Empirical evaluation of procedural level generators for 2D platform games

Robert Hoeft
Agnieszka Nieznańska

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Authors:
Robert Hoeft
E-mail: roho12@student.bth.se
Agnieszka Nieznańska
E-mail: agni12@student.bth.se

External advisor:
dr. Mariusz Szwoch
Gdańsk University of Technology

University advisor:
dr. Johan Hagelbäck
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
ABSTRACT

Context. Procedural content generation (PCG) refers to algorithmical creation of game content (e.g. levels, maps, characters). Since PCG generators are able to produce huge amounts of game content, it becomes impractical for humans to evaluate them manually. Thus it is desirable to automate the process of evaluation.

Objectives. This work presents an automatic method for evaluation of procedural level generators for 2D platform games. The method was used for comparative evaluation of four procedural level generators developed within the research community.

Methods. The evaluation method relies on simulation of the human player’s behaviour in a 2D platform game environment. It is made up of three components: (1) a 2D platform game Infinite Mario Bros with levels generated by the compared generators, (2) a human-like bot and (3) quantitative models of player experience. The bot plays the levels and collects the data which are input to the models. The generators are evaluated based on the values output by the models. A method based on the simple moving average (SMA) is suggested for testing if the number of performed simulations is sufficient.

Results. The bot played all 6000 evaluated levels in less than ten minutes. The method based on the SMA showed that the number of simulations was sufficiently large.

Conclusions. It has been shown that the automatic method is much more efficient than the traditional evaluation made by humans while being consistent with human assessments.

Keywords: procedural content generation, procedural level generation, player experience, human-like bots, platform games
# Contents

**ABSTRACT** ......................................................................................................................... 3  
Contents .................................................................................................................................. 4  
List of tables............................................................................................................................ 6  
List of figures............................................................................................................................. 7  
1 Introduction.......................................................................................................................... 9  
  1.1 Overview of procedural content generation................................................................. 9  
    1.1.1 PCG as a form of data compression......................................................................... 9  
    1.1.2 PCG as a human designer assistant........................................................................ 10  
    1.1.3 PCG as a human programmer assistant .................................................................. 10  
    1.1.4 PCG as a personalized game content creation for an individual player ............... 11  
1.2 Problem statement.......................................................................................................... 12  
1.3 Scope and motivation....................................................................................................... 12  
1.4 Aim and objectives.......................................................................................................... 13  
1.5 Related work................................................................................................................... 14  
1.6 Outline............................................................................................................................. 15  
2 Research methodology....................................................................................................... 16  
  2.1 Research questions.......................................................................................................... 16  
  2.2 Research design............................................................................................................. 17  
    2.2.1 Literature reviews and discussions with experts.................................................... 17  
    2.2.2 HPS method ........................................................................................................... 19  
  2.3 Validity threats................................................................................................................ 21  
3 HPS method - system design............................................................................................. 23  
  3.1 Simulation based evaluation - overview ....................................................................... 23  
  3.2 Level generators............................................................................................................. 24  
  3.3 Bot component............................................................................................................... 25  
  3.4 Quantitative models of player experience .................................................................... 28  
4 Implementation...................................................................................................................... 32  
  4.1 Platform game engine (Mario AI Benchmark).............................................................. 32  
  4.2 Procedural level generators............................................................................................ 33  
    4.2.1 Random level generator ........................................................................................ 34  
    4.2.2 Design pattern-based level generator .................................................................. 35  
    4.2.3 Feasible-Infeasible Two-Population genetic level generator................................. 36  
    4.2.4 Occupancy-Regulated Extension level generator .................................................. 38  
  4.3 AI human-like bot player ............................................................................................... 41  
5 Simulation.............................................................................................................................. 45  
  5.1 Simulation environment and parameters ....................................................................... 45  
  5.2 Simulation results............................................................................................................ 46  
  5.3 Analysis and discussion.................................................................................................. 47  
    5.3.1 Maxima and minima of moving averages - testing the adequacy of the samples.... 47  
    5.3.2 Averages and standard deviations - comparing the generators............................ 53  
    5.3.3 Histograms - analysing the expressive ranges......................................................... 56  
6 Conclusions and future work.............................................................................................. 62  
References................................................................................................................................. 64  
Appendix A............................................................................................................................... 66
List of tables

Table 2.1 Research methods used to answer research questions ................................................. 17
Table 4.1 Identified shortcomings and bugs of the VLS bot and how they were overcome ........ 42
Table D.1 Estimated values of Fun for levels from the first group ............................................. 89
Table D.2 Estimated values of Fun for levels from the second group ......................................... 89
Table D.3 Estimated values of Fun for levels from the third group ............................................ 89
List of figures

Figure 3.1 HPS method – system design ................................................................. 24
Figure 3.2 The Tanagra intelligent level design tool with level generator [5] .................... 25
Figure 3.3 Robin Baumgarten’s Mario bot and a visualisation of its A* search algorithm [14] .. 26
Figure 3.4 VLS bot and the look ahead positions grid [17]........................................ 27
Figure 3.5 VLS bot and the terrain field [17]......................................................... 28
Figure 3.6 VLS bot and the enemies’ field [17]....................................................... 28
Figure 4.1 An example level generated by Random generator split into three fragments ...... 34
Figure 4.2 An example level generated by Random generator split into three fragments ... 35
Figure 4.3 An example level generated by Pattern-based generator split into three fragments ... 36
Figure 4.4 An example level generated by Pattern-based generator split into three fragments ... 36
Figure 4.5 An example level generated by FI-2Pop generator split into three fragments ...... 38
Figure 4.6 An example level generated by FI-2Pop generator split into three fragments ...... 38
Figure 4.7 The idea of level extension by attaching level chunk in anchor position [22] .... 39
Figure 4.8 An example level generated by ORE split into three fragments .................... 40
Figure 4.9 An example level generated by ORE split into three fragments .................... 41
Figure 5.1 SMA of fun for the levels generated by Fi-2Pop, window size = 10 (left), 50 (right) 49
Figure 5.2 Maxima and minima of SMAs of fun for the levels generated by Fi-2Pop......... 49
Figure 5.3 SMA of fun for the levels generated by ORE, window size = 10 (left), 50 (right).... 50
Figure 5.4 Maxima and minima of SMAs of fun for the levels generated by ORE ............ 50
Figure 5.5 SMA of fun for the levels generated by Patterns, window size = 10 (left), 50 (right) 51
Figure 5.6 Maxima and minima of SMAs of fun for the levels generated by Patterns .......... 51
Figure 5.7 SMA of fun for the levels generated by Random, window size = 10 (left), 50 (right) 52
Figure 5.8 Maxima and minima of SMAs of fun for the levels generated by Random ....... 52
Figure 5.9 Comparison of the averages and standard deviations of the six affective states for all 
four generators ........................................................................................................... 55
Figure 5.10 Distribution of levels for fun (all generators) ............................................ 59
Figure 5.11 Distribution of levels for challenge (all generators) ..................................... 59
Figure 5.12 Distribution of levels for frustration (all generators) ................................. 60
Figure 5.13 Distribution of levels for predictability (all generators) ......................... 60
Figure 5.14 Distribution of levels for anxiety (all generators) ...................................... 61
Figure 5.15 Distribution of levels for boredom (all generators) .................................. 61

Figure A.1 Maxima and minima of SMAs of fun for the levels generated by Fi-2Pop ....... 66
Figure A.2 Maxima and minima of SMAs of fun for the levels generated by ORE .......... 66
Figure A.3 Maxima and minima of SMAs of fun for the levels generated by Patterns .... 67
Figure A.4 Maxima and minima of SMAs of fun for the levels generated by Random ...... 67
Figure A.5 Maxima and minima of SMAs of challenge for the levels generated by Fi-2Pop .... 68
Figure A.6 Maxima and minima of SMAs of challenge for the levels generated by ORE ...... 68
Figure A.7 Maxima and minima of SMAs of challenge for the levels generated by Patterns ... 69
Figure A.8 Maxima and minima of SMAs of challenge for the levels generated by Random .... 69
Figure A.9 Maxima and minima of SMAs of frustration for the levels generated by Fi-2Pop ... 70
Figure A.10 Maxima and minima of SMAs of frustration for the levels generated by ORE ... 70
Figure A.11 Maxima and minima of SMAs of frustration for the levels generated by Patterns .. 71
Figure A.12 Maxima and minima of SMAs of frustration for the levels generated by Random . 71
Figure A.13 Maxima and minima of SMAs of predictability for the levels generated by Fi-2Pop
........................................................................................................................................72
Figure A.14 Maxima and minima of SMAs of predictability for the levels generated by ORE.. 72
Figure A.15 Maxima and minima of SMAs of predictability for the levels generated by Patterns
........................................................................................................................................73
Figure A.16 Maxima and minima of SMAs of predictability for the levels generated by Random
........................................................................................................................................73
Figure A.17 Maxima and minima of SMAs of anxiety for the levels generated by Fi-2Pop......74
Figure A.18 Maxima and minima of SMAs of anxiety for the levels generated by ORE........74
Figure A.19 Maxima and minima of SMAs of anxiety for the levels generated by Patterns......75
Figure A.20 Maxima and minima of SMAs of anxiety for the levels generated by Random .....75
Figure A.21 Maxima and minima of SMAs of boredom for the levels generated by Fi-2Pop ....76
Figure A.22 Maxima and minima of SMAs of boredom for the levels generated by ORE .......76
Figure A.23 Maxima and minima of SMAs of boredom for the levels generated by Patterns ....77
Figure A.24 Maxima and minima of SMAs of boredom for the levels generated by Random....77
1 INTRODUCTION

This chapter provides a definition of the subject matter, an overview of various applications of procedural content generation, a problem statement, an exposition of the aim and scope of this thesis and a survey of the literature. The chapter concludes with an outline of the organization of this work.

1.1 Overview of procedural content generation

Game development companies desire to reduce the time and costs of game production. This is not an easy task since games are becoming more and more complex, especially in terms of game content. In order to sustain players’ interest in a game for a longer time it is necessary to release new characters, levels, quests and other types of game content on a regular basis. An important question is: How can we reduce manual work done by human designers and at the same time produce more game content?

Procedural content generation (PCG) can be a promising solution to the above-specified problem. According to [1] PCG refers to any method which creates game content algorithmically, with or without the involvement of a human designer. PCG is a technology that on the one hand can support or to some extent replace the manual work done by human designers, while on the other it enables game developers to create new game genres or make feasible some game design decisions. The type of generated content and possible applications of PCG techniques depend on the game genre and design requirements. The PCG researchers have identified several reasons why the PCG technology may be useful [1], [2], [3]. In the following sections we will clarify the most important PCG applications.

1.1.1 PCG as a form of data compression

Due to technical limitations or design challenges some game developers may not want to store all game content but instead generate it at runtime, when it is necessary. It is worth noting that PCG algorithms used for game content compression must be deterministic - each time the same PCG algorithm (with a single seed value) is run we should get exactly the same content output [2]. An example of game, which uses this type of PCG is Elite - a space trading game published

---

1 Game content refers to such game assets as levels, terrain, maps, vegetation, weapons, plot, stories, quests, dialogue, rulesets, characters, sound effects and the like [2], [7].
by Acornsoft in 1984. In this game we can explore the whole universe, which consists of galaxies and planets - thanks to algorithmically generated worlds the game’s size has been compressed to 22KB [3]. Since memory and hard drives are cheaper nowadays, this application of PCG becomes less important.

1.1.2 PCG as a human designer assistant

PCG can help human designers in creating game content at design-time [4], and thus speed up the process and provide designers with new original ideas they would never come up with. We all know from our experience that quite often it is easier to modify or improve something that already exists than to start creating from scratch. This also applies to the process of game content design. Most design programs do not assist much in the designing process - while creating a new project a designer is provided only with a toolbox and a blank starting page [4]. This can be changed for the better thanks to incorporating PCG into the game content design process [4]. Let us imagine an intelligent design program which provides the designer with, instead of a blank page, some automatically generated prototypes of game content of a given type (prototypes are generated based on input parameters the designer has specified). The designer can choose the best prototypes and then modify and improve them - this process of communication between human-designer and computer-designer can end at this point or be more interactive and continue through a number of iterations. The degree of control over generated content the designer has depends on a specific design program. Most existing PCG systems are not designer-friendly though. Quite often in order to change some input parameters or edit the output from the generator a person should know how to change the generator’s source code. Further research is required in this area of PCG, but a good example of an intelligent human designer assistant developed within the research community is Tanagra - *the first ever AI-assisted level design tool that supports a designer creating levels for 2D platforming games* [4].

1.1.3 PCG as a human programmer assistant

This case is quite similar to the one described in section above. It is worth noting that PCG systems are not only for human designers - human programmers also use them and want to achieve the same goals - making a lot of original and diverse game content in a short time. The difference between designers and programmers is that while the former do not need to know how to program or understand the source code of PCG generators, the latter work with source code on
a daily basis, including writing and modifying the source codes of PCG generators. This significant difference has the effect that the programmers’ requirements for PCG systems are different - PCG generators do not need to have a sophisticated graphical user interface (GUI) or other features and functionalities that can be particularly important from the non-technical designers perspective. In this case programmers create PCG systems mostly for themselves, they can interact with them by modifying their source codes, specifying the number, types and values of input parameters, and thanks to it game content can be generated without a need to hire a designer. This type of PCG is very promising, especially for small game development studios, and also is of strong interest among researchers. Most scientific papers describe different PCG generators (different, in terms of the type of generated content and used algorithms). An example of such generator is Launchpad - a rhythm-based level generator for 2D platform games [6], [4].

