on the field longest time and thereby tends to be close to the tree.

At last, two new bug types were implemented in this iteration, in order to make it possible to vary and progress the game over time (criterion 1.1). The first bug is a twinkling bug. This bug slowly twinkles between being fully visible and semi-transparent. When the bug is semi-transparent it becomes invulnerable, forcing the player to wait until it gets visible again. However, the bug can always be squished when it comes near the tree, giving the player a last chance to squish it, so he does not feel like he was matchless (criterion 4.4). The second new implemented bug is a flying bug that flies away once or twice when the player tries to squish it. Stopover positions are marked on the map so the player should be able to foresee where the bug will fly and initiate his next move in advance. Flight bugs can be a great tool to increase the player’s motions, by simply just placing stopovers on different sides of the map. This can in turn increase the engagement to the game (section 2.3.1).

4.1.2 Testing

The user test for the first iteration took place in a public place at daytime. The people who came to test the game were mostly children and elderly people. Six people filled in the SUS survey, which resulted in a score of 81 points. A few things were also possible to discover through the observation and the discussions with the participants. The first thing that was noticed was that the game rules were not that clear for some people. They often tried to take the bug closest to the tree instead of the bug that really was the current target. This will hopefully be solved by introducing an in-game tutorial, which probably will work, as they understood the rules after being told about them. Although, it was still hard for them to understand that some of the bugs were special and could be squished at any time. This will partially be solved by the in-game tutorial, but as an additional measure, all special bugs will get a more characteristic appearance.

4.2 Iteration 2

The second iteration’s goal was to improve the map PCG and thereafter the creep PCG that was built in the first iteration. Both of these were taken care of, even though none of them works perfectly. After iteration two was finished, it was primarily the map PCG that did not reach the desired level, which can be seen in Table 4.2.

<table>
<thead>
<tr>
<th>Task</th>
<th>Importance</th>
<th>Status</th>
<th>Goal</th>
<th>New Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improve the map PCG</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Improve the creep PCG</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Evaluation iteration 2

4.2.1 Implementation

After the first iteration, the map PCG used a DFS algorithm with the Euclidean distance as a heuristic to find the tree from start tiles. The maps that could be generated with that PCG were rather boring and not that different from each other. The first improvements that fixed that were to update the heuristic and allow it to produce non-optimal solutions. Instead of always favoring the best neighbor the heuristic now choose to move randomly or straight forward now and then. This makes it harder for the DFS to find the goal which leads to more various and winding paths. A maximum path length was also added to prevent the paths from becoming too long.
Another improvement was to rebuild the constructive map PCG to an SBPCG (an EA with a single solution instead of a set of candidate solutions). Instead of just producing as many paths as needed, this algorithm continues to generate paths until the algorithm cannot find more paths to the tree from start tiles that are available. The algorithm then uses all generated paths and tries to produce a great map, with as many paths as needed, by picking out which paths to use and what type they should be (bug or water path). The algorithm discards all paths and tries again if it generates fewer paths than needed.

The first step in the algorithm is to choose an initial solution. The solution consists of the first $n + m$ generated paths, where $n$ is the number of required bug paths and $m$ the number of required water paths. In the same manner, the first $n$ generated paths are set to be bug paths and the remaining paths are marked as water paths. All paths that are not in the current solution are placed in the candidate path list (not to be confused with candidate solutions). After the initial set up, the algorithm starts to iterate and randomly picks one of two mutations in each iteration — either to swap a path in the current solution for a candidate path or to swap the type between two paths in the current solution. An important thing to note with the first mutation is that the path from the candidate solution gets the same path type as the replaced path from the current solution previously had. That is to ensure that the current solution always has $n$ bug paths and $m$ water paths. Figure 4.2 illustrates how the algorithm works and Figure 4.3 shows how the map may look when the algorithm has terminated.

