UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Procedural Content Generation in Computer Games

Level Generation for Angry Birds using Genetic Algorithms

Autor
Laura Calle Caraballo

Directores
Juan Julián Merelo Guervós

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

June 16, 2018

ABSTRACT

ACKNOWLEDGMENTS

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ALGORITHMS

ACRONYMS

PCG	Procedural Content Generation
SBPCG	Search-Based Procedural Content Generation
NPC	Non-Playable Character
GUI	Graphic User Interface
EA	Evolutionary Algorithm

Part I

INTRODUCTION

1 MOTIVATION AND OBJECTIVES

1.1 MOTIVATION

It is common for AI researchers to turn to games in general as a testing environment for AI techniques. Some of the earliest problems that AI attempted to solve were checkers and chess in the 50s, even before AI was defined and recognized as a field in Dartmouth Conference in 1956 [8].

What made board games attractive in first place? They have a rather simple set of rules but winning a game could be a challenging task even for a human brain. It is not surprising that soon video games too drew researchers' attention –along with board games–. They offer a wide range of dynamic and competitive elements that resemble real-world problems to some extent.

Of course, this interest works in both ways. Many video games use AI techniques to deliver better experiences, mainly involving Non-Playable Character (NPC) behaviour and Procedural Content Generation (PCG).

Having this relationship between academia and industry in mind, the IEEE Conference on Computational Intelligence and Games started as a symposium in 2005, and as a conference in 2009. It brings professionals from both parts together to discuss the latest advances in AI and Computational Intelligence and how they apply to games.[6]

Among other events, the conference hosts competitions. In 2018, most proposed competitions are centred around playing AI agents for specific games or genres. One of the exceptions is the *3rd Angry Birds Level Generation Competition*. Participants must build computer programs that are able to generate levels for the *Angry Birds* game.

Angry Birds is a mobile game by the Finnish company Rovio Entertainment Corporation[1], first launched in 2009. The game was a huge success in the *AppStore* – being the most downloaded mobile game in history–, has been ported to several other platforms, it has an animated movie and way too many *spin offs* –around seventeen–. In the game, green pigs have stolen the birds' eggs, and they proceed to rescue them. The pigs have built a variety of defensive structures made out of blocks, so the player has to fire the birds from a slingshot – apparently they never learnt to fly– so the structure is destabilized.

However, what is interesting to us is not its wide success, but how heavily the game relies in gravity to create interesting puzzles. The main challenge is to build stable structures that are robust enough to take more than a single shot before crumbling to the ground.

1.2 OBJETIVES

The ultimate objective of this project is to build a program capable of creating levels for *Angry Birds*. This by itself is too vague to convey the desired approach for the project, so we can break it down to the following, more concise, objectives:

- Explore the expressiveness and variability of Search-Based Procedural Content Generation (SBPCG) using evolutionary programs
- Adapt the game to extract data from execution, with the aim of evaluating the levels.
- Produce stable structures under gravity.
- Place other elements on the structures to complete the levels

2 CONTEXT

2.1 THE NECESSITY OF PROCEDURAL CONTENT GENERATION

Computer games are a relatively new form of media whose popularity has been increasing non-stop since they appeared in the 70s. Many challenges have arisen –and will keep doing so– throughout the evolution of the industry, from the early arcades to the most complex modern open-world video games. Developers have come up with all sorts of creative ways to overcome hardware limitations, delivering better graphics and audio. They have pushed the boundaries of the medium by finding new forms of interaction and engaging players with compelling storytelling and original game mechanics. Many of them are related to the fast pace at which the game industry is growing, reaching every passing year to broader audiences that demand a wider range of experiences. Crafting them requires great effort and a high consumption of resources. How do we create a vast amount of content that suits players expectations with lower investment? The answer can be replayability, adaptative content or reduction of designers' workload. All of which can be tackled using Procedural Content Generation (PCG).[15]

Replayability, also referred to as replay value, relies on how interesting is playing a game more than once. It is easy to understand why it would be a desirable feature for both players and developers: from the player's point of view, they can extend the game experience further–past the credits roll. For designers, replay value means their product offers more with less manually crafted content.

Games can also engage players by adapting gameplay elements to each individual player. In a literal sense, this would be impracticable. Instead, users are usually presented with options to adjust to their preferred style. What is more interesting, the game itself can change based on in-game player behaviour. It can regulate its difficulty level to fit the player's learning curve or create content that matches the player's taste.

Both applications above use PCG as a replacement for human designers. However, PCG can be used as a tool to assist developers. It can suggest what might be a base for later development, enhancing human creativity rather than displacing it.

2.2 WHAT IS PROCEDURAL CONTENT GENERATION

The definition of Procedural Content Generation (PCG) has been broadly discussed but there is no agreement. We have plenty of examples of what *is* and what *is not* PCG, but every definition struggles to cover all cases, either being too inclusive or too exclusive. The one we choose here balances well between the two, defining PCG as *the algorithmical creation of game content with limited or indirect user input* [17].

Although PCG often uses AI techniques, this definition does not include all uses of AI in games. We do not consider NPC behaviour or AI playing agents as content, thus they are not PCG either. Aesthetic elements, game rules, levels, items, stories and characters among others are considered content in this definition.