1.1.4 PCG as a personalized game content creation for an individual player

PCG can be used to adapt game content to diverse skills and playing styles of individual players and thus improve the replayability of a game. For instance, advanced players will get levels, maps or race tracks with higher difficulties than beginners, while players who prefer to explore the game will get something different than those who enjoy finishing the game with the fastest time - thanks to it they will not get bored with the game so quickly. While the player is playing a game data describing his playing style are collected and based on them new content is generated. So we can say that players have indirect control over content generators, but it depends on design decisions how much visible this process is to the players during play [4]. The process of content generation can be done either at runtime or between runtimes. In the first case (generation at runtime) a response of PCG system is very fast and allows for creating infinite adaptive games. Every player’s action has the influence on what is next generated and presented to the player. An example of a game with an infinite adaptive world that is generated at runtime is Endless Web [3], [4]. It is a 2D platformer and, what is particularly interesting, it is the first game that has the player directly interact with a content generator, i.e. the player is given the opportunity to build strategies around the generator [4]. It should be noted that this approach requires extremely fast PCG algorithms. The process of generation personalized content can also be done between runtimes - the generated content, e.g. a new level or map, is personalized to an individual player based on his actions in the past game session or sessions. In this approach PCG
algorithms do not need to be runtime-fast. Thanks to it we can use PCG techniques with longer execution times (e.g. genetic algorithms). A good example of generators developed within the research community that create personalized game content between game runtimes are those submitted to the Level Generation Track - a part of the Mario AI Championship organized each year since 2010 [1].

1.2 Problem statement

PCG generators are able to produce huge amounts of game content of various quality. Hence, a very important issue related to PCG is evaluation of the quality of procedurally generated content. Although the most desirable form of evaluation is evaluation made by humans, it can be impractical to carry out in the case of huge amounts of generated content. Thus, it is necessary to automate the process of evaluation in order to identify the high-quality content generated by a particular generator and also to evaluate and compare different PCG generators based on the quality of their outputs. The latter, in particular, is becoming very important because of a growth in scientific publication on PCG generators in recent years.

1.3 Scope and motivation

The thesis focuses on empirical evaluation of selected procedural level generators for 2D platforms games and the rest of this work is devoted to procedural level generation (PLG). The main assumption is that the PLG generators are compared with their default input parameters\(^2\). We chose levels as the type of game content to deal with because creating levels is a very time-consuming process compared to creating other types of game content. Moreover, levels have a great influence on how a game is experienced by players. A game with poorly designed levels will usually be considered boring, uninteresting, and thus doomed to failure.

The reason why we chose 2D platformers is that it is still a popular game genre. An example of 2D platform game which achieved a notable success is Super Mario Bros (SMB) developed by Nintendo in 1985. This game is considered to be a classic of the genre of 2D platform games.

\(^2\) The reason why we compare the generators with their default input parameters is that every modification of the input parameters vector affects the space of generated levels. And there can be quite a lot of different combinations of such vectors.
It has been a source of inspiration for the next platformers, including Infinite Mario Bros (IMB)\(^3\) - an open-source Java clone of Super Mario Bros developed by Markus Persson [2], [7]. The fundamental difference between Infinite Mario Bros and Super Mario Bros is that levels in IMB are procedurally generated while in SMB they were human-created [7]. Because of its availability and close similarity to the most influential 2D platform game, Infinite Mario Bros has gained popularity within the research community. Modified versions of Infinite Mario Bros have been used in a number of research works within procedural level generation and other related areas [1], [8]. The research conducted for the thesis was also based on this game (see Section 4.1).

### 1.4 Aim and objectives

The aim of this work is to evaluate and compare several procedural level generators for 2D platform games in terms of the quality of generated output, and also to develop an automatic method for such comparative evaluation.

The main objectives are:
1. to identify existing procedural level generators for 2D platform games and to select four of them,
2. to develop an automatic method for evaluation of procedural level generators for 2D platform games,
3. to apply this method for comparative evaluation of selected generators,
4. to evaluate and compare the selected generators based on the results obtained from the automatic method.

The main contribution of this work is a comparative evaluation of selected four PLG generators and an automatic method for such comparative evaluation. In addition, this work also suggests a method for testing whether the number of generated levels is sufficiently large for analysis of a particular PLG generator (the method is based on a simple moving average). The automatic evaluation method is applicable to PLG generators for 2D platform games, especially

---

\(^3\) The game and its source code are available at https://mojang.com/notch/mario/.
those that have been developed for the Mario AI Benchmark (which is based on the Infinite Mario Bros game).

### 1.5 Related work

Most papers related to PCG and PLG generators focus on such aspects as the system design and implementation and do not evaluate the quality of generated output [24]. Only few works are devoted to the evaluation of the quality of generated content and usually they also compare several different generators or evaluate only one generator but with different combinations of its input parameters. These works can be classified into two groups according to the type of evaluation of PLG generators: (1) human evaluation or (2) metrics-based evaluation.

Shaker et al. [1] used human evaluation to rank the six PLG generators submitted to the Level Generation Track organized within the 2010 Mario AI Championship. It was the first PLG competition organized within the research community. 15 human players played the generated levels and ranked them according to how fun they were to play (they had to fill in a two-alternative forced-choice questionnaire after playing a pair of levels). The best generator was considered that one which reached the highest score. The main advantage of this approach is that human evaluation seems to be the most desirable form of evaluation (games are developed for humans and they know best which levels are the most fun for them). However, by using this approach it is difficult to fairly evaluate and compare different PLG generators. The space of generated levels that can be evaluated by humans is very limited, and definitely not sufficiently large (a generator is represented by only several generated levels).

The other method - metrics-based evaluation - was first introduced in [23], and later adapted in [24]. Smith and Whitehead [23] introduced a framework for analyzing the expressive range of PLG generators. By determining different metrics that measure different properties of the generated levels, they could examine the space of levels that can be produced. They suggested a way to visualize the expressive range of the generator by means of 2D histograms. They applied their method for only one PLG generator to visualize and analyze how the space of generated levels changed for different combinations of the generator’s input parameters. Shaker et al. [24] adapted the framework from [23] by defining two new metrics and applied it for comparative evaluation of the expressive range of three different PLG generators. The main advantage of metrics-based approach is that we can quantitatively evaluate and compare different level
generators based on their vast space of generated content. Such comparison is fairer, because in this case a generator is represented by a large number of its levels (e.g. 1000 levels generated in [24] and 10,000 levels generated in [23]). The main disadvantage is that the defined metrics are not related to human player’s perspective.

It would be interesting to try to combine the advantages of these two approaches - generators represented by a vast space of generated levels and metrics related to the human player’s perspective.

1.6 Outline

The work is organised as follows. In Chapter 2 (Research methodology) we present and motivate the research questions and the research methods that were used to address those questions. Chapter 3 (HPS method - system design) describes the automatic evaluation method, which was designed as an answer to research question 1.2, and was used to answer research question 1. Chapter 4 (Implementation) provides some implementation details of the components of the automatic evaluation method. In Chapter 5 (Simulation) this automatic method is applied for comparative evaluation of four selected PLG generators. This chapter describes the process of simulation the human player behaviour in a 2D platform game environment, the obtained results and three kinds of analysis. Chapter 6 (Conclusions and future work) presents the conclusions of the work and make some suggestions concerning the future work. Certain portions of this work that logically belong to Chapter 5 are presented as appendices. Appendix A contains a complete set of graphs that was used in the first kind of data analysis. The graphs presented in Chapter 5 and Appendix A were obtained using two MATLAB programs - their complete source codes are given in Appendices B and C.
2 RESEARCH METHODOLOGY

The first part of this chapter presents the formulated research questions followed by the research methodology overview. The undertaken research methodology process is further discussed and motivated in Section 2.3.

2.1 Research questions

As the aim of this thesis (stated in Section 1.4) was to evaluate and compare selected procedural level generators for 2D platform games in terms of the quality of generated output, the following main research question was formulated:

RQ1. Which procedural level generator produces levels that provide the best player experience?\(^4\)

In order to answer the main research question two additional sub-questions had to be investigated in the first place:

- **RQ1.1.** Which selection criteria should be applied to choose procedural level generators for 2D platform games?

- **RQ1.2.** What components should an automatic evaluation method for procedural level generators consist of?

Both **RQ1.1** and **RQ1.2** were addressed by literature reviews and discussions with experts (see Section 2.2.1). The answer to **RQ1.1** helped us choose procedural level generators for comparative evaluation. The answer to **RQ1.2** was used in the design process of the automatic evaluation method. This method relies on a simulation of the human player's behaviour in a 2D platform game environment (see Section 2.2.2 and Section 3). The human player simulation (HPS) method was then used to address **RQ1**. Table 2.1 summarizes what research methods were used to answer the research questions.

---

\(^4\) Player experience refers here to six player affective states: fun, challenge, frustration, predictability, anxiety and boredom [8].
Table 2.1 Research methods used to answer research questions

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Research methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1.</td>
<td>Simulation (see Section 2.2.2 and Section 3)</td>
</tr>
<tr>
<td>RQ1.1.</td>
<td>Literature review and discussion with experts (see Section 2.2.1.)</td>
</tr>
<tr>
<td>RQ1.2.</td>
<td>Literature review and discussion with experts (see Section 2.2.1)</td>
</tr>
</tbody>
</table>

2.2 Research design

The following subsection describes in detail research steps of the undertaken research methodology process. It presents how RQ1.1 and RQ1.2 were answered and explains how obtained answers were used to design the HPS method. Finally it describes the idea behind the HPS method and motivates why this method was used.

2.2.1 Literature reviews and discussions with experts

The first two steps of the research were to identify existing procedural level generators for 2D platform games and existing player experience evaluation methods. In order to make a comprehensive selection from as many generators and explore as many methods as it was possible two literature reviews had been performed. The following databases were searched using an iterative approach:

1. Engineering Village web-based discovery platform with Inspec and Compendex databases
2. IEEE Xplore Digital Library
3. ACM Digital Library

During the initial phase of the level generators literature review the following keywords and their variations were used: *procedural level generation, automatic level generation, PLG, procedural content generation + levels, PCG + levels, procedural level generator, 2D platform level generator*. This set was later extended by keywords related to specific level generation techniques (*grammatical evolution, design patterns, search-based, genetic*) identified after reviewing articles from earlier iterations. The keywords for the player experience related
literature review were mainly: player experience evaluation, player experience model, procedural content generation + player experience. Both groups of identified articles were filtered based on specific inclusion and exclusion criteria. The inclusion criteria were:

1. Articles written in English and available in full-text
2. Articles which domain was related to Computer Games or Computer Software
3. Articles which mentioned procedural level generation or player experience in their abstracts
4. Articles which content was related to level generators that could be applied in a 2D platform game
5. Articles whose co-author was J. Togelius (a prominent researcher in PCG field) or papers which come from conferences and symposiums related to games were selected with higher priority

The exclusion criteria were:

1. Articles describing level generators for other video game genres, not applicable for 2D platform games (applied only to papers from the literature review about level generators)
2. Articles in which player experience was only mentioned and in fact focused on different subject (applied only to papers from the literature review about player experience evaluation methods)

The study selection process was made up of 5 sequential stages - the output from one stage was the input for another. During the first stage papers, which were written in English and contained the specified keywords were selected. In the second stage the researchers read titles and short article descriptions (not abstract) returned by the database search engine. The next stages were about reading the abstract, then reading the introduction/background, conclusions and skimming through the references (in some cases analysing the figures and tables). Finally, the most relevant papers were fully read.

The results of the first literature review (about existing procedural level generators) included 23 papers describing 17 different level generators. The data obtained were analysed and discussed with experts including thesis supervisors, whose main research areas are applied artificial intelligence in games (dr. Johan Hagelbäck), game design and affective computing (dr. Mariusz Szwoch) and a professional game developer from Alien Worm studio who has more than 10 years of experience in developing 2D games. Based on the discussion the following selection criteria for choosing level generators for further evaluation had been established:
1. The level generator should be possible to implement based on the description from the article or its implementation should be available online or from its authors.

2. The level generator should be known inside the PCG research community – it should be referenced by other researchers.

3. Level generators should represent the most popular constructive approach for generating levels (see Section 4.2.1) on different levels of complexity. Experts indicated that it would be interesting to compare simple level generators (easy to implement, based on simple algorithms) with more complex solutions.

4. To have a broader picture, one level generator should represent different approach than constructive based.

5. Because of the limited time frame of the Master Thesis project the number of selected level generators was set to four.

Here RQ1.1 was answered and the following four procedural level generators were selected (see Section 4.2 for their detailed description and motivation of their choice):

1. Random level generator [14] – basic example of constructive approach
3. Occupancy-Regulated Extension level generator [22] – complex example of constructive approach

### 2.2.2 HPS method

During the process of the research methodology selection for answering the RQ1 a few different approaches to the player experience evaluation method had been considered. The main challenge was to find a balance between solid, reliable results, making the research outcome possible to be used inside the industry and fitting into the limited time frame of the master thesis project. The existing solutions raised too many problems, thus under the guidance of the thesis supervisors and the industry expert a new method had been developed.