Figure 4.2: An initial solution that the Search-based algorithm may start with before it starts to mutate the solution. The red arrow represents the first kind of mutation; a swap between a path in the current solution and a path in the candidate path list. The blue arrow represents a path type mutation between two paths in the current solution.
A utility value is also calculated in each iteration. The utility function that is used is

\[
20 \times \text{meanCrossovers} + 2 \times \text{meanArea} + (1 - \text{meanDensity}) + (1 - \text{meanDistance})
\]

The constants in the numerator have been tweaked manually until all parameters had a fair influence on the utility and maps in general looked good. The constant in the denominator is used to normalize the utility to a value between zero and one. The parameters are contemplated to give the map qualities that are considered essential for a great map. We will now go through one and each of them to explain how they work and to describe their purposes.

**meanCrossovers**

This parameter is not exactly what it sounds like. It does count the average number of times each path crosses another path in the solution path set. But, there are two things to notice. The first is that it only counts the crossovers to paths of another type. This is because the primal reason of this parameter is to ensure that the water paths do not get isolated from the bug paths. If they get isolated, players will not find the game challenging (criterion 2.1). The other thing to notice is that the value it produces is the mean crossovers per path length unit and not simply means crossovers. The reason is to make this parameter independent of the path lengths, since the intention is to ensure that the paths have many crossovers relative to their own length.
4.2. Iteration 2

**meanArea**
This parameter is the average of all paths’ areas. The area is counted as the minimum rectangular square that a path can fit in. A greater area is favorable since it means that the path goes all over the map instead of straight to the tree. This partly results in more crossovers, but it also requires the player to move more that in turn increases the enjoyment (section 2.3.1). The area is normalized and a value of one corresponds to the area of the whole map.

**meanDensity**
The meanDensity is the density of the actual path parts in all paths’ area boxes. This value should not be too high, because we do not want paths that are lumped into a minor area. Paths with massive density cause the player to move less and do not normally contain as many crossovers as low-density paths.

**meanDistance**
This is the mean distance between all of the path’s center points. A center point refers to the center point of a path’s area box. The value is normalized and one corresponds to the diagonal of the map. The meanDistance should be small. The reason is that great distances tend to result in paths that are isolated from each other, which can result in way too easy maps.

The utility value is not only used to measure which solution that is the best, but also to rate individual paths and their type. If it repeatedly turns out to be favorable to include a specific path in the solution, it will get locked, meaning that it cannot get affected by mutations anymore. The same applies in the other direction (good to exclude it) and also to lock path types to a specific value (bug or water). The algorithm terminates when one of three end condition occurs: after 300 iterations, after 100 consecutive iterations without improvement, or when no more mutations are possible to do due to locked paths and path types. The numbers 100 and 300 are arbitrary chosen to prevent the algorithm from taking too long time but still let it produce acceptable maps. An example of a generated map with eight paths can be seen in Figure 4.4.

![Figure 4.4: A generated map with eight paths. The paths do not go straight to the tree anymore, thanks to the new SBPCG algorithm. It also produces many more crossovers](image)
However, no matter how great the map gets, the game will not be fun without challenge (criterion 2.1). This is exactly what the game lacked at the end of the first iteration. The game became trivial for advanced players, since they were able to squish all bugs directly when they spawned. This made it completely meaningless to have a challenge list that forced the player to squish the bugs in a specific order. This problem was mitigated by a few modifications.

The first modification was to simply spawn an extra bug whenever the map got empty. The second one was to prevent two consecutive bugs to spawn on the same path, which forces the player to move. The last modification was to introduce flight bugs in the challenge list. Earlier on, they were special bugs that could be taken anytime. The advantage of including them in the challenge list is that it gives short natural breaks in the game while the player waits for bugs to finish their flight trips (section 2.3). An adjustment was also made to the algorithm that spawns bugs. It now prevents flight bugs from spawning when the total flight time for all flight bugs on the map exceeds the time it will take for the new bug to reach the tree. This rule is necessary to avoid bug flows that are impossible to pass.

Back in iteration 1, the challenge queue was changed to a combined queue and stack. This change was undone in this iteration. The reason for that was to introduce patterns. In this case, patterns refer to a short series of bugs (e.g. green, red, green or pink, green, orange, red). The main reason to do this is to open up for variation and progression as described under Rhythm Groups in chapter 2, and to stimulate the player’s brain (criterion 1.5). Two different pattern solutions were tried; a constant pattern solution and a pattern group solution. The first one encountered some problems, probably mainly because of the graphical representation. This is explained more thoroughly in the Testing section. For now, we content ourselves with a brief explanation of them both.