Note that neither computers nor video games are mentioned in the definition. In fact, PCG has its roots in analogical games. This may conflict with the *limited or indirect user input*, but it is reasonable to assume that following a detailed set of instructions –even if it is done by a human– is not *input*. The underlying concepts used much earlier by non-digital games still prevail in modern video games. Using an algorithm to assemble pre-designed pieces is a common technique in tabletop roleplaying guides –where the algorithm usually consists in several dice rolls– such as *Dungeon & Dragons*. It is not surprising that one of the early adaptations of PCG to digital platforms aimed to generate monsters and dungeons for physical games.[12]

2.2.1 *Taxonomy*

There are many PCG methods and it is necessary to look at some traits that characterize and differentiate them from each other: [15]

- *Online/offline*: In online generation, the PCG occurs during the game session, while the user is playing the game. If it is done before the game session or during development, we have offline generation.
- *Necessary/optional*: Procedurally generated content may be part of the essential structure of the game or can be additional to the game experience and can be discarded. In the first one the content is *necessary* and needs to be correct while the latter is *optional*.
- *Degree and dimensions of control*: As any other algorithm, a PCG method can have a number of parameters which affect the output. If it uses random numbers, the seed is one of them.
- *Generic/adaptative*: Adaptative generation takes into account player's behaviour while generic does not. Although there are exceptions, most commercial games choose generic over adaptative.

- *Stochastic/deterministic*: Deterministic PCG will produce the same output given the same input, in contrast to stochastic generation that is not easily replicated.
- *Constructive/generate-and-test*: Generate-and-test produces potentially correct solutions that are tested and adjusted in each iteration before giving the actual output. Constructive methods build partial solutions and add on them.
- *Automatic/mixed authorship*: PCG can be used as an assisting tool for designers, whether the output is used as a base or as an interactive process. Then we talk about mixed authorship, as opposed to the automatic generation where the designer does not take part.

2.2.2 *Search-based Procedural Content Generation*

A special kind of *generate-and-test* approach to PCG is Search-Based Procedural Content Generation (SBPCG), which is usually –but not always– tackled with Evolutionary Algorithms. The problems faced by SBPCG are not very far from those encountered in Evolutionary problems.

The test function does not determine if a solution is valid, but it grades how good the solution is. This is often called *fitness function*. In SBPCG, how to evaluate the quality of a solution has no straight forward answer. It requires formalizing what is fun, exciting or engaging content as a *fitness function* which are usually based in tricky assumptions.

There are three main classes of *fitness functions* in SBPCG[16]:

- *Direct Fitness Function* where certain features are extracted from the generated content and directly mapped to a fitness value. Those features must be easily measurable.
- *Simulation-Based Fitness Function* where an agent plays throw some part of the game that involves the generated content. The fitness is calculated using features from the agent's gameplay.
- *Interactive Fitness Function* where the fitness value is obtained from the player, whether it is explicitly by asking them or implicitly by measuring certain responses to the game.

2.3 EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are optimization methods inspired by biological evolution. They simulate the optimization carried out by nature with genetic inheritance and natural selection, even borrowing the vocabulary from genetics. The following components are necessary to apply an evolutionary algorithm to an specific problem:

- A genetic representation of potential solutions, which are *individual* or *chromosome*. Those are formed by smaller units called *features* or *genes* that represents a certain character of the solution.
- an initialization method for the population of individuals.
- a *fitness function*, which evaluates each individual and gives it a *fitness value*. It plays the role of the environment, determining how likely an individual will survive.
- genetic operators, such as crossover, mutation, selection and replacement.
- values for all its parameters, i.e. population size, probabilities of applying certain operators, number of generations, etc.

2.3.1 *Genetic operators*

2.3.1.1 *Crossover*

This operator takes two individuals and produces an offspring, one or more individuals who inherit certain traits from their parents. It is also known as *recombination*. The genes of both parents are combined resulting in new individuals, hopefully with better fitness. However, that is not always the case and some of those new solutions can be infeasible. Recombination is equivalent to sexual reproduction in biological evolution.

2.3.1.2 *Mutation*

While the rest of the operators focus on exploitation of promising solutions, mutation—along with initialization— usually generates diversity. This is key for escaping local optima, balancing between exploration and exploitation. It is applied after crossover and modifies with a certain probability some genes of the offspring.

2.3.1.3 *Selection*

Selection operator determines which individuals of a given generation will mate and produce offspring. It is usually based in the fitness value of the individuals: better solutions are more likely to breed than others. The output of selection operator is the input for crossover.

2.3.1.4 *Replacement*

In nature, better fit individuals are more likely to survive. In evolutionary algorithms, the replacement operator decides which individuals make it to the next generation. In some strategies, offspring completely replace the previous generation, while others maintain the best individuals through several generations.

Part II

THE STATE OF THE ART

3 PREVIOUS ENTRIES

Let's have a look at some participants in previous editions of the Angry Birds Level Generation Competition.

3.1 CONSTRUCTIVE APPROACH

In the two solutions proposed by Matthew Stephenson and Jochen Renz [14] [13] the structures displayed on the level are constructed from the top down, in several phases.

First, they build an structure recursively, each row composed of a single type of block (with a fixed rotation). The likelihood of selecting a certain block is given by a probability table. Then the blocks are placed using a tree structure, where the first selected block is the peak and blocks underneath it are split into subsets that support the previous row with one, two or three blocks. This ensures local stability, but not global stability, which is tested once the whole structure is completed. The second solution also tries to add variety by replacing some blocks with others of the same height.

After that, other objects (pigs and TNT) are placed. Potential positions for those objects are recorded, first trying to place them in the centre of each block, then right above the edges of it, checking if there is enough space. Available positions are ranked based on structural protection, dispersion, etc. and then filled with the desired number of objects.