The most commonly used approach for player experience evaluation purpose is a survey conducted on human players - it has already been applied multiple times during the Mario AI Championship. However, using the human-based method we can only test a small number of
levels and collect a limited amount of information. This process is both time-consuming and difficult in terms of collecting an appropriate group of survey participants. The process of filling in a sophisticated questionnaire to gather more specific data requires more experienced players with better knowledge within game design aspects. Each procedural level generator may produce thousands of different levels, thus it is necessary to analyse player experience data from as many of them as it is possible.

The second literature review (about player experience evaluation methods) resulted in 6 articles. Two of them described quantitative models for predicting player experience based on actions performed by the player in the game, and on parameters of the level that was played [8], [18] (see Section 3.4). The predicted values corresponded to fun, challenge, frustration, predictability, anxiety and boredom caused by the game level design. After a discussion with experts (thesis supervisors and game developer from Alien Worm studio) a decision was made to use these quantitative models of player experience together with a simulation of the human player's behaviour in a 2D platform game environment. The simulation based method helped to overcome problems related to human-based methods and analyse data that would be difficult to obtain in any other way. It both speeded up and automated the whole process of levels evaluation.

In order to use the human player simulation method we needed a bot, which behaviour was a good estimation of a human player’s playing style. This led us to the third literature review, performed in the same way as the previous ones. The keywords searched for were: 2d platform game + bot, mario + bot, human-like + bot, believable + bot, believability + bot, Turing test + bot, 2d platform game + human-like + bot, mario + human-like + bot. The final search results included 10 articles about different bots, bots’ believability and the Turing Test Track of the Mario AI Championship. We have decided to choose a bot that has won the Turing Test Track of the Mario AI Championship held at CIG 2012 called VLS bot [16], [17] (see Section 3.3 and Section 4.3 for detailed description and motivation). 73 judges evaluated it there and it has appeared to be the most human-like bot on the competition. Here the RQ 1.2 was answered. The components that the automatic evaluation method for procedural level generators should consist of were:

1. Quantitative models of player experience
2. A simulation of the human player’s behaviour based on a human-like bot – the VLS bot
The method is later called human player simulation (HPS) method and it was used to answer the RQ1. It has been implemented with a use of the existing Mario AI Benchmark tool thoroughly described in [12], extended by adding to it a simulation data-collection component based on an artificial human-like bot player. The data-collection component was responsible for monitoring the set of selected gameplay features during the bot’s play and saving gathered information for further evaluation. The gathered data were then used as inputs to quantitative models of player experience (the models described in [8]). Each level was characterized by six numerical values - each representing a different affective state: fun, challenge, frustration, predictability, anxiety and boredom. The higher the value was, the stronger was the influence of the level on player’s experience from the perspective of the specific emotion. Finally, all selected level generators have been evaluated based on the values assigned to hundreds of generated levels. The HPS method and all its components are described in Section 3: HPS method – system design.

2.3 Validity threats

During the research a set of potential validity threats were identified and analysed. First of all it was necessary to answer a question: how generalizable are the results obtained from the HPS method? The bot played a sampled number of levels from each level generator and a too small sample size could affect the ability to generalize the study outcomes. In order to address this threat a method based on the simple moving average was used to test if the number of performed simulations was sufficient (see Section 5.3.1). The verification showed that the sample size was sufficiently large.

Another potential threat was related to the selection of the VLS bot as the human-like bot component of the HPS method. It was important to use the most human-like bot as possible that could be considered representative to the average human player. The human-likeness of the VLS bot was verified during the Turing Test Track of the Mario AI Championship in 2012, where after being evaluated by 73 spectators it was decided to be the most human-like bot on the competition. The representativeness of the VLS bot was increased thanks to the fact that its parameters were tuned through an experiment described in [17], where it was compared with human players, selected from a group of different players with various playing styles. During the literature review process a few other bots were identified, but none of them fulfilled the above-
mentioned criteria. Most of them were designed with different goals than human-likeness e.g. completing a level as fast as possible. Such bots would not be able to gather all data required by the player experience models and they could not be considered representative to human player.

A potential threat of decreasing the VLS bot’s human-likeness was also associated with additional assumptions and improvements applied to the original implementation of the VLS bot (see Section 4.3). Even though the original implementation was modified, the artificial potential fields algorithm responsible for the bot’s behaviour remained unchanged. In fact, all changes made to the VLS bot increased its human-likeness as they addressed problems that were causing completely not human-like behaviours e.g. from time to time the bot got stuck in places that would not be a problem for a human player (like dead ends). Moreover, the assumptions made in Section 4.3 were necessary from the perspective of the player experience models, as they allowed for gathering more comprehensive data about levels.

Finally, there was a potential threat that the results obtained with the use of the HPS method would not be consistent with human evaluation. During the literature reviews and discussions with experts a special attention was paid to selecting components built, trained and verified by humans. The VLS bot was trained with human and the player experience models were built based on data gathered from human players. In order to check if the method was consistent with human evaluation, we conducted a limited-scale experiment described in Appendix D. Whereas the verification was limited, it showed that the HPS method indications where the same as the human indications in most cases (it was different only in 2 out of 28 pairs of levels).
3 HPS METHOD - SYSTEM DESIGN

This chapter explains the main idea behind the HPS evaluation method for PLG and presents how the system has been designed (see Section 3.1). It further provides a basic knowledge about what a level generator and a bot component are and describes in detail the selected VLS bot controller. Finally we look into the implemented quantitative models of player experience and we explain how they were utilized in our research.

3.1 Simulation based evaluation - overview

The process of the simulation based evaluation is structured according to three macro phases depicted below in Figure 3.1: Level generation, Simulation of human player behaviour and Evaluation of player experience. In the first step a set of selected level generators (see Subsection 3.2) is used to generate a certain number of game levels. All levels are produced and saved as separate files using the same binary data format, thus they can be utilized together during the next phase without any additional modification. In order to simplify the data analysis process, each file is named using the following convention: level_PLGGenerator_id.lvl.

During the simulation of human player behaviour phase an artificial bot player (see Subsection 3.3) plays the selected platform game with all previously generated levels and tries to complete as many of them as possible. Each bot’s step is monitored by the gameplay data recorder component and all data required by the quantitative models of player experience (see Subsection 3.4) are recorded and stored in proper data structures. The system has been designed to make the player experience evaluation process possible both internally (within the system) immediately after the bot finishes its task or externally by exporting the gameplay data and using additional tools allowing for better visualisation or data presentation. In addition to the internal implementation we have created both a Microsoft Excel spreadsheet composed of several sub-spreadsheets and MATLAB scripts, which let us to simplify our analysis and to verify our model implementation by comparing results obtained from the system and the external tools.
3.2 Level generators

A procedural level generator is a tool implementing a selected PLG algorithm for game level content creation. It can be built on a large variety of different techniques, including grammar-based methods [6], [9], genetic algorithms [11], reinforcement learning [13] or pattern-based approaches [10]. In case of 2D platform games similar to Super Mario Bros many of them construct levels from small hand-authored pieces called *chunks*. The type of chunks and the method of putting them together depend on the PLG technique.
From a technical perspective a level generator can be implemented as a part of an existing game, external set of scripts or complex, visual tool as shown in Figure 3.2. As the aim of the research was not only to evaluate existing PLG solutions, but also to provide game developers with a reusable level generators evaluation method it was crucial to make the system independent of level generators implementation. In order to do that we designed our own binary data format representing generated levels and we used it in case of all generators. This approach improved reusability, avoids needing to reimplement existing, available generators (in each case we only needed to make some adaptations and to convert generated levels to the common data format) and allowed us to spend more time on generator that we had to implement from scratch.

![Figure 3.2 The Tanagra intelligent level design tool with level generator [5]](image)

### 3.3 Bot component

For many years researchers from the Computational Intelligence field together with professional game developers have been interested in creation of bots that can play a game as well as or even better than human. The most common goal remains the same and it is to provide real players with an adequate, challenging opponent or a useful, independent ally. In case of
platform games this led us to bots getting to the end of the level by traversing it from left to right as fast as possible and achieving the highest score by collecting coins and killing enemies. Since 2009 such bots take part in the Mario AI Competition organized by Julian Togelius and Sergey Karakovskiy in association with the IEEE Games Innovation Conference and the IEEE Symposium on Computational Intelligence and Games [14]. Even during the first edition of the competition many different bot implementations were presented, such as Robin Baumgarten’s A* agent (see Figure 3.3), Slawomir Bojarski’s and Clare Bates Congdon’s REALM agent (a rule-based evolutionary computation agent), Erek Speed’s rule-based agent or the organizer’s Forward Agent. After the contest in 2009 it was concluded that the playing style exhibited by the winning agents was nothing like that demonstrated by human players [14]. This issue has been addressed in subsequent editions of the competition by creating the Turing Test Track of the Mario AI Championship, focused only on developing human-like controllers.

![Figure 3.3 Robin Baumgarten’s Mario bot and a visualisation of its A* search algorithm](image)

Since the main goal of our research was to evaluate generated levels in terms of player experience, we needed a bot, which behaviour is a good estimation of a human player’s playing
style. Julian Togelius et al. in [15] call it *player believability*, i.e. when someone believes that the controller (human or bot) controlling the character is a human. During the last Turing Test Track of the Mario AI Championship held at CIG 2012 73 spectators evaluated believability of three new bot implementations and the winner was the VLS bot (named after the first names of its authors Vinay, Likith and Stefan) [16]. Each judge had to watch two pairs of videos presenting different bots (or a human player) playing the game and answer which bot seemed to be more human-like. We have decided to choose the winning one and adapt it to our requirements (see Subsection 4.3).

The VLS bot is an artificial potential field (APF) based controller and it has been initially implemented as a result of a Master Thesis project at the Blekinge Institute of Technology in Sweden [17]. The base idea of applying APF technique in a platform game is to split the world into a grid with several look ahead positions for the character and calculating for each of them local attracting and repelling forces (see Figure 3.4, which comes from [17]).

![Figure 3.4 VLS bot and the look ahead positions grid](image)
The authors of the VLS bot defined four fields influenced by different aspects of the game:
1. The field of progression – it is more attractive for the character to go right instead of left. The difference cannot be big, as we still want the bot to consider other sources of attraction.
2. The field of rewards – power-ups and coins attract the character.
3. The field of terrain – gaps and other dangerous positions repel the bot (see Figure 3.5 from [17]).
4. The field of opponents – opponents repel the player, but since in Super Mario Bros game some enemies can be killed by jumping on their heads, the position above them is attractive (see Figure 3.6 from [17]).

![Figure 3.5 VLS bot and the terrain field](image1)

![Figure 3.6 VLS bot and the enemies' field](image2)

Thanks to such approach the bot player instead of following some pre-defined set of rules can behave in a more human-like way by applying more real-world approach for making certain decisions e.g. collecting attractive items or avoiding repelling enemies. The potential field parameters have been tuned through an experiment with human players described in [17].

### 3.4 Quantitative models of player experience

As we wanted to have the ability to automatically evaluate and compare generated levels, we needed to implement a quantitative model based on precise numerical data collected by the bot
player. Following these conditions, we have decided to adapt a theory introduced by Pedersen, Togelius and Yannakakis in [18] and [8]. In both articles they work on quantitative models for predicting certain player affective states based on actions performed by the player in the game, and on parameters of the level that was played. The predicted values correspond to fun, challenge, frustration, predictability, anxiety and boredom caused by the game level design.

At the first step Pederson et al. defined three types of data they were gathering:

1. Controllable features of the game, i.e. the parameters directly describing the generated level e.g. the number of gaps in the level or the average width of gaps.
2. Gameplay characteristics, representing player’s skill and playing style in a particular game level e.g. number of time the player was killed by an opponent or by jumping into a gap, number of collected items or the game completion time.
3. The player’s experience of playing the game, based on ranking the games by players in order of emotional preference through a questionnaire.