![Figure 4.5: The bug game when it uses a constant pattern. The bug previewer to the right highlights the bug that should be squished next. When the last bug gets squished it starts over from the top again. The number of bugs in the pattern depends on the level](image)

The constant pattern solution (Figure 4.5) only uses one pattern per level. The bugs must be taken in the specific order and the pattern is constant in a level. This may feel a bit repetitive, but has one big advantage — it allows the player to memorize quite long patterns and lower the need of looking at the bug previewer, i.e. the box that shows in which order the bugs can be squished. On the contrary, the pattern group solution (Figure 4.6) only allows short patterns and feels less monotonous. This is because the pattern group solution defines a set
4.3. Iteration 3

of patterns for each level. When all bugs in a pattern have been squished, it will be replaced by another pattern from the set. In this way, it is still possible for the player to recognize patterns but not to the same extent.

Figure 4.6: The bug game when it uses a group of patterns. The bug previewer at the top shows the current pattern and which bug to squish next. When all bugs in the pattern are squished, the bug previewer shows a new pattern.

4.2.2 Testing

The second user test was arranged at an exhibition at a campus at Linköping University. The participants were exclusively high school students. This test was conducted as an A/B test between the constant pattern solution and pattern group solution described in the previous section. As mentioned earlier, the constant pattern was the most difficult one to play. People who played that version received significantly lower scores. These players also rated the game worse on the survey; with an average score of 4.9 compared to 6.0 (a total of 10 people participated). As a result of that, the constant pattern solution was not considered anymore.

4.3 Iteration 3

At the time iteration 3 started, the game had both a functional map PCG and creep PCG. The overall game variation and progression were very restricted though. Therefore, iteration 3 focused on solving this problem by tweaking the game design.

4.3.1 Implementation

A big problem with the game after iteration 2 was the lack of game experience the game could deliver. The only goal was to survive as long as possible and the only profit given was a score, each time starting over from the beginning. The first thing done in iteration 3 was therefore to change the basic game model to a level based game. That opens up for a never-ending game with unlimited possibilities for variation and progression. The levels are divided into groups called chapters. Each chapter consists of ten levels. The main reason for dividing levels into chapters is to be able to naturally introduce new features to the game in between each chapter, letting the player know that new features will come as soon as a new chapter has been unlocked (criterion 2.4).
On top of that, the level based system makes it not only easy to run the map PCG offline, but also advantageous. By generating the maps on beforehand it is possible to cleverly select levels that seem to produce an adequate progression. This is done in the game, by introducing more complex maps further into the game. However, since every chapter introduces new content to the game and thereby makes the game more exciting and possibly harder, each chapter rolls back the map difficulty a bit (criterion 2.1, 2.3). This means that the first map in a chapter always is less complex than the last map in the preceding chapter.

The creep PCG follows the same chapter-sprint principle as the map structure; it increase the difficulty (creep speed and quantity) per level, but rewinds it a bit in between every chapter. The difficulty does not increase forever. At some point, it stops to grow. This is due to the never-ending game structure. If the difficulty always would increase, all players would eventually get stuck on a level they cannot pass, and then stop playing (criterion 2.1). At this point, it is important to continue to introduce new types of challenges into the game. Challenges that not necessarily needs to be harder, but that uphold the game variation (criterion 1.1, 2.4).

Variation is achieved in several ways. First of all, each chapter brings new content to the game. This content can either be new creep types or game features like a new game mode (more about those later). Creeps may exist in a single chapter only, or continue to exist in future chapters. Creeps that continue to exist are not always present. Instead, a subset of available bugs is randomly picked out for each level, making each level more unique and less predictable.

Another variation that is included is seasons, which is a graphical variation for chapters. Even though seasons are pure visual modifications they do have an effect on the gameplay. The most obvious reason for that is that they vary the field of sight for the player. Figure 4.7 shows two different seasons. One can see that it is easier to locate the bugs and to see the player’s own interaction shadow in the spring season, because of all the snow piles and the background’s contrast to the shadow color on the winter level.