The second solution also selects material —which can be stone, ice or wood— based on trajectory analysis, clustering, row grouping, structure grouping or randomly. Weak points are set to stone material and the rest using one of the previous strategies. Trajectory analysis based strategy sets to the same material all blocks in the trajectory of a shot aimed for a particular pig. Clustering strategy takes a random block, sets its material and propagates to the surrounding blocks that have not been assigned yet. Row grouping and structure blocking apply the same material to a whole row or structure respectively.

Materials also determine which kind of birds will be used in the level, since some birds are more efficient against certain materials than others. The number of birds available for the player to solve the level is estimated using AI agents designed for a different competition. If the AI agents are unable to solve the level, it is discarded. The lowest number of birds used by the agents is the chosen for the level.

The probability table was tuned using search-based optimization methods.

3.2 SEARCH-BASED APPROACH

Lucas Ferreira and Claudio Toledo[3] present a solution based on Search-Based Procedural Content Generation (SBPCG) using a genetic algorithm and a game clone developed in Unity Engine to evaluate the levels.

In the genetic algorithm individuals correspond to levels, each represented by an array of columns. Each column is a sequence of blocks, pigs and predefined composed blocks, using an identification integer. This representation also includes the distances between different columns.

The population is initialized randomly following a probability table which defines the likelihood of a certain element to be placed in a certain position inside a column.

For the evaluation, levels are executed in the game which stores data about the simulation. The fitness function is described as:

$$f_{ind} = \frac{1}{3}\left(\frac{1}{n}\sum_{n-1}^{i=0} v_i + \frac{\sqrt{(|b|-B)^2}}{Max_b - B} + \frac{1}{1-|p|}\right)$$

where v_i is the average magnitude of the velocity vector for block i, $|b|$ is the total of blocks in the level and $|p|$ the amount of pigs. The rest are parameters: B the desired amount of blocks and Max_b the maximum number of elements.

The recombination process uses an Uniform Crossover where the new individual is generated selecting for each position a column from one of the parents which occupies the same position. When the parents are not the same length, the remaining columns are selected with a 50% of probability. Mutation simply changes with a certain probability each element of the individual randomly (either being a column element or the distance between columns).

3.2.1 *Open Source Simulator*

The fitness evaluation requires a simulation to test the behaviour of the levels under the game's physics. Angry Birds is not open source software so the code is not available, so a game clone was developed to fill this gap, using the Unity Engine.

Levels are described in XML files which the game takes as an input. They are parsed to run the simulation which then generates new XML files as an output. Those files contain information about the execution of a certain level such as average velocity of each element, the amount of collisions and the final rotation.

4 CONSTRAINED OPTIMIZATION WITH EAS

Section 2.3 presents an overview of what is and which are the main components of an EA. Here look in more detail to fitness functions and how to direct the search.

In evolutionary optimization the fitness function (being $f(x)$) is defined in a search space that contains a feasible region ($\mathcal{F}$), that can be limited by constraints (being $g_j(x)$).

$$f(x), \qquad x = (x_1, \ldots, x_n) \in \Re^n$$

$$\mathcal{F} = \{x \in \Re^n | g_j(x) \leq 0 \quad \forall j \in \{1, \ldots, m\}\}$$

$$g_j(x), j \in \{1, \ldots, m\}$$

A constrained problem can be turned into an unconstrained one with the inclusion of a penalty function as shown below. The penalty function can be controlled by parameters or penalty coefficients (r_g) [10], being ϕ the real-valued function in charge of applying the penalty.

$$\psi(x) = f(x) + r_g \phi(g_j(x); \quad j = 1, \ldots, m)$$

Using this parameter —that can be dynamically adjusted— the objective function can range from purely dominated by the fitness function to dominated by the penalty function. Canonical evolution strategies would *overpenalize*, meaning that infeasible solutions would be discarded immediately. On the other hand, if the objective function *underpenalizes*, the solution evolved by the algorithm may be belong to the infeasible region. It is recommended that the value for the penalty coefficient is balanced during the evolution since the optimal value is dependant on the problem and the population.

Another approach to constrained optimization is to transform it into multiobjective optimization[11], being one of the objectives to minimize the penalty function. Penalty functions usually guide the algorithm, producing a bias in the search. Multiobjective optimization does not produce such bias but it will spend more time exploring infeasible regions.

Part III

METHODOLOGY

Chapter 5 discusses the general workflow in this project and its phases, later described in detail in Chapters 6, 7 and 8

5 METHODOLOGY

In this chapter we discuss the tools used for this project as well as the workflow followed in the development.

Agile development is an iterative and incremental method. It focuses on delivering working software in short iterations which makes it a perfect approach to undertake this project considering its time limitations and goals.

The defined phases for this work are:

- Modification of the existing open source simulation. The goal of the currently available software is not to simulate *Angry Birds* levels but to allow the user play the game using previously generated ones.
- Determine the different elements of the evolutionary algorithm, such as fitness, representation and operators. This requires a previous study of feasible options.
- Selecting or developing a framework for the evolutionary algorithm. There are several frameworks available but not all of them will satisfy our requirements.
- Experiments and results.

Those phases are not sequential as they overlap. Obviously, there is a logical order for them and therefore there was a major focus on the first phase early in the development, moving this focus to the following part as the project progressed.

The overlapping of the other three tasks is greater, providing a better environment for agile development. Each iteration expanded for about two weeks, in which an increment was delivered.

First increments consisted in modifications to the simulation and a study of different representations based on previous work on the same topic or related. After that, development started with integration of the simulation and the evolutionary program as well as a comparison of different frameworks. The framework and genetic operators were developed in parallel along with tests.