With the use of an online game survey, based on a modified version of Infinite Mario Bros\(^5\), authors collected data regarding game controllable features and gameplay characteristics. Information about relevant player emotions were obtained at the end of the survey through forced choice questionnaires. These data were utilized together to determine a relationship between reported emotions and extracted features. The selected features were:

\[
\begin{align*}
\mathbf{n}_s & \quad \text{number of times the player kicked an opponent shell} \\
\mathbf{C} & \quad \text{whether the level was completed or not} \\
\mathbf{n}_{cb} & \quad \text{number of coin blocks pressed over the total number of coin blocks existent in the level} \\
\mathbf{n}_p & \quad \text{number of power-up blocks pressed over the total number of power-up blocks existent in the level} \\
\mathbf{k}_T & \quad \text{the total number of kills over the total number of opponents} \\
\mathbf{d}_j & \quad \text{number of times the player was killed by jumping into a gap over the total number of deaths} \\
\mathbf{n}_r & \quad \text{number of times the run button was pressed} \\
\mathbf{d}_g & \quad \text{number of times the player was killed by jumping into a gap} \\
\mathbf{t}_L & \quad \text{percentage of time that the player was moving left} \\
\mathbf{J}_d & \quad \text{jump difficulty heuristic, which is proportional to the number of the player’s deaths due to gaps, number of gaps and average gap width} \\
\mathbf{k}_P & \quad \text{number of opponent kills minus number of deaths caused by opponents} \\
\mathbf{E}\{G_w\} & \quad \text{the average width of gaps}
\end{align*}
\]

\(^5\) The game and questionnaire are available at www.bluenight.dk/mario.php
\( n_i \) number of collected items (coins, destroyed blocks and power-ups) over total items existent in the level
\( t_r \) percentage of time that the player was running
\( n_d \) number of times the player ducked
\( t_s \) percentage of time that the player was standing still
\( t_l \) playing duration of last life over the total time spent on the level
\( k_f \) number of opponents died from fire-shots over the total number of kills
\( G \) number of gaps
\( n_c \) number of coins collected over the total number of coins existent in the level
\( t_R \) time player was moving right

For the need of the automatic method for evaluation of procedural level generators the human-like bot player plays each generated level and collects the following features that have influence on the fun, challenge, frustration, predictability, anxiety and boredom emotions \((FF, CF, FRF, PF, AF, BF)\) based on the data from [8]:

\[
\begin{align*}
FF &= \{n_w, n_{cb}, k_T, n_r, t_L, k_p, t_r\} \\
CF &= \{C, n_p, d_p, d_g, J_d, E(G_w), n_d, t_L, n_{cb}, G\} \\
FRF &= \{C, n_p, n_{cb}, d_p, d_g, d_i, n_i, t_s, k_f, t_L, n_c\} \\
PF &= \{J_d, E(G_w), d_i, C, d_g, t_L, t_R\} \\
AF &= \{C, d_i, E(G_w), d_g, J_d\} \\
BF &= \{E(G_w)\}
\end{align*}
\]

(1)

Then the obtained data are used to calculate the values of specific emotions induced by level \( l_i \) by summing up the normalized feature values of the level multiplied by the corresponding coefficient \( c(x) \) determined by Togelius et al. in [8]:

\[
\begin{align*}
\text{fun}(l_i) &= \sum_{x \in FF} c(x) \ast \text{norm}(\text{value}(l_i, x)) \\
\text{challenge}(l_i) &= \sum_{x \in CF} c(x) \ast \text{norm}(\text{value}(l_i, x)) \\
\text{frustration}(l_i) &= \sum_{x \in FRF} c(x) \ast \text{norm}(\text{value}(l_i, x)) \\
\text{predictability}(l_i) &= \sum_{x \in FF} c(x) \ast \text{norm}(\text{value}(l_i, x)) \\
\text{anxiety}(l_i) &= \sum_{x \in AF} c(x) \ast \text{norm}(\text{value}(l_i, x))
\end{align*}
\]

(2)
boredom(l_i) = \sum_{x \in BF} c(x) \ast \text{norm(value}(l_i, x))

where: \( l_i \) is the \( i \)th generated level and \( x \) is the feature from the corresponding set of features.

In the end all levels were ranked based on the estimated emotion values and sorted in descending order by fun, challenge, frustration, predictability, anxiety and boredom. By using this model it is possible to compare individual levels as well as the procedural level generators based on the numerical values assigned to their levels.
4 IMPLEMENTATION

This chapter looks into implementation details of the HPS method components and the selected level generators. Section 4.1 describes the adaptation process of the Mario AI Benchmark for the need of our research work. Section 4.2 presents all of the four selected procedural level generators. Finally, the last section provides knowledge about the AI human-like bot player improvements and the assumptions made during its implementation process.

4.1 Platform game engine (Mario AI Benchmark)

From a technical perspective the core functionality of the HPS method has been implemented as an extension to the existing Mario AI Benchmark thoroughly described by Julian Togelius and Sergey Karakovskiy in [12]. The benchmark is a game-based tool utilizing a public domain clone of the original Super Mario Bros game made by Nintendo. It has been used in several AI researches and scientific competitions organized during international academic conferences since 2009 [14]. It provides the developer with a number of programming interfaces that can be used as required and organizes them in a convenient way. Thanks to Java implementation it is possible to run the benchmark on multiple different platforms and systems without any modifications. Hence, it made the presented player experience evaluation method available for all game developers regardless of their development environments (Windows, Linux, Mac OS etc.).

The first step of adapting the benchmark to HPS method needs was to modify the existing way of loading generated levels to use the designed binary format, common for all level generators. The raw binary levels’ representation method had been chosen because it results in a decreased size of generated files and shorter time of parsing (in comparison to storing objects), which are important in case of analysing thousands of levels. Additionally it makes level files easier to use with different technologies like C++, which is the most popular programming language among game developers.

The next step was to configure the benchmark in such a way that the bot was able to play multiple levels sequentially with or without game visualization. The visualization is helpful when the developer wants to analyse certain levels, but it should be turned off during the automatic evaluation process. The gathered data was used together in the estimation of player affective states with the proper parameters of the player experience models. Even though the
The original benchmark was able to collect some gameplay related features, a number of new metrics had to be added:

- Number of times the player kicked an opponent shell.
- Number of coin blocks pressed.
- Number of power-up block pressed.
- Number of collected items.
- Number of times the run button was pressed.
- Percentage of time that the player was running.
- Number of deaths caused by a creature.
- Number of deaths caused by a gap.
- Number of times the player ducked.
- Percentage of time that the player was standing still.
- Percentage of time that the player was moving left.
- Percentage of time that the player was moving right.
- Information whether the level was completed or not.
- Total number of coins blocks inside the level.
- Total number of power-up blocks inside the level.
- Total number of items inside the level (coins, destroyed blocks and power-ups).
- The sum of all gaps’ width.
- The number of all gaps inside the level.

Finally, to support the research report with screenshots of selected generated levels the editor’s GUI has been modified and screen capture functionality of whole levels has been implemented into it.

## 4.2 Procedural level generators

The following subsection presents the four selected procedural level generators. Three of them: Random, Design pattern-based and Occupancy-Regulated Extension (ORE) were chosen as examples of the popular constructive approach on different levels of complexity. The constructive solutions are confronted with a promising representative of search-based methods: the Feasible-Infeasible Two-Population (Fi-2Pop) genetic level generator.
4.2.1 Random level generator

The first selected procedural level generator is an example of the most basic, but very popular constructive approach based on traversing an empty level from left to right and adding random segments from a pre-defined library of chunks. All segments have the same height equal to the height of the level and they are placed one after another. The PLG algorithm does not contain any additional validity checks and does not guarantee that the output levels will be possible to complete. Hence, the library of chunks should not contain complex elements, which might not fit each other and the generated levels are simpler than those from the original Super Mario Bros game. As shown during the Mario AI Championship [14] many of levels generated by the Random generator can be completed by constant running and jumping at the right time without backtracking or looking for hidden passages. However, despite its limitations the generator had been successfully used in the Infinite Mario Bros game and had become the basic tool for evaluation of bot controllers in Mario AI Championship editions. From the automatic evaluation process perspective it was interesting to compare the very basic, but popular level generator with other, more complex and novel approaches. Two example levels generated by the Random generator are presented in Figures 4.1 and 4.2. They consist of small groups of the basic enemy and randomly placed platforms that have no big influence on the levels’ difficulty.

Figure 4.1 An example level generated by Random generator split into three fragments
Figure 4.2 An example level generated by Random generator split into three fragments

4.2.2 Design pattern-based level generator

The concept of the design pattern-based level generator is an extension of the basic constructive random generator described above. It focuses on utilizing specific design elements that guarantee more enjoyable experience for the player. The main challenge in such approach is to build a library of proper design patterns. Dahlskog and Togelius in [10] solved this problem in case of the Infinite Mario Bros game by performing an analysis of rhythm groups found inside the original Super Mario Bros levels, designed by Nintendo. They assumed that by extracting patterns from levels prepared by Nintendo designers and evaluated by millions of players they will be able to generate as much fun levels as the original ones. As a result they identified 23 different patterns separated into five groups: enemies, gaps, valleys, multiple paths and stairs. Each pattern has its own purpose and was designed in order to solve a specific game design problem e.g. a tight formation of four enemies (4-Horde pattern) is not possible to be passed over with a long jump and it forces player to jump on one of the enemies.

For the need of the automatic evaluation of procedural level generators the pattern-based generator had been implemented from scratch using all 23 patterns listed in [10]. Dahlskog and Togelius suggested that patterns could be parameterized for higher variety, thus each of the implemented patterns had randomized length, number of gaps, length of a platform, enemy types and sometimes the number of enemies. Since it was not remarked what authors did in order to avoid strange repetitions of the same pattern (several enemy patterns next to each other is something not encountered in the original Super Mario Bros) the generator had been designed to
not combine patterns from the same group. The example levels generated by the final version of the generator are shown in Figures 4.3 and 4.4. By comparing results obtained for this approach and other generators including two constructive methods: straightforward Random and more advanced ORE it was possible to see how simple ideas and improvements may affect the player experience and how they behave in comparison with more complex solutions.

![Figure 4.3 An example level generated by Pattern-based generator split into three fragments](image)

![Figure 4.4 An example level generated by Pattern-based generator split into three fragments](image)

### 4.2.3 Feasible-Infeasible Two-Population genetic level generator

While many existing procedural level generators for platform games use complex and nested rule-based, iterative approaches there are examples of different solutions. Since the research field is relatively new and they often have not yet been compared with each other, they were
considered to be of special interest during this research. Such a novel, evolutionary computational approach for procedural level generation task, combining genetic algorithms (GA) and constraint satisfaction (CS) methods has been proposed by Nathan Sorenson, Philippe Pasquier and Steve DiPaola in [19] and [20]. It has been built upon authors’ previous work about challenge-based model of fun described in [21], which is used by them as one of their fitness functions in the GA implementation. As reported in [1] the generator has participated in the 2010 Mario AI Championship Level Generation Track and has taken the third place out of six. The assessment of 15 judges showed that Sorenson’s search-based method can compete with more popular, constructive approaches and it is worth further investigation.

In order to address the complexity of a level generation process in an evolutionary way Sorenson et al. have split the process into two phases. Because the generated level not only has to be optimized in terms of fun, but also has to fulfil a set of constraints making it possible to complete, authors implemented the Feasible-Infeasible Two-Population genetic algorithm (FI-2Pop). They created two separate populations of level designs, where genotype was based on ordered design elements and evolved them in parallel by two different fitness functions. One population contained only feasible levels (all constraints fulfilled) and was evolved in terms of fun, while the other contained only infeasible levels, evolved in terms of satisfying constraints. If a level from one population changed its status from infeasible to feasible or the opposite, it was moved to the other group.

As mentioned above, the feasible population fitness function utilizes the authors’ earlier challenge-based model of fun. The model predicts how fun levels are based on their distribution of challenge, calculated with a use of a challenge metric $c(t)$. In case of a 2D platform game like Super Mario Bros, Sorenson et al. defined a formula describing how difficult it is to jump over a particular gap (enemies are also represented as gaps in this model). It is presented below, where $d(p_1, p_2)$ is the distance between platforms $p_1$ and $p_2$, $fp$ is a jump margin error and $2fp_{max}$ is a constant making the metric positive:

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

The algorithm looks for changing periods of high and low challenge that create together rhythm groups, described by Smith et al. [9]. Level designs that contain more rhythm groups with an appropriate amount of challenge (in comparison to the assumed, ideal amount of difficulty) are scored higher and they are selected by the genetic algorithm.
Thanks to Nathan Sorenson and Philippe Pasquier it was possible for us to use their original source code of the FI-2Pop generator written in JAVA and Clojure. As the automatic method for evaluating the procedurally generated levels was designed to be independent from generators’ implementation, the fact of using Clojure programming language had no influence on the evaluation process. Two example levels generated by the FI-2Pop generator are presented in Figures 4.5 and 4.6.

![Figure 4.5 An example level generated by FI-2Pop generator split into three fragments](image)

![Figure 4.6 An example level generated by FI-2Pop generator split into three fragments](image)

### 4.2.4 Occupancy-Regulated Extension level generator

The Occupancy-Regulated Extension (ORE) based level generation algorithm has been presented by Peter Mawhorter and Michael Mateas in [22]. In contrast to many other existing
approaches that focus on building the level within strict constraints and domain-specific gameplay mechanics it does not require game-related knowledge on the implementation level. This may lead to more interesting, original level designs that would not emerge in case of more limited generators and can to some extent simulate the human creativity.

The main idea of the algorithm is based on extending the initial level by adding selected level chunks from a pre-defined chunk library depending on the current state of the level and possible positions that the player can occupy (occupancy) during the gameplay called *anchors*. Each level chunk has its own anchors that can be used as attachment points to the existing level structure or in case of already used chunks serve as positions for further level extension. The concept of attaching new level chunks has been depicted in Figure 4.7. In the top left corner there is an existing, partial level with an anchor. The top right corner shows the selected level chunk with its 2 anchors. In the bottom left the level chunk has been attached to the partial level in the anchor position that has been marked as used. The unused (colourful) anchor can be utilized iteratively for further expansion. In the last part there is the final level after post-processing.

![Figure 4.7 The idea of level extension by attaching level chunk in anchor position](image)

*Figure 4.7 The idea of level extension by attaching level chunk in anchor position [22]*
As suggested by Mawhhorter and Mateas, the ORE algorithm can be split into three phases:
1. Decide which anchor from the existing level will be used for further expansion (context selection).
2. Select a compatible chunk from the library by filtering all available chunks.
3. Attach the selected chunk to the existing level.