Figure 4.7: Two different levels from two different chapters. The first one takes place in the spring and the second one in the winter.
As mentioned earlier, the previous objective was to survive as long as possible and in that way, save as many trees as possible. This goal is dismissed in the new level based game model. Instead, the player saves one to three trees, depending on his score when he completes a level (criterion 3.6). Higher score is received by squishing bugs earlier, in order to motivate the player to move more (section 2.3.1). The current score and how many trees it corresponds to are shown in the game (criterion 6.3). As stated in the Theory chapter, players should be stimulated from different sources and also be offered new experiences throughout the game (criterion 1.1). Therefore, a few game modes are included in the game (Table 4.3). These game modes are also supposed to give the player a new type of challenge that in some way is harder (criterion 2.4). The reason is to ensure that the game gets harder periodically (difficulty spikes) so the player can feel relieved and rest for a while when he passes one of the game modes (criterion 2.3) (section 2.3).

The game modes are unlocked one at a time when reaching new chapters and are assigned to a specific level within the chapters (e.g. the game mode no mercy is always at the last level in a chapter). A menu to choose which level to play is also implemented, in order to let the player go back and play levels that already have been completed. In this way, he can improve old high scores and choose to play game modes that he likes and finds challenging enough (criterion 2.1, 2.2). The map PCG is also used to generate the level menu. It generates a map from the left to the right side of the screen with level dots scattered along the way. The chapter menu however, consists of boxes arranged in a grid system to make a clear distinction between chapter and levels.

<table>
<thead>
<tr>
<th>Game Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Play a level for 60 seconds. Do not let any bug reach the tree. Get higher score by squishing the bugs earlier. 25% of the maximum score is needed to pass the level. To earn two or three trees 50% and 75% is needed, respectively. The amount of water drops that is allowed to be squished before losing the game varies between two and four. The speed of the creeps increases for each level but drops a bit for each new chapter. This is to keep up with the player’s skill progression and to ensure that the difficulty is not constantly too intense (criterion 2.3). A too intensive game would probably exhaust the player (section 2.3).</td>
</tr>
<tr>
<td>Points challenge</td>
<td>Same as the normal game mode, except that the score needed to pass the level is higher. The new objectives are now 45%, 65% and 80%.</td>
</tr>
<tr>
<td>Survival challenge</td>
<td>This mode may be stressful. The difference from the normal game mode is that the bugs increase their speed throughout the level. They start at normal speed and right before the level has been passed, the speed of the bugs has increased by 30%.</td>
</tr>
<tr>
<td>No mercy</td>
<td>No mercy is an unforgivable hard game mode. The game is over as soon as the player squishes his first water drop. This game mode is placed to be the last level in its chapter. The point of that is to let the player feel relieved when passing it, since the next chapter will start out easier and there is also always a reward coming in the shape of a new game feature, such as a new bug type (criterion 3.6).</td>
</tr>
</tbody>
</table>

Table 4.3: Game modes
Apart from the seasons and the game modes, there is one more major source to variation in the game, namely themes. Themes are both a graphical and a mechanical modification. The player should both be able to try out and buy new themes in order to play the game as he wants (criterion 1.2, 2.2). The only additional theme that exists for the moment is a desert theme, where the tree is replaced with a cactus, the background environment with a desert, the paths with spider web and finally the bugs with spiders. The mechanical change for this specific theme is to increase the average speed of the bugs, since spiders are naturally fast. People who likes to play more speed intense games, or just do not know what the word “arachnophobia” means, can now do that by swapping to this theme.

4.3.2 Testing

In the third and last test occasion the participants were high school students. The intention was to observe how well-adjusted the early game difficulty was. The conclusion was that it seemed to work alright. However, one problem could be noticed. It was not completely understandable why the bugs had to be squished in a certain order. Therefore, that feature is put on hold until a better solution comes up. Eleven people filled in the survey which resulted in an average of 5.3.
As in every project, everything do not work out exactly as planned. Despite that, this thesis turned out pretty well. This chapter discusses various problems with the thesis work and what could have been done better, both regarding the received results and the arrangement of the thesis. At the end, there is a short section about how this thesis affects the players.