The increments also included experiments as they were based on hypotheses from previous results.

The tools involved in the development of this project were:

- Unity Engine (C#), for the game adaptation.
- Python standard library, for the evolutionary algorithm.
- LaTeX, for this document.

6 MODIFICATIONS TO THE OPEN SOURCE SIMULATOR

As a part of their solution, Lucas Ferreira and Claudio Toledo[3] developed an open source simulation for the *Angry Birds* game. Since that implementation is available on Github, it will be used as a starting point for this part of project.

6.1 PRE-EXISTING GAME OVERVIEW

The game is implemented using the Unity Engine and its code is mainly written in C#. Levels are not part of the game itself and instead they are parsed from XML files contained in an specific folder in the game hierarchy which corresponds to *Assets/StreamingAssets/Levels* directory. Each level is described in a different file, similar to listing 6.1

```xml
<?xml version="1.0" encoding="utf-16"?>
<Level width="2">
 <Camera x="0" y="0" minWidth="20" maxWidth="25">
 <Birds>
  <Bird type="BirdRed"/>
  <Bird type="BirdBlack"/>
 </Birds>
 <Slingshot x="-7" y="-2.5">
 <GameObjects>
  <Block type="RectMedium" material="wood" x="2.78" y="-3.18" rotation=
      "0"/>
  <Block type="RectFat" material="wood" x="2.75" y="-2.66" rotation="
      90.00001"/>
  <Block type="SquareHole" material="stone" x="2.04" y="-2.3" rotation=
      "0"/>
  <Block type="RectSmall" material="wood" x="4.05" y="-3.08" rotation="
      0"/>
  <Pig type="BasicSmall" material="" x="2.74" y="-1.98" rotation="0"/>
 </GameObjects>
</Level>
```

Listing 6.1: Sample level input to show format

The XML file sets a number of parameters for the level, some of them are key to level generation and others that will be ignored by the generator—and will be fixed, as it will be stated in following chapters— but still needed for the simulation:

- Level width
- Camera position

- List of birds, each one with its type (for our objective of making stable structures this will remain constant, although it is important from a game designer's point of view)
- Slingshot position
- Game objects list, this one is the most important for level generation. It can contain blocks, pigs or TNT.

Those elements are then used to build a level, placing the objects in the specified coordinates (note that 0,0 correspond to the centre of the screen, being the ground at $y = -3.5$ in game world units).

Although the simulation described in [3] does offer an output and it is automated to run through all levels, the implementation provided does not. However, some of these features can be found in the source code.

6.2 CHANGES IN THE ADAPTATION

Considering the functionality of the existing code, there are a few issues preventing us to use the game as a simulation:

- (Issue #1) Human interaction is needed to proceed: there is a main title screen and a level selection.
- (Issue #2) A level will only be finished if the user skips it or there are no pigs left. If there were not pigs in the level description, the game may have inconsistent behaviour, displaying the *Level cleared* banner or crashing.
- (Issue #3) Once a level has been skipped or cleared, data from execution is not stored.
- (Issue #4) Skipping the last level results in loading the first level again.

It is also worth mentioning that the simulation was intended to run under a Linux distribution. However, this was not possible probably due to issues with the Unity Engine compiler for Unix. Those were presumably fixed in later releases but it is not working on this project, originally created in a much older version although the code itself does not seem to be platform dependant.

6.2.1 *Output XML*

Modifying the simulation to write data output—Issue #3— is fairly simple since the functions were already in the code. However, the data that was being stored was not enough for its purpose. Output

XML level was a valid input XML level, which misses some key information about the execution and in this case there were not much interest in obtaining a new valid level. The desired output would look like listing 6.2 (note that encoding was also changed from original):

```xml
<?xml version="1.0" encoding="utf-8"?>
<Level width="2">
 <Camera x="0" y="0" minWidth="20" maxWidth="25" />
 <Birds>
  <Bird type="BirdRed" />
 </Birds>
 <Slingshot x="-5" y="-2.5" />
 <GameObjects>
  <Block type="RectSmall" material="wood" x="3.78" y="-3.294594"
      rotation="90" id="0" aVelocity="6.974828E-08" />
  <Block type="SquareTiny" material="wood" x="3.78" y="-2.551131"
      rotation="90" id="1" aVelocity="1.464643E-07" />
 </GameObjects>
</Level>
```

Listing 6.2: Sample level output to show format

To calculate average magnitude of velocity for each block, it should be measured at regular time intervals; but, since this and other processes seem to be frame rate dependant, this was achieved by forcing the game to a constant frame rate. Blocks that were destroyed on execution are not present in the output.

Encoding was changed from `utf-16` (C# XML reader/writer default) to `utf-8` (only format available for Python's Standard Library XML reader).

6.2.2 *Automatization*

The next logic step is getting rid of interaction. The simulation we are interested in does not require a player (human or AI agent) to play the level since the goal is to know how the level behaves under the game's physics that do not need —and they do not— to match real physics.

A level will run for a certain amount of time, and then continue to the next one, skipping the main menu and level selection. When the simulation reaches the last level it will end the execution.

Unity Engine projects are usually divided into scenes. This one has a different scene for the main menu, level selection, loading screen, levels and failure screen. Removing scenes from the screen flow causes side effects such as the first level crashing so instead, the same functions invoked by the GUI are called after initialization takes place in both scenes (main menu and level selection) before the first level. This solves interaction with menus (Issue #1).

In order to skip levels and keeping in mind the velocity of the blocks is already being tracked, the game will load the following scene —the next level— once all blocks reach velocity 0, which means the blocks have stabilized. Just in case this condition is never met, there is also a time limit of $10s$ for each level (Issue #2).