The only domain-specific elements are the library of chunks and the final, post-processing algorithm. Two example levels generated by the ORE technique are presented in Figures 4.8 and 4.9. Even by examining only these examples it is possible to make an observation that ORE levels can be much more complex, varied and unpredictable than those generated by previously described techniques. They often consist of multiple paths for passing a single fragment of a level and it is not easy to predict, which path is the best one. As it can be seen in Figure 4.8, the first part of the level can be traversed both by going on the top of it (above stone stairs) or by going under the ground. The second path is much more risky, but it offers coins as a reward. It is worth to point out that this path is available only for the small form of the Mario character and the player may want to be hit by an enemy on purpose in order to pass it. In case of the level from Figure 4.9 (the middle part) there is also a path that is a deadly trap. Once the player enters it, it is not possible for him to pass it or leave it. The player has to commit a suicide. Such complexities are not present in any levels generated by the other generators.

Figure 4.8 An example level generated by ORE split into three fragments
By courtesy of Peter Mawhorter and Michael Mateas we were able to utilize their latest Java implementation of the ORE generator and their most recent library of chunks tested during the level generation track of the Mario AI Championship in 2010, 2011 and 2012. One of the reasons for selecting the ORE algorithm in our research, except its non-standard context-based approach and high variety of generated levels was the fact that it was the winning algorithm in the 2011 edition of the competition\(^6\).

### 4.3 AI human-like bot player

In the previous chapter regarding the HPS method design we have described the main idea of the VLS bot controller and we explained the main reasons of its selection. Thanks to Vinay Ethiraj’s and Likith Satish’s help it was possible to work with the original code that they used during the Mario AI Championship. Although it has appeared to be the right choice at the end of the research, we had to spend more time to adapt and improve the bot than we initially planned. After a few tests it became clear that levels produced by selected generators are much more complex and difficult to complete than those from the competition. The initial version of the bot was constantly dying, falling into gaps and permanently getting stuck on different obstacles. For the purpose of the competition Ethiraj and Satish did not need to consider and address many issues that were raised by more complex levels. Because of that it was necessary to enhance the

---

\(^6\) [http://www.marioai.org/LevelGeneration/Results](http://www.marioai.org/LevelGeneration/Results)
bot implementation and make some additional assumptions without changing the idea behind the controller and influencing its human-likeness. By testing it on thousands of different levels we identified a set of shortcomings and bugs not related to the player believability factor, but definitely worsening bot’s results. Some of them are presented in Table 4.1 together with a description of how they were overcome.

Table 4.1 Identified shortcomings and bugs of the VLS bot and how they were overcome.

<table>
<thead>
<tr>
<th>Problem description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>The bot was not able to jump over wide gaps.</td>
<td>There were no gaps at all in levels generated during the Mario AI Championship. The VLS bot was able to jump over small gaps, but was not programmed to use long jump (run button pressed) over wide gaps. It has been modified to use the run button while jumping.</td>
</tr>
<tr>
<td>In some situations the APF algorithm led the bot to irrational, bad decisions by not considering certain factors and all possible solutions e.g. bot was often jumping when it was under Bullet Bill instead of waiting until the Bullet Bill will pass by.</td>
<td>A set of additional methods was implemented in order to detect dangerous situations and prevent the bot from making bad decisions. If the action chosen by the APF algorithm was classified as leading to death, then another action was taken depending on the context.</td>
</tr>
<tr>
<td>It was common for the bot to get stuck in one place when it was on a platform built from bricks with a few coins or a power-up below the platform.</td>
<td>The APF algorithm was making the final decision based on all look ahead positions, even if the bot was not able to reach them. The coins under the platform or bricks with power-ups were attracting the bot and they were not letting it to move forward. The algorithm has been modified to ignore forces from places that are not reachable at the moment (e.g. under a wall).</td>
</tr>
<tr>
<td>The bot was not able to solve more</td>
<td>The visual analysis of the bot’s behaviour</td>
</tr>
</tbody>
</table>
complex problems, which required specific sequence of actions e.g. a vertical wall higher than five bricks on its way (impossible to jump over) preceded by a platform. The APF bot did not know, that it should first jump on the platform and then perform a long jump from the edge of the platform above the high wall. performed on 300 different levels showed that there were only a few different cases that were common among levels and were causing significant problems to the bot. The bot has been extended by a set of additional rules and actions letting it to pass them.

| It happened from time to time that the bot got stuck in some places that would not be a problem for a real player like dead ends. | If the bot was not able to move further to the right for more than 10 seconds the progression potential field was reversed for the next 10 seconds. Thus the bot gained another chance to choose the good way. |

| Other problems and shortcomings. | During the extended visual analysis of bot’s play many other, small problems have been found and overcome. Some of them were strictly programming errors like division by zero or returning NaN values in specific cases. |

The most important aspect from the player experience evaluation perspective is the amount of data gathered from a single level. If we stop the data gathering process after the bot dies or gets stuck in the beginning of a level we will not have any information about the further sections. Despite many improvements and fixes the bot was still far from completing most of the generated levels. A human player may also not be able to finish a level during the first play, but he or she progressively learns how to do it. As the VLS bot plays the level in a deterministic way and it is not able to learn, there was no point in letting the bot to play a single level multiple times. Instead, we have decided to make the following assumptions:

1. The bot was invincible (immune to enemies) and the data-gathering component just saved the places and the number of dangerous events in which the bot would normally die.
2. After the bot fell into a gap it was dropped down from the top of the level 4 blocks further to the right.
3. If the bot got stuck in the same place more than once and reversing the progression potential field did not help, then the bot was also dropped down from the top of the level 4 blocks further to the right.

4. For the purpose of the player experience evaluation model calculations we considered a level to be completed if there were no more than 2 places where the bot would normally die.

Thanks to the above-mentioned assumptions the gameplay data recorder was able to gather detailed information about all levels without the risk that the process was interrupted in the beginning part of a level.
5 Simulation

This chapter describes the whole process of simulation the human player behaviour in the IMB game, i.e. how the simulation was configured and run, the results of the performed simulations and three kinds of analysis of these results.

5.1 Simulation environment and parameters

Simulation of the human player behaviour is the main part of the HPS method (see Figure 3.1). Before the right simulation can begin, the set of levels for each evaluated generator has to be produced. After this step, the simulations with the human-like bot are run, the bot plays the game with the generated levels and for each level the gameplay data are recorded. Finally, the gameplay data corresponding to all generators and levels are the inputs to the quantitative models of player experience. The values output by the models are the simulation results.

We followed exactly the above described steps. All of them were performed on a MacBook Pro Early 2011 computer (2.7 GHz dual-core Intel Core i7 processor, 16 GB of 1333 MHz DDR3 memory) with Mac OS X 10.9.2. Level generation and simulation phases were performed within Eclipse Indigo 3.7.2. For player experience evaluation phase we used Microsoft Excel and MATLAB.

The simulation results were obtained in the following way:

1. Each of the four procedural level generators (Fi-2Pop, ORE, Patterns, Random) was run 1500 times, producing 6000 levels in total. It should be noted that each generator was run with its default input parameters (in agreement with the thesis assumptions). It also should be noted that each level generation was an independent stochastic process.

2. The next step was to run simulations with the human-like bot. The bot played the IMB game with the generated levels and for each level the gameplay data was recorded.

3. The gameplay data corresponding to all generators and levels were the inputs to the quantitative models of player experience. Each level was characterized by six numerical values - each representing a different affective state: fun, challenge, frustration, predictability, anxiety and boredom.
The sample size of 1500 was chosen for two reasons:
1. the sample sizes that were used by the PLG researchers in metrics-based evaluations were 1000 [24] or 10 000 [23]; we wanted to generate the samples as large as possible, but not less than 1000;
2. the level generation phase was time-consuming - the cause was the Fi-2Pop generator which is based on the genetic algorithm. It took Fi-2Pop around one minute to generate one level (so 1500 levels were generated by that generator in around 25 hours without a break); since we had a limited-time frame for the thesis project and the computer used for level generation and simulation phases was also used for everyday work on the thesis, we decided to carry out the process of generation and simulation within two days\(^7\). In Section 5.3.2 it will be shown that the sample size of 1500 is sufficiently large.

The main assumption of the thesis was to run the four generators with their default settings and values. We did not type in any input parameters when starting the generators, we did not modify any values in the source codes of the generators either. The reason for this was that the defaults are usually the most commonly selected options, and it also simplified comparison of four generators (every change would affect the simulation results).

5.2 Simulation results

It took the bot 9:11:05 minutes to play all 6000 levels and collect the gameplay data which were input to the models of player experience (on average one level was played in less than 0.092 second). The numerical values output by the models are the simulation results, all values are normalized to the [0-1] range.

The simulation results is a large collection of data - it consists of 24 samples of 1500 elements each (4 procedural level generators multiplied by 6 affective states). For our convenience we split it into four text files, each file representing one generator\(^8\). There are six columns and 1500 lines within each file, each column represents one emotion, each line represents one generated level. The order of the columns within each file is as follows - the first represents fun, the second

---

\(^7\) In fact, it took more than two days due to repeating the whole process several times because of some software/hardware problems.

\(^8\) The text files with the simulation results are available at https://www.dropbox.com/sh/wvf0mgzer9m2yk4/AADhwo7Gc0Ln_i3xz6hsjM-pa
represents challenge, the third represents frustration, the fourth represents predictability, the fifth represents anxiety and the sixth represents boredom. The order of levels within each file is the same as the order of both level generation and bot simulation.\(^9\)

### 5.3 Analysis and discussion

This section describes three kinds of analysis of the simulation results. The aim of the first one was to test the adequacy of the sample size. The aim of the other two was to answer RQ1.

#### 5.3.1 Maxima and minima of moving averages - testing the adequacy of the samples

The first step in the analysis was to assess if the sample size is sufficient to conduct the next steps of the analysis and draw general conclusions about the generators. To address this problem a method based on a simple moving average (SMA) was adopted. It was a convenient tool for examining the convergence of the averages for increasingly larger sample sizes.

The method (described below) was applied to 24 samples of 1500 elements each (4 procedural level generators multiplied by 6 affective states). In other words, the analysed samples were the sequences of 1500 numerical values corresponding to each emotion of each generator, all normalized to the \([0-1]\) range. It is worth stressing that the order of elements in each sample was the same as the order of both level generation and bot simulation.

The method consisted of two steps. First, the SMAs were computed for window sizes increasing from 10 to 500 by the step of 1 (example graphs are shown in Figs. 5.1, 5.3, 5.5, and 5.7). Next, the maximum and minimum values of each of the SMAs were found and plotted within one graph (versus the window size). All the computations were performed according to the following formulas:

\[
\text{sma}(n) = \frac{e(n) + e(n - 1) + e(n - 2) + \cdots + e(n - (w - 1))}{w}
\]

\[
y_{\text{max}}(n) = \max(\text{sma}(n))
\]

\[
y_{\text{min}}(n) = \min(\text{sma}(n))
\]

\(^9\) The order of elements was significant and had to be preserved, because the first type of analysis that was conducted was based on time series analysis (see Section 5.3.1).
where: \( n \) is the \( n \)th element in the sample, \( e(n) \) is the value of an affective state for \( n \)th element, \( w \) is the window size.

Figs. 5.2, 5.4, 5.6 and 5.8 show the graphs of the maxima and minima of the SMAs for all four generators and one particular affective state - fun. The complete set of graphs for all four generators and all six affective states can be found in Appendix A\(^{10}\). As seen in these four figures, the maxima and minima curves delimit the space where all computed averages lie. What is particularly important, these curves seem to converge to a common limit, which can be interpreted as the large sample average. In other words, the maxima and minima curves determine the range of computed averages which looks to contract as the window size increases. The rate of convergence for all generators and all affective states is different, but one can conclude from the graphs that the sample size of 1500 is sufficiently large. As seen in the graphs, even smaller sample sizes such as 400 or 500 would be adequate, but the larger the sample size, the more accurate the analyses.

\(^{10}\) The plots in Figs. 5.1-5.8 and Figs. A.1-A.24 were obtained using MATLAB. The complete source codes are presented in Appendices B (SMA) and C (the maxima and minima of the SMAs).
Figure 5.1 SMA of fun for the levels generated by Fi-2Pop, window size = 10 (left), 50 (right)

Figure 5.2 Maxima and minima of SMAs of fun for the levels generated by Fi-2Pop
Figure 5.3 SMA of fun for the levels generated by ORE, window size = 10 (left), 50 (right)

Figure 5.4 Maxima and minima of SMAs of fun for the levels generated by ORE
Figure 5.5 SMA of fun for the levels generated by Patterns, window size = 10 (left), 50 (right)

Figure 5.6 Maxima and minima of SMAs of fun for the levels generated by Patterns
Figure 5.7 SMA of fun for the levels generated by Random, window size = 10 (left), 50 (right)

Figure 5.8 Maxima and minima of SMAs of fun for the levels generated by Random
5.3.2 Averages and standard deviations - comparing the generators

After the adequacy of the sample size was empirically shown, the averages and standard deviations were computed for each sample of 1500 elements according to the following formulas:

\[
\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i
\]  

(7)

\[
s = \left( \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}
\]  

(8)

where \( n \) is the number of elements in the sample.