5.1 Results

This section discusses the results that have been attained in this thesis, both regarding the results of the implementation and the performed tests.

5.1.1 Implementation

Some parts of the implementation took a little longer time than expected, for example to try out and test new ideas and to tune game parameters to make the game fun. Also some optimization related things took longer time. Neither object pooling nor the delta time in between two subsequently screen renderings seemed to work properly in the JavaScript framework (Phaser) that the game is built in. That was solved by implementing an own object pooling system and to keep the old FPS-dependent movement solution, that moves the creeps more slowly at low FPS-rates. The latter problem was also tried to be solved in other ways, but all of them were considered to consume to much CPU power. Apart from those problems the case study turned out pretty well. There are some things worth to mention though, which will be explained in the following sections.

5.1.1.1 Game Design

The Theory chapter brings up many designing guidelines for both regular and motion games. All guidelines have not been used in this thesis’ case study. That does not mean that the guidelines do not work well. It rather tells us that all guidelines are not suitable to use in every game. This means that the case study only serves an example of how to apply the guidelines in a real game. The guidelines that have not been used therefore remain untested. One of them is to include rhythm groups. There were some thoughts about controlling the creep flow to include rhythms, but it turned out to be a rather complex thing to do in this
5.1. Results

To do that, there would have to be an easier way to control when and where the player should be able to squish the bugs. Although, even if it did not work out very well in this case study, it probably still is a very great thing to include in motion games that can adapt it more easily. As described in the Theory, games like Guitar Hero and Just Dance have been very successful. Just Dance 3 for example, sold over 7 million copies in the first three months [5].

A feature that suited this game very well was the rule that forced the players to squish the bugs in order. The preview box that was used to show the order was not that easy to understand though, according to the user tests. This made it necessary to deactivate that feature because it was completely vital to understand how that worked in order to play the game. To include unclear rules like that violates several criteria in the GameFlow model, in fact almost every criterion about the player’s skill level and about clear goals [12]. Another solution has been implemented to be able to keep the rule. That solution uses chains around the bugs to show which of them that can be squished. More chains on a bug means that more bugs have to be squished before that one. That solution needs better graphics, some animations and preferably a convenient reason to why the bugs have chains on themselves in the game story. The last part is important according to Sweetser & Wyeth. They clarify that the player should feel that the goal and the game rules are meaningful [12]. After that has been fixed, it probably will be entirely self-explanatory. Until then, it will continue to be deactivated.

A great thing with the squish-in-order rule is that it makes the game a lot more challenging and forces the players to move and think more by preventing them from squishing all nearby bugs and then moving their hands to another spot on the map and do the same thing there. Neither can they keep their hands near the tree and wait for the bugs to come there (this is not completely possible without that rule neither, since the player will not raise enough points to pass a level if he only does so). Sweetser & Wyeth who came up with the GameFlow model point out that a game has to be challenging to be fun [12]. Bianchi-Berthouze et al. also claim that more movement increases the engagement to the game [21]. Having that knowledge makes it quite easy to understand why the game feels a bit boring when the squish-in-order rule is deactivated.

The squish in order rule is not the only thing that is deactivated. There are also some diverse bug types that are not in use. Like a blinding bug that darkens the map, but still shows the creeps, that become highlighted and can be squished in any order (planned to be used when the bugs have to be squished in order). Many different creep types are necessary to include in the game to achieve variation, which also is stated by Sweetser & Wyeth [12]. To continuously add new creep types to the game, as a chapter feature or as a mechanical change in a new theme, is something that really keeps the game alive. New creeps may have qualities that are more or less hard to combat, and always provides a new game experience. As an example, one could implement a builder bug that constructs shortcuts in the map. That shortens the time it takes for bugs to reach the tree, which could make the game significantly harder, depending on how easy it is to stop it from doing that. It is also possible to make the game easier by introducing creeps that have positive effects on the game. Some creeps can both be positive and negative at the same time, like the blinding bugs, that hides the map but relaxes the game rules.