The remaining issue (Issue #4) is solved just by checking how many levels have been loaded and the index of the currently playing level.

7 EVOLUTIONARY ALGORITHM

In this chapter, level representation is discussed, as well as several genetic operators that will be later used to run the experiments.

7.1 FITNESS FUNCTION

The most challenging part of the level generation process is to create stable structures. As obvious as it is, the main feature of a stable level is that it is not in motion. So it seems reasonable to evaluate their stillness as opposed to their speed.

$$fitness_{ind} = \frac{1}{|V|}\sum_{i=0}^{b} V_i + P_{broken} \cdot (b - |V|)$$

As mentioned in Chapter 6 the game output provides the average magnitude of velocity for each block. This vector is noted as V, with $|V|$ being the length of the vector. The number of blocks in an individual is b and it can differ from the length of V since broken blocks are not tracked. The number of broken blocks is $b - |V|$ and it is multiplied by a penalization factor, since a level whose blocks break without user interaction would not be considered valid. This happens when a block free falls from a certain height or collides with a falling object. P_{broken} is set to 100 since objects in a level do not usually reach that velocity, therefore it will separate not valid levels from potentially good ones.

Not all levels are evaluated using a simulation, since it is a costly process. Although we may know that a given level will perform poorly in the simulation, removing them from the population will cause a high loss of diversity. For that matter, before testing a level, there are some indicators that a level would not be suitable and can be skipped in the simulation. Those are its distance to the ground and the number of blocks that overlap.

If the lowest object is not close to the ground is very likely that all blocks will break in the impact. Levels that have all their blocks higher than a certain threshold will not be simulated in the game. The threshold used is 0.1 in game units and the penalty applied to the fitness value is 10:

$$f_{distance} = \begin{cases} P_{distnace} \cdot D_{lowest}, & \text{if } D_{lowest} > threshold \\ 0, & \text{otherwise} \end{cases}$$

The other measure is the number of overlapping blocks. The separating axis theorem —explained in appendix A— determines if two convex shapes intersect. It is commonly used in game development for detecting collisions. A level with blocks that occupy the same space is not likely to be stable, as the Unity Engine will solve the issue moving the blocks until there is no collision. Since Unity Engine is not open source and there is no documentation on how exactly those collisions are solved, we assume that a precise prediction of the positions of the blocks is not possible and therefore the fitness value obtained could be inaccurate. A penalization is applied and the level is not simulated in game. In this case it is $f_{overlapping} = P_{overlapping} \cdot N_{overlapping}$ where the first factor is a penalty set to 10 and the second is the number of blocks that overlap with each other.

If both $f_{distance}$ and $f_{overlapping}$ are 0 then the level is suitable for the simulation and fitness is calculated as $fitness_{ind}$. As discussed in Chapter 4, this would be considered *overpenalization* but exploring infeasible regions is a serious overhead that we need to minimize. On the other hand, levels with multiple blocks broken during the simulation are not feasible either but running the simulation is necessary. In this case penalization does not prevent the region to be explored.

7.2 LEVEL REPRESENTATION

Previous approaches to this problem (studied in Chapter 3) provide fairly constrained output. The constructive method presented creates pyramid-like structures and, even though there is a variety of levels, the method is highly specialized in this kind of levels. The search-based approach only produces tower structures adding some variety by having pre-built composed blocks.

Since one of the objectives of this project is to explore the expressiveness and variability of SBPCG, it seems reasonable to use a flexible representation.

One of the early considered options, was the use of grammars. In this category we can find Grammatical Evolution[7] and Generative Grammatical Encodings[5]. The first one focuses on optimizing how a grammar is applied, which production rules expand and in which order. This means there is a grammar previous to the evolution process. In our specific case, designing a grammar that solves the problem is a deeply complex problem itself in addition to the optimization process. The generative grammatical encodings use a range of operations that are combined in the evolution process to form a grammar. Designing those operations is much easier which also allows greater variability in the output.

Although those approaches are promising, the scope of this project and its time limitations made these options little feasible. Instead, we will try a less directed search than previous solutions while keeping a simple representation.

7.2.1 *Gene representation*

Individuals are composed by a list of blocks, each of them being a gene. Special pieces such as platforms, TNT boxes or pigs are not considered in this phase. The building blocks for the game have several attributes that characterize them:

- Type: there are eight regular blocks that can be placed in the level with distinct shapes or sizes. Represented as an integer between 0 and 7.
- Position: coordinates x and y of the centre of the block in game units
- Rotation: rotation of the block in degrees. Here are considered four different rotations, 0°, 45°, 90° or 135° represented as integers between 0 and 3.

Using this representation a gene will be formed by two integers and two floating point numbers. The position of the corners of the block is frequently required, so it is stored along with those attributes even though it can be calculated using the size, the position and rotation of the block.

There are three types of materials in the game, which determine the durability of the block. However, this does not affect their stability, so it will remain constant for now as *wood* material.

7.2.2 *Chromosome representation*

Individuals are a collection of genes, in the same way a level is a collection of building blocks. The number of blocks is variable and the order in which they are listed is not important.

As previously stated, only promising individuals are tested in-game while those who do not, are penalized. This penalty is stored, separately from the fitness value for the individual. The reason for this is that it may change over generations. The goal of the penalization is to maintain fitness value of not tested level above —it is a minimization problem— the in-game tested levels, so the starting point for fitness of such individuals is the worst in-game score.