The averages and standard deviations are presented in bar-chart form in Fig.5.9. Each bar within each graph represents one generator and shows the average and standard deviation of a particular emotion.

As seen in Fig.5.9 a), the most fun levels are generated by Fi-2Pop, while the least fun are generated by Random. In the middle there are Pattern-based generator and ORE which have achieved very similar results (ORE is slightly better). This is interesting, because these both generators quite significantly differ from each other. Although they both have been classified as constructive generators, Pattern-based approach is much closer to Random than to ORE. The levels generated by ORE are the most unusual and the least similar to those of the original Super Mario Bros, compared to the levels generated by the other three generators.

As seen in Fig.5.9 a), the most fun levels are generated by Fi-2Pop, while the least fun are generated by Random. In the middle there are Pattern-based generator and ORE which have achieved very similar results (ORE is slightly better). This is interesting, because these both generators quite significantly differ from each other. Although they both have been classified as constructive generators, Pattern-based approach is much closer to Random than to ORE. The levels generated by ORE are the most unusual and the least similar to those of the original Super Mario Bros, compared to the levels generated by the other three generators.

As seen in Fig.5.9 a), the most fun levels are generated by Fi-2Pop, while the least fun are generated by Random. In the middle there are Pattern-based generator and ORE which have achieved very similar results (ORE is slightly better). This is interesting, because these both generators quite significantly differ from each other. Although they both have been classified as constructive generators, Pattern-based approach is much closer to Random than to ORE. The levels generated by ORE are the most unusual and the least similar to those of the original Super Mario Bros, compared to the levels generated by the other three generators.

It can be seen from Fig. 5.9 that three bar charts: b), c) and e) have similar shapes (but different y-axis scales). From these bar charts one can observe that the most challenging, frustrating and anxious levels are generated by ORE. Random, the same as for the previously considered fun, reaches the lowest values for challenge, frustration and anxiety among all evaluated generators. The Pattern-based generator is again in the middle, but this time together with Fi-2Pop. While for challenge and frustration they both are very close to each other (the Pattern-based average is a little bit higher than the Fi-2Pop’s), the average value difference between both generators is rather noticeable in the case of anxiety. Again, what can be observed here are the similar results of the dissimilar generators (Pattern-based and Fi-2Pop), and quite different results between the most similar generators (Pattern-based and Random).
As seen in Fig.5.9 there are also another two similar bar charts: d) and f). This is the only situation when Random is placed first in two categories, as the most predictable and boring generator. Fi-2Pop also took first place as the most boring generator, but it got a second place in the case of predictability. The least predictable and boring, on the other hand, are levels generated by ORE. Although the Pattern-based generator got a third place (behind Random and Fi-2Pop, and before ORE) its results are closer to the first two generators.

What may be a bit surprising or incomprehensible to the reader is that one generator (Fi-2Pop) reaches the highest values in the most desirable emotion such as fun and at the same time it also reaches the highest values in the most undesirable emotion - boredom. This can be explained in two ways. First, it should be noted that these two emotions are not complementary (boredom is not a complement of fun, i.e. boredom is not 1-fun). Secondly, boredom is modeled less accurately than fun. According to the authors of the models of player experience [8] boredom is the hardest of the six reported emotions to predict, with the lowest accuracy. Of all the six models, the model of boredom takes into account the smallest number of features. For this reason, the model of boredom shows lower ability to differentiate levels than the other models (see Fig.5.9. graph f), especially Fi-2Pop and Random bars, Figs.A.21-A.24 in Appendix A, and Fig.5.15 in Section 5.3.3).

Combining the above observations, one may finally conclude that Random is the worst of the four generators. It reaches the lowest values in desirable emotions (such as fun, challenge) and the highest values for undesirable emotions (like boredom, predictability). While it is rather easy to identify the worst generator, it is quite tricky to do the same with the best one. Taking into consideration the most important emotion - fun, Fi-2Pop would be the winner. Taking into account more emotions, ORE could be considered the best with its most challenging and frustrating, quite fun, least predictable and least boring levels. Although the results of the Pattern-based generator are somewhere in the middle in all six categories, to some extent it can also be considered the winner - the results are quite good and this generator has the simplest implementation (together with Random) of all the generators discussed. Thus, the answer to RQ1, the main research question of the thesis, depends on what we rank first. If fun is ranked first - then Fi-2Pop produces levels that provide the best player experience. If we take into account a broader spectrum of emotions, then the ORE can be considered the best generator.
Figure 5.9 Comparison of the averages and standard deviations of the six affective states for all four generators.
5.3.3 Histograms - analysing the expressive ranges

The aim of the last kind of analysis was to examine the expressive range of each of the four generators. According to [23] expressive range is the quality of a generator referring to the style and variety of generated levels. In order to characterize the expressivity of a particular generator a set of metrics describing its levels has to be specified [23]. In our case these metrics are six affective states: fun, challenge, frustration, predictability, anxiety and boredom. A convenient way to visualise the expressive range is by plotting histograms [23], [24]. Thus, examining the expressivity means examining how the generated levels are distributed over a specified set of metrics. It is especially useful for analysing how different sets of the generator’s parameters affect its output or towards which types of levels a particular generator is biased. In our analysis we focused on this other aspect. Fig. 5.10 - 5.15 show the expressive ranges for all four generators along all six affective states. For each emotion and each generator we obtained a different distribution.

The expressive ranges of all four generators, as can be seen in Fig.5.10, appear to be slightly biased towards levels of low and medium fun (the highest bars are for the bins [0.2-0.3) and [0.3-0.4)). It should be noted that Fi-2Pop creates a wide variety of levels of different fun values - its levels are present in each of the histogram bins. This means that Fi-2Pop is also capable of generating levels that achieve very high values of fun (see the interval [0.6-1.0]). The Random generator, on the other hand, generates the least diverse levels with the lowest values of fun (1499 values out of 1500 are within the interval [0.1-0.3)). The other two generators (ORE and Patterns) have similar distributions, their levels are more diverse than Random’s, but less than Fi-2Pop’s.

In Fig.5.11 we can observe that the ORE expressive range seems to be biased towards more challenging levels (67.07% of all its levels are within the interval [0.5-1.0])). ORE is also capable of creating a wide variety of levels of different challenge values - its levels are present in each of the histogram bins. The Patterns generator produces less challenging levels than ORE (most Pattern-based levels are of medium challenge) but still its output is quite diverse. Random, on the other hand, is biased towards generating the least challenging levels - 89.87% of all its levels have values within the interval [0.1-0.3). The Random’s output is also the least diverse. Fi-2Pop generate levels of low and medium challenge. The Fi-2Pop expressivity is greater than Random’s, but less than ORE and Patterns.
As seen in Fig.5.12, the ORE expressive range seems to be biased towards more frustrating levels (87.07% of all its levels are within the interval [0.5-1.0] and 53.13% are within the interval [0.7-1.0]), but ORE is also capable of generating levels of medium frustration. Random produces the least frustrating levels of all generators - most of them are of medium frustration. It is worth noting that the levels generated by the Patterns generator are approximately normally distributed (most of the data are in the center - 53.6% of its levels are within the interval [0.4-0.6]). This means that Patterns’ levels are of medium and more-than-medium frustration. Fi-2Pop also generates levels of medium and more-than-medium frustration (73.2% of all its levels lie within the interval [0.3-0.6]).

As can be seen in Fig.5.13, the expressive ranges of all four generators appear to be biased towards more predictable levels. Random produces very predictable levels (all its levels lie in the interval [0.9-1.0]). The other three generators produce levels of high and medium predictability. Of all the four generators, Patterns and ORE produce the most diverse levels in terms of predictability - their levels are scattered across seven and eight bins respectively.

In Fig.5.14 we can observe that three generators (Random, Fi-2Pop and Patterns) are biased towards generating low anxious levels. The ORE, on the other hand, is slightly biased towards creating levels of medium and high anxiety (59.6% of all its levels are within the interval [0.5-1.0]). Of all the four generators, Patterns and ORE produce the most diverse levels in terms of anxiety - their levels are scattered across seven and nine bins respectively.

As can be seen in Fig.5.15, the expressive ranges of all four generators are strongly biased towards very boring levels. All levels generated by three generators (Fi-2Pop, Random and Patterns) lie within the interval [0.9-1.0]. Although ORE also generates very boring levels, it is capable of creating levels of low boredom (five levels lie within the interval [0-0.5]). The reason why most levels generated by four different generators were rated as very boring may be that, as previously suggested (see Section 5.3.2), model of boredom is modeled less accurately than the other models and shows lower ability to differentiate levels.

Considering all six histograms one can conclude that of all four generators ORE is the most expressive. This means that the space of potential levels ORE can create is the most diverse. Random, on the other hand, is the least expressive generator. Patterns and Fi-2Pop are more expressive than Random, but less expressive than ORE. Thanks to its high expressivity ORE is capable of generating levels that provide the best player experience (we can choose levels of
different levels of fun, challenge, frustration, predictability, anxiety and boredom). However, in terms of fun - which is considered the most important emotion - Fi-2Pop has the highest expressivity range, and it is the only generator which is capable of creating levels of very high fun that lie within the interval [0.8-1.0]. The analysis of the expressive ranges confirmed what was discussed in Section 5.3.2. The answer to RQ1 remains the same.
Figure 5.10 Distribution of levels for fun (all generators)

Figure 5.11 Distribution of levels for challenge (all generators)
Figure 5.12 Distribution of levels for frustration (all generators)

Figure 5.13 Distribution of levels for predictability (all generators)
Figure 5.14 Distribution of levels for anxiety (all generators)

Figure 5.15 Distribution of levels for boredom (all generators)
6 CONCLUSIONS AND FUTURE WORK

Four selected procedural level generators for 2D platform games have been evaluated and compared based on the quality of their outputs. An automatic method for evaluation of PLG generators for 2D platformers has been developed and applied for this comparative evaluation of the four PLG generators. The evaluation was based on the quantitative models of player affective states introduced in [18] and described in section 3.4. The automatic method has adapted the already-existing components, described in scientific papers: a 2D platform game Infinite Mario Bros, four procedural level generators (three of them developed within the research community, one developed by us based on the description in the paper), the AI human-like bot (with our improvements) and the quantitative models of player experience. A method based on the simple moving average has been suggested to assess whether the sample size was sufficiently large for the analysis of the results. A limited-scale experiment with a human player was performed in order to verify the adequacy of the automatic bot-based evaluation.

The idea of using an artificial bot as the component of the evaluation of game content quality is not new. However, to the best of our knowledge, this work is the first to use the AI bot which imitates a human player behaviour in evaluating the quality of game content. The main assumption while selecting the generators for comparison was to choose those representing a constructive approach. To have a broader picture, one generator representing a search-based approach was included in the evaluation.

The results of the comparative evaluation show that the Random is the worst generator. It is difficult to decide which generator is the best though - it depends on what we rank first. The Fi-2Pop generates the most fun levels, the ORE generates the most challenging and frustrating, quite fun, least predictable and least boring levels and the Pattern-based generator achieves quite good results taking into account its simple implementation.

The results of the empirical studies are promising. The main advantage of the automatic method is that it is much more efficient compared to the traditional evaluation made by humans (the bot played 6000 levels in less than 10 minutes). What is particularly important, the method seems to be consistent with human assessments. It is worth noticing that two of the compared generators (Fi-2Pop and ORE) were evaluated by human players during the Level Generation Track (a part of the 2010 Mario AI Championship) and Fi-2Pop was ranked above ORE in terms of fun - just the same by the automatic method.
The method was applied for comparative evaluation of four generators (with their default inputs), but this is not its only application. It can be used, for example, to evaluate only one generator, but with different combinations of the generator’s input parameters - every time a different set of input values will be specified, the space of generated content will change. Thus, this method can help level designers find the sets of the generator’s inputs which increase the quality of generated content, test the characteristics of a particular generator and thus better understand the generator itself.

One of the strengths of the automatic evaluation is that it can be used at various phases of generator development process, not only while evaluating the final version. It can serve as a constant feedback while coding, testing and debugging. For example, while coding the Pattern-based generator we were considering three ways of patterns drawing. To decide which way was the best, we applied the automatic method to three different implementations of the Pattern-based generator. That way was chosen for which the highest values of fun were obtained.

There are several possible directions for future work. Since the automatic method is component-based, each of its element can be further improved or replaced with completely new one. For example, the AI bot’s human-likeness can be further increased, but it can also be replaced with totally new implementation of a human-like bot. Another thing to improve could be more accurate models of player affective states (especially boredom). The adequacy of the automatic evaluation method can be more completely validated by performing a broader set of experiments with human players.
References


Appendix A

This appendix includes the complete set of graphs of the maxima and minima of the SMAs (Figs. A.1 - A.24).