5.1.1.2 Procedural Content Generators

The Theory chapter brought up many approaches of how to construct a PCG. All of those have not been used in the case study. The creep PCG uses a purely constructive approach and is quite simple to understand. It is lightweight and has no problem to run online. The map PCG is a bit more complex but works well to run online, even if it does not do that in this game. It uses a constructive approach to generate each path, but has a search-based PCG on top of that to generate the whole map. Some other approaches were considered for the map PCG, but none of them felt to adapt well to this game. For example, generation as
content selection was not really an option, due to the small size of the map. Approaches like
generation with grammar and constraint satisfaction did not feel as suitable either. Grammar
would probably be more appropriate in a game where the content is constructed dependent
on the player's input. Three different factors have been noticed to be crucial to the choice of
the PCG approach: game genre, content type and whether the algorithm should run online
or offline. The first two are important because they pretty much define every requirement
and constraint on the PCG. The last one determines how advanced algorithms you can use.

Back in the first iteration, the map PCG was a simple constructive algorithm. The search-
based top level was added first in the second iteration. This was something that was com-
pletely necessary. Before it was added, there was not that many crossovers between paths
at all and most of the paths went relatively straight to the tree. A reason to its success is of
course that the SBPCG chooses the paths it wants to use among a set of several paths. But that
may not be the most influencing reason. One that probably is likewise important is that the
algorithm always produces paths until the map gets full. That leads to more winding paths,
since the last ones generated have to go detours to find the tree because of that all previously
generated paths blocks the way. This really shows how advantageously it can be to upgrade
to a SBPCG.

G. Smith explains that search-based PCGs may require much CPU time [11]. This has
not been a problem when constructing this map generator. It may be slow compared to other
solutions, but it runs fast enough even on slow machines. So it would still work to run online,
especially since it only has to run in between levels, where the player still has to wait a while
when looking at the score menu. However, as it is now, it runs offline. All maps are generated
on beforehand, which also is positive, since it makes it possible for a human to approve the
maps and sort them from easy to hard. According to G. Smith it is a quite regular thing to
do to let a human be a part of the algorithm, especially in the validation part [11]. It has also
time to run more iterations and may therefore generate better maps. Using the PCG offline
means that all maps have to be stored in a database. This is done by representing maps as
short strings that get parsed in runtime when a map is loaded. That of course, does not take
very long time.

The chosen parameters for the utility function seem to work well. Several other para-
eters have been tried out. An example is a parameter that tried to vary the path lengths and
another parameter that tried to force the map to include paths on both the left and right side
of the map. These did not work out well, since they tended to imply rather boring maps that
did not have that many crossovers. Maps like those can still be generated with the current
algorithm though, even if most of them are good. The reason to why the PCG cannot guar-
antee that a map is good is because it does not test it. There are a few constraints that force
the map to include crossovers and so on, but nothing that really ensures a great map. The
quality and especially the performance could be improved with a heuristic for the SBPCG
part of the PCG. As it is now it is purely stochastic without any guidance. A heuristic could
make it converge faster and also help it to find better solutions faster. Although, that also
implies a risk to prune the most “optimal” solution, depending on how the heuristic function
is designed.

5.1.2 Testing

The tests have been very useful, both all the feedback that were received from all the people
who continuously tested the game and from the people who participated in the user tests.
This section will discuss a little more about all three user tests.
5.1.2.1 Test 1
The first test occasion was not that successful. The survey used was the SUS survey. That
did not work out well. It was easy to see that the children that filled in the survey did not
understand the questions completely, wherefore the PSSUQ survey was used in the other
two tests. However, what could be seen is that they liked the game. A score of 81 is far above
average, which is counted as 68 according to J. Brook [24]. This test occasion was also the first
time where it was possible to see the problem with understanding the game rules. It was a
good thing to notice that already in the first iteration.