This penalization is calculated in base to the distance of the lowest block to the ground, that can be easily obtained, and the number of blocks that collide. This requires a bit more of computation, so it will be stored and set in the initialization of the individual. When a gene is modified, the number of overlapping blocks is recalculated for that specific change.

Considering all of the above, the chromosome object is composed by:

- a list of genes
- a fitness value
- a penalty (set to -1 for in-game evaluated levels)
- number of overlapping blocks (calculated)

7.3 GENETIC OPERATORS

7.3.1 *Initialization*

Initialization is done randomly, with each individual having a random number of genes. Those genes can be initialized using several methods.

- Random: selects a random number for each attribute of the gene
- No Overlapping: also selects a random number but the gene is only added to the chromosome if it does not overlap with an already existing gene
- Discrete: selects a random number for type and rotation, but the position must be multiple of the dimensions of the smallest block (blocks will be aligned)
- Discrete without overlapping: it combines the second and third initialization method

7.3.2 *Selection*

Parents are selected using tournaments. Two individuals are chosen from the population and the best of them will be a parent in this generation. This is repeated until a certain percentage of pairs have been reached. It is important to note that individuals chosen are not removed from the population and therefore they can appear several times in the list of parents.

7.3.3 *Crossover*

Once the parents have been selected, we implement two different methods of combination

7.3.3.1 *Sample Crossover*

This operator gives a single individual per parent pair. It takes all genes from both parents —excluding genes that are repeated— and randomly takes a number of them to create the new individual. The number of blocks is the minimum between the maximum number of blocks allowed, the mean of the two parent individuals and the number of distinct genes.

7.3.3.2 *Common Blocks*

In this case, the operator produces two individuals. The common genes to both parents are passed on to both children. The remaining genes are randomly distributed to each child, half to one and half to the other.

7.3.4 *Mutation*

There are four different mutations:

- Type: type is represented as an integer, so it adds or subtracts one to the current value
- Rotation: similarly to type mutation
- Position X: a real value between 0 and 1 —excluding 0— is added or subtracted to the value of the position X
- Position Y: same as position X mutation, for position Y

7.3.5 *Replacement*

A the new generation is selected using an elitist strategy. Best individuals in both the old population and their offspring pass on the next generation, maintaining the size of the population.

8 EVOLUTIONARY ALGORITHM FRAMEWORK

The selected language for this project was Python. It has a vast Standard Library and is well documented. There are a number of evolutionary programming frameworks for Python. We considered two in particular for this project:

- DEAP (Distributed Evolutionary Algorithms in Python)[4]
- PyEvolve[9]

Since the game simulation will be used as a part of the fitness function, there is another aspect to keep in mind. The simulation takes several seconds to start, but much less between levels. When evaluating individuals in a evolutionary algorithm, it would be more efficient to avoid this overhead by simulating the whole population in a single execution than launching a a simulation for each individual. Then it seems compulsory that the framework used would allow this behaviour.

The information that describes a level can be too complex to have a binary representation as pure genetic algorithms suggest, so the framework should be flexible enough to support complex data structures.

Other features that would be desirable for a framework to have in order to be suitable for our goals are readability, ease of use and efficiency.

Looking at the table 8.1, both frameworks offer what this project is looking for. However, it is important to note exactly how each one provides said feature:

- Flexible data structures: both allow them, but the user has to provide the genetic operators. That is reasonable and expected.

	DEAP	**PyEvolve**
Flexible data structures	Yes	Yes
Customizable evaluation	Yes	Yes
Other considerations	Well documented, actively maintained	Less flexible

Table 8.1: Framework comparison

Using DEAP, the new data structure only needs to be registered in the *Creator* class, but PyEvolve requires *GenomeBase*—its base module for genomes— to be modified in order to use custom data types.

- Customizable fitness function calls: With DEAP you can register a function to evaluate individuals, but those have to take a single individual as a parameter. If we were to evaluate all of them at once, we will have to give up the algorithms the framework provides, executing each operator and managing parameters ourselves. In PyEvolve, although it is possible, it is rather inconvenient since it means , working around the *GPopulation* class.

The problem tackled in this project requires high customization. Both framework are fairly flexible and easy to use when presented with a pure numerical optimization problem, but that is not the case. Taking into account the time limitations for this project, productivity is really important. It is not worth adapting these frameworks, considering how little we gain from their use, even though it would not take a great amount of time. Instead, a new one was implemented, providing all the flexibility we were looking for.

8.1 IMPLEMENTED EVOLUTIONARY FRAMEWORK

The framework needed is not complex, but it covers all the features required. The most important one is a fitness function that executes all the individuals at once. This does not mean parallelism, in fact the steps to take cannot be concurrent. First the penalization function is applied to all individuals and XML is generated for those not penalized. Only after that, the simulation is run. The whole evolution process is summarized in algorithm 1.

It follows the basic structure of a generic genetic algorithm, with the difference in the fitness function that was mentioned before. The terminating condition can be a set number of generations, or until a certain fitness value for the best individual is reached. In our case this threshold is set to 0.01. Another stop criteria is several generations without any change in the population, which means that for several generations any new individual was better than the worst previous one. In this situation, we assume that a local minimum has been reached.

All operators described in Chapter 7 are implemented as a part of the framework. The chromosome and the gene are classes with the data members specified. The fitness value (once data from the simulation has been collected) and the penalty function are responsibility of the chromosome. Those two clases along with the genetic operators have been tested using Python Standard Library *unittest*.