Figure A.1 Maxima and minima of SMAs of fun for the levels generated by Fi-2Pop

Figure A.2 Maxima and minima of SMAs of fun for the levels generated by ORE
Figure A.3 Maxima and minima of SMAs of fun for the levels generated by Patterns

Figure A.4 Maxima and minima of SMAs of fun for the levels generated by Random
Figure A.5 Maxima and minima of SMAs of challenge for the levels generated by Fi-2Pop

Figure A.6 Maxima and minima of SMAs of challenge for the levels generated by ORE
Figure A.7 Maxima and minima of SMAs of challenge for the levels generated by Patterns

Figure A.8 Maxima and minima of SMAs of challenge for the levels generated by Random
Figure A.9 Maxima and minima of SMAs of frustration for the levels generated by Fi-2Pop

Figure A.10 Maxima and minima of SMAs of frustration for the levels generated by ORE
Figure A.11 Maxima and minima of SMAs of frustration for the levels generated by Patterns

Figure A.12 Maxima and minima of SMAs of frustration for the levels generated by Random
Figure A.13 Maxima and minima of SMAs of predictability for the levels generated by Fi-2Pop

Figure A.14 Maxima and minima of SMAs of predictability for the levels generated by ORE
Figure A.15 Maxima and minima of SMAs of predictability for the levels generated by Patterns

Figure A.16 Maxima and minima of SMAs of predictability for the levels generated by Random
Figure A.17 Maxima and minima of SMAs of anxiety for the levels generated by Fi-2Pop

Figure A.18 Maxima and minima of SMAs of anxiety for the levels generated by ORE
Figure A.19 Maxima and minima of SMAs of anxiety for the levels generated by Patterns

Figure A.20 Maxima and minima of SMAs of anxiety for the levels generated by Random
Figure A.21 Maxima and minima of SMAs of boredom for the levels generated by Fi-2Pop

Figure A.22 Maxima and minima of SMAs of boredom for the levels generated by ORE
Figure A.23 Maxima and minima of SMAs of boredom for the levels generated by Patterns

Figure A.24 Maxima and minima of SMAs of boredom for the levels generated by Random
Appendix B

Implementation of simple moving average

% "Simple moving average"
% Based on a sequence of n=1500 normalized values assigned to each generator's
% emotion and the window size, the program calculates the moving average
% and then plot a graph. At the beginning of the program the user is asked to
% specify 3 details: the size of the moving window, the name of the generator
% and the type of emotion.

clear

moving_window_size = input('Give the window size for the moving average: ');

% The PLG generators that are evaluated: FI-2Pop, ORE, Patterns, Random
generator_type = input('Choose the name of the generator from: fi, ore, pat, ran... ', 's');

% The 6 emotions that are analysed for each of 4 generators: fun, challenging, frustrating, predictable, anxious, boring
emotion_type = input('Choose the name of the emotion from: fun, chal, frus, pre, anx, bor... ', 's');

% load data (numerical values assigned to 6 emotions for each generator) from txt files
ore = load('levels_ore.txt', '-ascii');
pattern = load('levels_pattern.txt', '-ascii');
random = load('levels_random.txt', '-ascii');
fi = load('levels_fi.txt', '-ascii');

n = length(ore); % n is the size of the sequence of numbers; each generator has the
% same size of the sequence

% ORE data
fun_ore = ore(:,1);
chal_ore = ore(:,2);
frus_ore = ore(:,3);
pre_ore = ore(:,4);
anx_ore = ore(:,5);
bor_ore = ore(:,6);

% Patterns data
fun_pattern = pattern(:,1);
chal_pattern = pattern(:,2);
frus_pattern = pattern(:,3);
pre_pattern = pattern(:,4);
anx_pattern = pattern(:,5);
bor_pattern = pattern(:,6);

% Random data
fun_random = random(:,1);
chal_random = random(:,2);
frus_random = random(:,3);
pre_random = random(:,4);
anx_random = random(:,5);
bor_random = random(:,6);

% FI-2Pop data
fun_fi = fi(:,1);
chal_fi = fi(:,2);
frus_fi = fi(:,3);
pre_fi = fi(:,4);
anx_fi = fi(:,5);
bor_fi = fi(:,6);

% Normalization - for each emotion concatenate all its arrays vertically and then normalize the values
% Fun
fun = cat(1, fun_ore, fun_pattern);
fun = cat(1, fun, fun_random);
fun = cat(1, fun, fun_fi);
norm_fun = (fun - min(fun))/(max(fun) - min(fun));

% Challenge
challenge = cat(1, chal_ore, chal_pattern);
challenge = cat(1, challenge, chal_random);
challenge = cat(1, challenge, chal_fi);
norm_challenge = (challenge - min(challenge))/(max(challenge) - min(challenge));

% Frustration
frustration = cat(1, frus_ore, frus_pattern);
frustration = cat(1, frustration, frus_random);
frustration = cat(1, frustration, frus_fi);
norm_frustration = (frustration - min(frustration))/(max(frustration) - min(frustration));

% Predictability
predictability = cat(1, pre_ore, pre_pattern);
predictability = cat(1, predictability, pre_random);
predictability = cat(1, predictability, pre_fi);
norm_predictability = (predictability - min(predictability))/(max(predictability) - min(predictability));

% Anxiety
anxiety = cat(1, anx_ore, anx_pattern);
anxiety = cat(1, anxiety, anx_random);
anxiety = cat(1, anxiety, anx_fi);
norm_anxiety = (anxiety - min(anxiety))/(max(anxiety) - min(anxiety));

% Boredom
boredom = cat(1, bor_ore, bor_pattern);
boredom = cat(1, boredom, bor_random);
boredom = cat(1, boredom, bor_fi);
norm_boredom = (boredom - min(boredom))/(max(boredom) - min(boredom));

% The end of normalization

% split the normalized arrays into smaller arrays (for each generator)
% ORE normalized data
norm_fun_ore = norm_fun(1:n);
norm_chal_ore = norm_challenge(1:n);
norm_frus_ore = norm_frustration(1:n);
norm_pre_ore = norm_predictability(1:n);
norm_anx_ore = norm_anxiety(1:n);
norm_bor_ore = norm_boredom(1:n);

% Patterns normalized data
norm_fun_pattern = norm_fun(n+1:2*n);
norm_chal_pattern = norm_challenge(n+1:2*n);
norm_frus_pattern = norm_frustration(n+1:2*n);
norm_pre_pattern = norm_predictability(n+1:2*n);
norm_anx_pattern = norm_anxiety(n+1:2*n);
norm_bor_pattern = norm_boredom(n+1:2*n);
% Random normalized data
norm_fun_random = norm_fun(2*n+1:3*n);
norm_chal_random = norm_challenge(2*n+1:3*n);
norm_frus_random = norm_frustration(2*n+1:3*n);
norm_pre_random = norm_predictability(2*n+1:3*n);
norm_anx_random = norm_anxiety(2*n+1:3*n);
norm_bor_random = norm_boredom(2*n+1:3*n);

% FI-2Pop normalized data
norm_fun_fi = norm_fun(3*n+1:4*n);
norm_chal_fi = norm_challenge(3*n+1:4*n);
norm_frus_fi = norm_frustration(3*n+1:4*n);
norm_pre_fi = norm_predictability(3*n+1:4*n);
norm_anx_fi = norm_anxiety(3*n+1:4*n);
norm_bor_fi = norm_boredom(3*n+1:4*n);

% Analysis of the data of the chosen generator and emotion
figure_title_part2 = strcat(', window size=', num2str(moving_window_size));

switch generator_type
    case 'ore'
        switch emotion_type
            case 'fun'
                sequence_of_numbers = norm_fun_ore;
                figure_title_part1 = 'ORE - fun';
            case 'chal'
                sequence_of_numbers = norm_chal_ore;
                figure_title_part1 = 'ORE - challenge';
            case 'frus'
                sequence_of_numbers = norm_frus_ore;
                figure_title_part1 = 'ORE - frustration';
            case 'pre'
                sequence_of_numbers = norm_pre_ore;
                figure_title_part1 = 'ORE - predictability';
            case 'anx'
                sequence_of_numbers = norm_anx_ore;
                figure_title_part1 = 'ORE - anxiety';
            case 'bor'
                sequence_of_numbers = norm_bor_ore;
                figure_title_part1 = 'ORE - boredom';
        end
    case 'pat'
        switch emotion_type
            case 'fun'
                sequence_of_numbers = norm_fun_pattern;
                figure_title_part1 = 'Patterns - fun';
            case 'chal'
                sequence_of_numbers = norm_chal_pattern;
                figure_title_part1 = 'Patterns - challenge';
            case 'frus'
                sequence_of_numbers = norm_frus_pattern;
                figure_title_part1 = 'Patterns - frustration';
            case 'pre'
                sequence_of_numbers = norm_pre_pattern;
                figure_title_part1 = 'Patterns - predictability';
            case 'anx'
                sequence_of_numbers = norm_anx_pattern;
                figure_title_part1 = 'Patterns - anxiety';
            case 'bor'
                sequence_of_numbers = norm_bor_pattern;
                figure_title_part1 = 'Patterns - boredom';
        end
    case 'ran'
end
switch emotion_type
    case 'fun'
        sequence_of_numbers = norm_fun_random;
        figure_title_part1 = 'Random - fun';
    case 'chal'
        sequence_of_numbers = norm_chal_random;
        figure_title_part1 = 'Random - challenge';
    case 'frus'
        sequence_of_numbers = norm_frus_random;
        figure_title_part1 = 'Random - frustration';
    case 'pre'
        sequence_of_numbers = norm_pre_random;
        figure_title_part1 = 'Random - predictability';
    case 'anx'
        sequence_of_numbers = norm_anx_random;
        figure_title_part1 = 'Random - anxiety';
    case 'bor'
        sequence_of_numbers = norm_bor_random;
        figure_title_part1 = 'Random - boredom';
end

switch emotion_type
    case 'fun'
        sequence_of_numbers = norm_fun_fi;
        figure_title_part1 = 'FI-2Pop - fun';
    case 'chal'
        sequence_of_numbers = norm_chal_fi;
        figure_title_part1 = 'FI-2Pop - challenge';
    case 'frus'
        sequence_of_numbers = norm_frus_fi;
        figure_title_part1 = 'FI-2Pop - frustration';
    case 'pre'
        sequence_of_numbers = norm_pre_fi;
        figure_title_part1 = 'FI-2Pop - predictability';
    case 'anx'
        sequence_of_numbers = norm_anx_fi;
        figure_title_part1 = 'FI-2Pop - anxiety';
    case 'bor'
        sequence_of_numbers = norm_bor_fi;
        figure_title_part1 = 'FI-2Pop - boredom';
end

moving_sum = 0; % initial value of the moving sum
moving_avg = [ ];
cirbuff=zeros(1, moving_window_size); % circular buffer
cb_address=1; % circular buffer address

for i = 1:moving_window_size-1 % this loop contains the same instructions as the 'for' below, but without calculating the averages (first we have to reach the size of the moving window)
    moving_sum = moving_sum - cirbuff(cb_address) + sequence_of_numbers(i);
    cirbuff(cb_address) = sequence_of_numbers(i);
    cb_address = mod(cb_address, moving_window_size) + 1;
end

for i = moving_window_size:n
    moving_sum = moving_sum - cirbuff(cb_address) + sequence_of_numbers(i); % shift the moving sum forward, i.e. exclude the first number and include the next one
    cirbuff(cb_address) = sequence_of_numbers(i); % replace the oldest number with the new one
cb_address = mod(cb_address, moving_window_size) + 1;  % increment the cb_address
with modulo-moving_window_size reduction
moving_avg = [moving_avg, moving_sum/moving_window_size];  % calculate and save the
t h average of the moving sum
end

% plot a graph
figure_title = strcat(figure_title_part1, figure_title_part2);  % concatenate two
strings into one - a figure title
figure('name', figure_title);
plot(moving_window_size:n, moving_avg);
xlabel('level number');
ylabel('simple moving average');
h = axis;
axis([h(1) h(2) 0 1]);
grid;
Appendix C

Implementation of maxima and minima of simple moving averages

% "Maximum and minimum values of simple moving averages"
% Based on a sequence of n=1500 normalized values assigned to each generator's
% emotion, the program first calculates the moving averages of different
% lengths (i.e. different sizes of moving window) and then plot a graph with
% both maximum and minimum values of moving averages of different sizes.
% At the beginning of the program the user is asked to specify 2 details:
% the name of the generator and the type of emotion.
% The size of the first moving window is 10 and iteratively increases by 1 for the
% next moving averages. The size of the last moving window is 500.

clear

% The PLG generators that are evaluated: FI-2Pop, ORE, Patterns, Random

generator_type = input('Choose the name of the generator from: fi, ore, pat, ran... ', 's');

% The 6 emotions that are analysed for each of 4 generators: fun, challenging,
% frustrating, predictable, anxious, boring

emotion_type = input('Choose the name of the emotion from: fun, chal, frus, pre, anx, bor... ', 's');

% load data (numerical values assigned to 6 emotions for each generator) from txt
files
ore = load('levels_ore.txt', '-ascii');
pattern = load('levels_pattern.txt', '-ascii');
random = load('levels_random.txt', '-ascii');
fi = load('levels_fi.txt', '-ascii');

n = length(ore); % n is the size of the sequence of numbers; each generator has the
same size of the sequence

% ORE data
fun_ore = ore(:,1);
chal_ore = ore(:,2);
frus_ore = ore(:,3);
pre_ore = ore(:,4);
anx_ore = ore(:,5);
bor_ore = ore(:,6);