5.1.2.2 Test 2
In contrast to the first iteration’s user test, there were many more people around this time.
This may have made some of the people nervous which can have made them to pay less
attention to the game. They had to play in front of people that also saw how they played on
a TV screen. The players also got the rules explained for them in this test. Anyhow, one very
interesting thing to notice from the test is that the people who tried out the constant pattern
solution, were much more negative (according to the survey) to the game than those who
tried the pattern group solution. This is interesting because the game worked in the same
way with both solutions. They only played for a short while (3-5 minutes), so the constantly
repetitive pattern was probably not the cause of the disappointment. Instead, it is likely to be
the increased difficulty that angered the players. According to Sweetser & Wyeth’s model it
is important to adjust the difficulty to the players’ skill level, otherwise they may feel anxiety
[12]. It was also noticeable that the players probably were unfamiliar with playing motion
games, which explains why they found it so hard. This could be seen when a guy played
the game for a little longer while (around 15-20). He then managed to improve his score by
a factor of ten eventually. Back then, the score was proportional to the number of squished
bugs.

One reason to why the constant pattern solution was a bit harder than the pattern group
solution may have been due to the amount of information that the player had to concentrate
on. Sweetser & Wyeth taught us that a game should not generate too much content that can
steal the player’s focus [12]. Even if it is hard to say what exactly that caused the difficulty
increase, this may be a part of the problem. In this case it was easy to just remove the preview
box or to use the group pattern solution. In some cases that may be harder. The “disturbing”
content will often be the same content that creates variation in the game, which is very
necessary to include in the game according to other criteria in the GameFlow model [12].

5.1.2.3 Test 3
This user test did not give much feedback at all. The students who played the game were not
really interested in it because they were actually there to get information about the University
and its programs. That made them to pay more attention in asking questions about that rather
than the game. However, the survey result was worse than for the group pattern solution in
the second test occasion but better than the constant pattern solution. The results cannot be
compared to the first iteration’s survey, since it used the SUS survey instead of the PSSUQ
survey.

5.2 Method
The aim of this report was to investigate how to design PCG algorithms to acquire variation
and progression in a web-based motion game. To do that, a case study was carried out, to
confirm that the designing guidelines in the Theory chapter work in practice. The theory was
mostly based on references from trustworthy sources like ACM. Also the online references
comes from reliable sites and do not contain any information that is vital for the results of
5.2. Method

This thesis. The Theory chapter addressed three different areas: how to construct a PCG algorithm, how to make a game fun and how to design a motion game. The first and the last subject are without doubt relevant to the topic. The second subject, is not directly mentioned in the research question, but is of course something that should be considered. If this thesis were to be extended, it would probably be best to include more material for motion game design, since that probably is the most unexplored subject of those three.

The case study went mostly well. It can be redone by an experienced programmer with the information given in this report. If someone would do that, it would be a good idea to run more tests. A bunch of people have tested the bug game frequently throughout the whole project time. But since these people quickly get familiar with the game and the interaction they are not able to know how it is for a new person to play it. Therefore this thesis also included three test occasions. Despite that, much more tests have to be run. One type of test that really needs to be done is a long term test. This thesis did not last long enough to include that, but without a long term test, it is impossible to tell whether the progression in the game works or not. Variation can be felt in brief tests, but even that would need long term tests to be tested thoroughly.

The user tests in this thesis also included some surveys. They did result in positive response, but may be considered as unreliable, since only a few people participated and the survey used in the first test occasion did not seem to be well adapted for the attendees. The survey used in the other two tests seemed to be more appropriate. However, that survey, the PSSUQ, was not in its original form (section 3.2.2). Mathias Nordvall had made changes to that survey, which may have affected its effectiveness. But since all questions in the PSSUQ survey are supposed to be of equal worth, that probably did not affect the results too much. The observations from those test occasions did also seem to be in line with the survey results.

Fruhling & Lee evaluated the reliability and validity of the PSSUQ survey. They first stated that PSSUQ is reliable and that it is a good method to use when measuring usability. They did also show that it is adaptable and can be reliable even when it has been altered to suit a specific system in the same way as Mathias Nordvall did. [28]

The observations done during the test occasions were managed in different ways. Holzinger mentions three types of test methods for usability testing: thinking aloud, field observation and questionnaires. The first and the last test occasion used a field observation method, which works by observing how the players play the game without explaining anything for them. It only lasted for a short while though, which may have been too short. The second test occasion did not actually use any of the methods; instead the players got the rules explained for them beforehand. The reason for that was to see whether they liked the game or not with the different pattern solutions used in that test occasion. The other two test occasions were conducted to realize which parts of the game that the users found to be hard. The different purposes of the tests could be categorized as learnability and satisfaction tests. Holzinger mention five types of usability characteristics to test that always should be present in all projects. Learnability and satisfaction are two of them. [29]