Algorithm 1 : Evolution

```
begin
    population = Initialization(size, method);
    suitable = EvaluatePenalization(population);
    velocity = Simulation(suitable);
    EvaluateFitness(suitable, velocity);
    for $gen = 0 \to nGenerations$ do
        parents = Selection(population);
        children = Crossover(parents);
        Mutation(children);
        suitable = EvaluatePenalization(children);
        velocity = Simulation(suitable);
        EvaluateFitness(suitable, velocity);
        Update Global Penalization;
        population = Replacement(population, children);
    end
    return Best individual
end
```

The framework also allows logging data from evolution. The format chosen for this purpose was *JSON* since its integration with Python is straightforward and it is a human readable format, which will make easy to interpret the results. For convenience, the parameters to the algorithm are also in *JSON* format. The logs contains the configuration *JSON* object, the time of execution and the execution itself: a dictionary with all generations, each of them containing the value of the best fit individual, the fitness average of the population and the worst fitness value. It also includes the Shannon's entropy for the population.

9 EXPERIMENTATION AND RESULTS

As stated in Chapter 5, each experiment was based on an hypothesis formulated after the observation of previous results. The structure of this chapter reflects this methodology.

Four experiments have been carried out, with more than 15 executions per experiment. Table 9.1 shows an overview of the results.

	E1	E2	E3	E4
Time (h)	0.89	1.002	1.76	5.03 (h)
G	100.0	155.087	76.625	365.929
Best	61.334	110.66	0.0015	0.0018
Avg	383.701	327.547	0.54	0.203
Worst	510.515	367.895	0.828	0.2997

Table 9.1: Summary of results, G: number of generations

9.1 FIRST EXPERIMENT

9.1.1 *Hypothesis*

The premise of this experiment is that our basic EA should be able to minimize the movement of the blocks placed on the level and the flexibility of the representation should allow variety in the structures. This may be optimistic, but we need an estimation as a starting point, and this experiment will serve the purpose.

9.1.2 *Execution*

The EA in this experiments uses:

- initialization with the discrete method described in section 7.3.1
- basic sample crossover in section 7.3.3.1
- all four mutations (section 7.3.4)
- elitist replacement (section 7.3.5)
- selection using tournaments (section 7.3.2)

The parameters are:

- Population size: 100
- Number of generations: 100
- Percentage of parents: 0.5
- Percentage of type mutations: 0.5
- Percentage of rotation mutations: 0.5
- Percentage of axis x mutations: 0.5
- Percentage of axis y mutations: 1

9.1.3 *Results*

The results suggest that the hypothesis was not correct. The average best solution has a fitness value of 61.334 (as shown in Table 9.1) which indicates that probably most levels have blocks falling (and even breaking) when loaded. The standard deviation of this measure is 133.0209 which implies that while some executions performed poorly, some others may be good. Some of the levels with lowest fitness value can be found in section A.2.1. Even those levels have blocks that break after loading so they would not be valid. However, we can tell there is variety in the structures, since they clearly differ from each other.

In the experiment the only termination condition was reaching the maximum number of generations. However, looking at the results, it seems that the execution ended before the population stabilised or converged. Since mutation percentages are high it is normal that convergence where every single individual is the same one, is not reached. However, it could be possible that the population is stable, where every child has a grater fitness value than its parents, therefore no new members are allowed. If there is no new individuals and the population is completely *stuck*, the fitness value of the worst individual should be the same over several generations. In Figure 9.1 we can see that is not the case in average. Although some populations do remain the same for several generations close to the maximum, most of them do not.

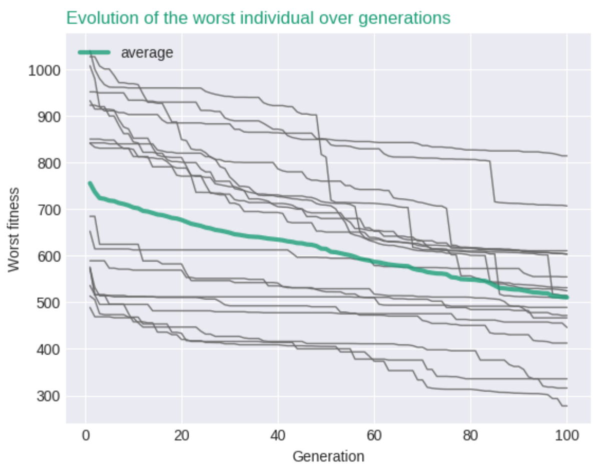

Figure 9.1: In grey, different executions of the first experiment 1

9.2 SECOND EXPERIMENT

9.2.1 *Hypothesis*

Most executions from the previous experiment reached the maximum number of generations without stabilizing or converging. This means the EA may need a larger number of generations to fully evolve a solution.

9.2.2 *Execution*

The set of operators is the same as the previous one. The parameters remain unchanged except for:

- Number of generations: 1000

We also added two stop conditions: 10 generations without changes (stable population) or best fitness value below 0.01.

9.3 RESULTS

A sample of four of the levels generated can be found in the Appendix (A.2.2). Three of the levels were selected for having the lowest fitness value, but Figure A.7 was visually interesting despite having a high fitness value due to a couple blocks broken on loading.

Although best levels obtained with this experiment are better than those evolved in the first experiment, the bad solutions have a really high fitness value. In table 9.1, we can see that average fitness of levels produced with this version of the EA is worse than the ones generated in the first experiment. It suggest that populations can be stuck for many generations before making any type of improvement.

Any of the generated levels have a fitness below 0.01 or reached maximum number of generations, which means the termination criteria that stopped the evolution was that the population was stable, without any new individuals added for 10 generations.