% Patterns data
fun_pattern = pattern(:,1);
chal_pattern = pattern(:,2);
frus_pattern = pattern(:,3);
pre_pattern = pattern(:,4);
anx_pattern = pattern(:,5);
b不仅如此，下面的代码片段还实现了计算每个生成器的情感最大值和最小值的功能。通过循环的不同大小的移动窗口，程序首先计算了移动平均值，并画出了最大值和最小值的图。在程序的开始，用户被要求指定2个细节：生成器的名称和情绪类型。移动平均窗口的大小从10开始，逐个增加1，直到最后一个移动平均窗口的大小为500。
% FI-2Pop data
fun_fi = fi(:,1);
chal_fi = fi(:,2);
frus_fi = fi(:,3);
pre_fi = fi(:,4);
anx_fi = fi(:,5);
bor_fi = fi(:,6);

% Normalization - for each emotion concatenate all its arrays vertically and then
% normalize the values
% Fun
fun = cat(1, fun_ore, fun_pattern);
fun = cat(1, fun, fun_random);
fun = cat(1, fun, fun_fi);
norm_fun = (fun - min(fun))/(max(fun) - min(fun));

% Challenge
challenge = cat(1, chal_ore, chal_pattern);
challenge = cat(1, challenge, chal_random);
challenge = cat(1, challenge, chal_fi);
norm_challenge = (challenge - min(challenge))/(max(challenge) - min(challenge));

% Frustration
frustration = cat(1, frus_ore, frus_pattern);
frustration = cat(1, frustration, frus_random);
frustration = cat(1, frustration, frus_fi);
norm_frustration = (frustration - min(frustration))/(max(frustration) - min(frustration));

% Predictability
predictability = cat(1, pre_ore, pre_pattern);
predictability = cat(1, predictability, pre_random);
predictability = cat(1, predictability, pre_fi);
norm_predictability = (predictability - min(predictability))/(max(predictability) - min(predictability));

% Anxiety
anxiety = cat(1, anx_ore, anx_pattern);
anxiety = cat(1, anxiety, anx_random);
anxiety = cat(1, anxiety, anx_fi);
norm_anxiety = (anxiety - min(anxiety))/(max(anxiety) - min(anxiety));

% Boredom
boredom = cat(1, bor_ore, bor_pattern);
boredom = cat(1, boredom, bor_random);
boredom = cat(1, boredom, bor_fi);
norm_boredom = (boredom - min(boredom))/(max(boredom) - min(boredom));

% The end of normalization
% split the normalized arrays into smaller arrays (for each generator)
% ORE normalized data
norm_fun_ore = norm_fun(1:n);
norm_chal_ore = norm_challenge(1:n);
norm_frus_ore = norm_frustration(1:n);
norm_pre_ore = norm_predictability(1:n);
norm_anx_ore = norm_anxiety(1:n);
norm_bor_ore = norm_boredom(1:n);

% Patterns normalized data
norm_fun_pattern = norm_fun(n+1:2*n);
norm_chal_pattern = norm_challenge(n+1:2*n);
norm_frus_pattern = norm_frustration(n+1:2*n);
norm_pre_pattern = norm_predictability(n+1:2*n);
```matlab
norm_anx_pattern = norm_anxiety(n+1:2*n);
norm_bor_pattern = norm_boredom(n+1:2*n);

% Random normalized data
norm_fun_random = norm_fun(2*n+1:3*n);
norm_chal_random = norm_challenge(2*n+1:3*n);
norm_frus_random = norm_frustration(2*n+1:3*n);
norm_pre_random = norm_predictability(2*n+1:3*n);
norm_anx_random = norm_anxiety(2*n+1:3*n);
norm_bor_random = norm_boredom(2*n+1:3*n);

% FI-2Pop normalized data
norm_fun_fi = norm_fun(3*n+1:4*n);
norm_chal_fi = norm_challenge(3*n+1:4*n);
norm_frus_fi = norm_frustration(3*n+1:4*n);
norm_pre_fi = norm_predictability(3*n+1:4*n);
norm_anx_fi = norm_anxiety(3*n+1:4*n);
norm_bor_fi = norm_boredom(3*n+1:4*n);

% Analysis of the data of the chosen generator and emotion
switch generator_type
    case 'ore'
        switch emotion_type
            case 'fun'
                sequence_of_numbers = norm_fun_ore;
                figure_title = 'ORE - fun';
            case 'chal'
                sequence_of_numbers = norm_chal_ore;
                figure_title = 'ORE - challenge';
            case 'frus'
                sequence_of_numbers = norm_frus_ore;
                figure_title = 'ORE - frustration';
            case 'pre'
                sequence_of_numbers = norm_pre_ore;
                figure_title = 'ORE - predictability';
            case 'anx'
                sequence_of_numbers = norm_anx_ore;
                figure_title = 'ORE - anxiety';
            case 'bor'
                sequence_of_numbers = norm_bor_ore;
                figure_title = 'ORE - boredom';
        end
    case 'pat'
        switch emotion_type
            case 'fun'
                sequence_of_numbers = norm_fun_pattern;
                figure_title = 'Patterns - fun';
            case 'chal'
                sequence_of_numbers = norm_chal_pattern;
                figure_title = 'Patterns - challenge';
            case 'frus'
                sequence_of_numbers = norm_frus_pattern;
                figure_title = 'Patterns - frustration';
            case 'pre'
                sequence_of_numbers = norm_pre_pattern;
                figure_title = 'Patterns - predictability';
            case 'anx'
                sequence_of_numbers = norm_anx_pattern;
                figure_title = 'Patterns - anxiety';
            case 'bor'
                sequence_of_numbers = norm_bor_pattern;
                figure_title = 'Patterns - boredom';
        end
end```

case 'ran'
    switch emotion_type
        case 'fun'
            sequence_of_numbers = norm_fun_random;
            figure_title = 'Random - fun';
        case 'chal'
            sequence_of_numbers = norm_chal_random;
            figure_title = 'Random - challenge';
        case 'frus'
            sequence_of_numbers = norm_frus_random;
            figure_title = 'Random - frustration';
        case 'pre'
            sequence_of_numbers = norm_pre_random;
            figure_title = 'Random - predictability';
        case 'anx'
            sequence_of_numbers = norm_anx_random;
            figure_title = 'Random - anxiety';
        case 'bor'
            sequence_of_numbers = norm_bor_random;
            figure_title = 'Random - boredom';
    end

    case 'fi'
        switch emotion_type
            case 'fun'
                sequence_of_numbers = norm_fun_fi;
                figure_title = 'FI-2Pop - fun';
            case 'chal'
                sequence_of_numbers = norm_chal_fi;
                figure_title = 'FI-2Pop - challenge';
            case 'frus'
                sequence_of_numbers = norm_frus_fi;
                figure_title = 'FI-2Pop - frustration';
            case 'pre'
                sequence_of_numbers = norm_pre_fi;
                figure_title = 'FI-2Pop - predictability';
            case 'anx'
                sequence_of_numbers = norm_anx_fi;
                figure_title = 'FI-2Pop - anxiety';
            case 'bor'
                sequence_of_numbers = norm_bor_fi;
                figure_title = 'FI-2Pop - boredom';
        end
    end
end

min_values = [];
max_values = [];
moving_window_size = [];

for k = 10:500  % initial size of a moving window is 10; in each iteration it increases by one
    moving_sum = 0;  % initial value of the moving sum
    moving_avg = [];
    cirbuff = zeros(1,k);  % circular buffer
    cb_address = 1;  % circular buffer address

    for i = 1:k-1  % this for is similar to that one below, but without calculating the averages (first we have to reach the size of the moving window)
        moving_sum = moving_sum - cirbuff(cb_address) + sequence_of_numbers(i);
        cirbuff(cb_address) = sequence_of_numbers(i);
        cb_address = mod(cb_address, k) + 1;
    end
for i = k:n
    moving_sum = moving_sum - cirbuff(cb_address) + sequence_of_numbers(i);  % shift
    the moving sum forward, i.e. exclude the first number and include the next one
    cirbuff(cb_address) = sequence_of_numbers(i);  % replace the oldest number with
    the new one
    cb_address = mod(cb_address, k) + 1;  % increment the cb_address with modulo-k
    reduction
    moving_avg = [moving_avg, moving_sum/k];  % calculate and save the kth average of
    the moving sum
end

    min_values = [min_values, min(moving_avg)];  % save the minimal value of averages
    max_values = [max_values, max(moving_avg)];  % save the maximal value of averages
    moving_window_size = [moving_window_size, k];  % save the length of the moving
    average
end

    % plot a graph
figure('name', figure_title);
plot(moving_window_size, max_values, 'r', moving_window_size, min_values, 'b');  % max
    values - red colour, min - blue colour
    legend('maxima', 'minima', 4);
    xlabel('size of moving window');
    ylabel('maxima and minima of moving averages');
h = axis;
    axis([h(1) h(2) 0 1]);
    grid;
Appendix D

A limited-scale experiment

This appendix includes a description of a limited-scale experiment - its aim was to verify the automatic bot-based evaluation method by confronting it with human player assessments. A limited-scale experiment was performed with one human player.

Experiment design

A Computer Science student, who knew the Super Mario Bros game and played it a lot as a child, took part in the limited-scale experiment. Only one affective state - fun - was taken into consideration. We chose fun because it seems that it is the most desirable and most important emotion.

The experiment consisted of three phases - in each phase a different group of levels was evaluated. In each phase the human player played a pair of levels in both orders. After playing each pair, he had to fill in the questionnaire. There were three options:

- Level 1 was more fun to play
- Level 2 was more fun to play
- I don't know which level was more fun to play.

In each phase the difference between the numerical values of fun within a pair of levels was decreased (we wanted to know how human player's eye is sensitive, if he, for example, is able to decide which level is more fun - that one with 0.982 value or 0.783 assigned by the automatic method). That is why we added the option "I don't know..." in the questionnaire.

The human player was asked to play 28 pairs of levels split into three groups (8, 10 and 10 pairs). In each group levels were selected based on their estimated value of fun and organized in the following way:

1. Group 1 - comparison of extrema - one pair of levels for each procedural level generator.
   Selected levels are the one with the minimum value of fun and the one with the maximum value of fun among all of the 1500 generated levels for each of the four generators.
### Table D.1 Estimated values of Fun for levels from the first group

<table>
<thead>
<tr>
<th>Pair</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fi-2Pop</td>
<td>0.0457</td>
<td>1</td>
</tr>
<tr>
<td>ORE</td>
<td>0.0293</td>
<td>0.6435</td>
</tr>
<tr>
<td>Patterns</td>
<td>0.0498</td>
<td>0.754</td>
</tr>
<tr>
<td>Random</td>
<td>0</td>
<td>0.5182</td>
</tr>
</tbody>
</table>

2. Group 2 - significant difference between levels – five pairs of levels generated by the Fi-2Pop level generator. The difference between estimated values of Fun for two levels is about 0.5.

### Table D.2 Estimated values of Fun for levels from the second group

<table>
<thead>
<tr>
<th>Pair</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fi-2Pop 1</td>
<td>1</td>
<td>0.505</td>
</tr>
<tr>
<td>Fi-2Pop 2</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>Fi-2Pop 3</td>
<td>0.804</td>
<td>0.303</td>
</tr>
<tr>
<td>Fi-2Pop 4</td>
<td>0.703</td>
<td>0.203</td>
</tr>
<tr>
<td>Fi-2Pop 5</td>
<td>0.613</td>
<td>0.113</td>
</tr>
</tbody>
</table>

3. Group 3 – small difference between levels – five pairs of levels generated by the Fi-2Pop level generator. The difference between estimated values of fun for two levels is about 0.2.

### Table D.3 Estimated values of Fun for levels from the third group

<table>
<thead>
<tr>
<th>Pair</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fi-2Pop 1</td>
<td>0.982</td>
<td>0.783</td>
</tr>
<tr>
<td>Fi-2Pop 2</td>
<td>0.875</td>
<td>0.673</td>
</tr>
<tr>
<td>Fi-2Pop 3</td>
<td>0.751</td>
<td>0.550</td>
</tr>
<tr>
<td>Fi-2Pop 4</td>
<td>0.45</td>
<td>0.25</td>
</tr>
<tr>
<td>Fi-2Pop 5</td>
<td>0.345</td>
<td>0.145</td>
</tr>
</tbody>
</table>

Each group of levels had been played twice in separate turns. During the first turn level 1 was played before level 2 while in the next turn it was the opposite. Thanks to this approach we were able to check whether the order of playing levels did not affect player choice and if he did not change his mind.
Experiment results

The results obtained from the above-described experiment showed that decisions made by the human player are very similar to those taken by the automatic evaluation component. 20 out of the 28 player’s answers matched the scoring of the automatic method. In the case of 6 pairs human player was not able to decide, which level was more fun for him and in case of 2 pairs he ranked differently than the automatic method. The majority of uncertain or inconsistent answers were given in case of the group 3, where according to the automatic evaluation method the difference between levels was small.

Analysis

Human player assessment of fun is not precise, so the hesitation in choosing between similar levels is something natural. The uncertainty of human participant even proves that the automatic evaluation method correctly assigned close values to similar levels.

Whereas the conducted verification was a limited-scale experiment with only one human player, it showed that the idea of using the bot-based evaluation method is promising. An interesting future research would be to extend the range of the verification process and to make a comparison between all affective states estimated by the designed method and a bigger group of human players. It would also be interesting to compare a pair of levels with the highest values of a particular emotion (e.g. fun) generated by two different generators (e.g. Fi-2Pop and ORE).