Another thing done in this thesis was to use a task-pool with graded tasks to choose what should have been done in each iteration. This turned out to be an unnecessary thing to do and was therefore not done in the last iteration. It would have been a good idea in a project with several project members, but for a lonely guy it did not serve any purpose, especially since the basic game idea was revised a few times. The sudden changes also proved that the agile methodology served its purpose. It was really crucial for this thesis, which is not that strange, since the thesis had an exploratory focus.
5.3 The Work in a Wider Context

Playing games is something that often is seen as an unnecessary and bad behavior. Many people see it as a waste of time and also very unhealthy since many gamers sit still on their couch and play games for hours without any breaks. This thesis, among many other studies, is a step forward to get rid of that bad reputation and turn gaming into a positive thing. Motion games are hopefully an appropriate solution to protect gamers from all of those uninterrupted hours of sedentary gaming. It is not only a “less harmful” alternative, it can in fact be of great use as a rehabilitation tool and have both physical and mental health benefits [18] [19]. Studies like this one are therefore very meaningful, because of their possibility to help game developers to develop better motion games than those who exist today. In that way, motion games may be able to compete more with regular games, which could improve the health of a large part of the earth population. An important thing to notice though, is that excessive use can have a negative effect on the health too [27].
motion games have many health benefits that regular video games lack. This thesis was therefore done as an attempt to investigate how procedural content generation should be used in order to increase the variation and progression in web-based motion games. In that way, it may help game developers and researchers to invent better games and thereby increase the popularity of motion games.

The thesis was carried out in two steps. The first step was to learn how to construct a PCG and to gather information about game entertainment and motion games. The second step was to conduct a case study to see whether it was possible or not to apply the attained guidelines in a real web-based motion game. The case study demonstrated that it works well; PCG can be used to achieve both variation and progression in web-based motion games. PCG can be used just like in a regular game, but there is a bunch of designing guidelines that should be borne in mind when doing that. The most important ones are to encourage the player to move a lot and to let him rest once in a while. The game should also be easy in the beginning due to the unfamiliar way to interact with the game. This means that one should be careful to generate too much unnecessary content and to have complex game rules early in the game. On top of that, a motion game must be designed in accordance with the restrictions that the interaction entails. In motion games, you do not usually have any buttons to press and your limbs will be attached in the same way as they always have been unless you use any technique to filter body parts of interest, such as color or shape detection.

Which PCG approach that should be used is dependent on the type of the content that should be generated, if it should run online or offline, and also on the game genre. The game genre also determines which guidelines those are suitable to use in the game. Due to this reason, the guidelines in this thesis should be considered as recommendations rather than strict rules to follow. The guidelines that came to use in this case study seemed to work well according to the tests. However, the thesis did not include any long term tests and it is therefore not possible to tell if the progression was well-adjusted. This is therefore something that future studies have to examine. It is also important to do more studies for other game genres to produce even more designing guidelines.
Bibliography


### Appendix: Post-Study System Usability Questionnaire

<table>
<thead>
<tr>
<th>PSSUQ</th>
<th>Häller inte</th>
<th>Häller helt</th>
<th>Vet ej</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grupp:</td>
<td>alls med</td>
<td>med</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Jag är överlag nöjd med hur lätt det var att använda spelet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Det var enkelt att använda spelet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Jag kände mig säker på hur jag skulle spela spelet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Det var enkelt att lära sig använda spelet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Det var enkelt att åtgärda misstag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Det var lätt att hitta informationen jag behövde</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Uppgifterna gick att utföra effektivt med informationen i spelet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Informationen i spelet visades på ett tydligt sätt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Jag var nöjd med spelets gränssnitt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Jag gillade att använda spelets gränssnitt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Spelets innehåll var vad jag förväntade mig</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 2 3 4 5 6 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Jag är överlag nöjd med spelet</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Dennis Persson