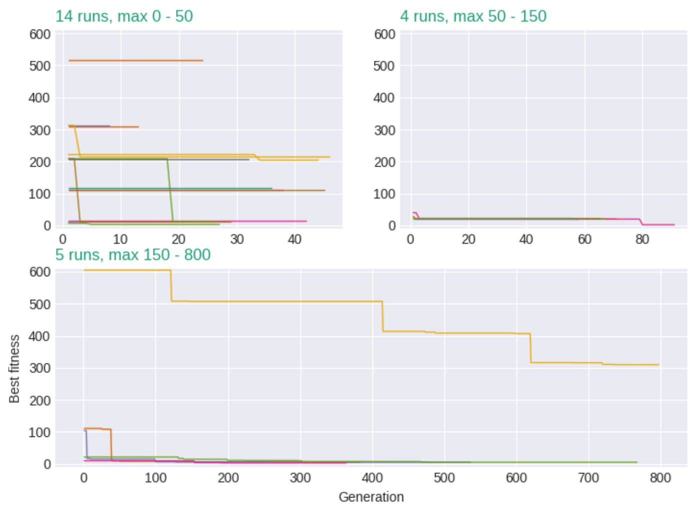

Figure 9.2: Best individual evolution for all executions, grouped by number of generations

Figure 9.2 represents the evolution of the best individual of each execution. Most of them have no more than 50 generations, therefore the hypothesis for this experiment is not correct. Short evolutions show that the best individual at initialization is very similar to the last one. Slight improvements may be achieved by small mutations but it seems difficult for new generations to outperform previous ones. We can appreciate that significant improvements are most common in those executions with a poor initial population. Even the ones with several hundreds of generations struggle to improve th intial population.

9.4 THIRD EXPERIMENT

9.4.1 *Hypothesis*

The previous experiment proved that the problem is not with the number of generations. Then it is more likely that there this EA is biased towards exploration rather than exploitation. The genetic operators are failing to create new individuals that inherit good traits from their parents. A new crossover operator could shift the focus to exploitation.

9.4.2 *Execution*

This time the change introduced is in the crossover operator used. It was described in section 7.3.3.2. The rest of the operators as well as the termination criteria are kept the same.

9.4.3 *Results*

Table 9.1 shows that the results have radically improved as the average fitness of the best solutions drops to 0.0015, a decrease of almost 100%. Additionally, it took less generations in general to reach those results. However, executions took longer on average, which makes sense given that a greater number of individuals would have been simulated. The average fitness of the population and the worst individual have similar values now, which suggest that in most executions the population did converge. The Appendix section A.2.4 contains four levels generated with this EA. The selected levels are not necessarily the ones with less fitness value since with such small values the difference in stability is imperceptible.

The levels are stable and the blocks do not fall when loading the level, but they could arguably be considered structures, since most of them consist in a few blocks displayed on the floor. The average amount of blocks is 6.26, which is really close to the minimum amount of blocks allowed. However, given the proposed fitness function, it is completely logical that the evolution leads to this kind of arrangements. The more object placed on the level, the more likely the individual is to not meet the requirements imposed by the constraints. It also makes sense to place objects near the ground, instead of one on top of the other.

9.5 FOURTH EXPERIMENT

9.5.1 *Hypothesis*

The previous EA generates levels with a number of blocks that tends to the minimum of blocks allowed which is 5. Such a small number of elements does not create interesting structures. A higher number of blocks may lead to more appealing levels.

9.5.2 *Execution*

The only parameter that was changed for this experiment is the minimum number of blocks from 5 to 10.

9.5.3 *Results*

The first thing to notice in these experiment(Table 9.1) is the increase of the average execution time: it went from 1.76 hours in the third experiment to 5.03 hours in this one. The time spent running the simulation for each population in this experiments and in the previous one should be similar. However, the number of executions drastically increased too. There is no doubt that placing at least ten objects is more difficult than placing 5. The average best fitness value slightly raised, while the average and worst values are lower. This suggest that the latest generations of this EA are less diverse than those from the third experiment.

Examples of the levels can be found in the Appendix section A.2.4. Even though levels are a little less stable this time, they are more interesting, which does not mean they could be considered fun levels, or levels at all.

Part IV

CONCLUSION

APPENDIX A

A.1 SEPARATING AXIS THEOREM

This method determines if two convex shapes intersect. It can be also used to calculate the minimum penetration vector but we are only interested in knowing if two shapes overlap. Even though we will be working with two-dimensional shapes, it is also applicable to 3 dimensional objects. [2]

The idea behind the algorithm is that if two shapes are not intersecting, then there is an axis that separate both without touching them. It tests a number of axes, looking for the one that meets the condition.

To determine if an axis does separate the shapes, both of them are projected onto the perpendicular line to the axis. By doing this, we reduce in one dimension the problem and now it is only necessary to check if the projections intersect. In one dimension, they overlap if at least one of the extremes of one of them is between the extremes of the other projection.

The number of axes that could be separating axis is high, but the algorithm will not check them all. For this matter, every axis is equivalent to one of the axes parallel to the edges of the shapes. In our specific case, since all the shapes the game uses are regular rectangles and the rotation is also known before hand, we can skip this step. Instead of calculating the parallel axis to all edges, we project shapes in four axes: one corresponding with x axis, another with y axis, 25° or -25°. Every allowed rotation will use one of these four.

To project the blocks into the axes, we perform a dot product operation between the axis and the vertices. The maximum and minimum will be the extremes of the projection.

A.2 VISUALIZATION OF RESULTS IN GAMES

A.2.1 *Experiment 1*

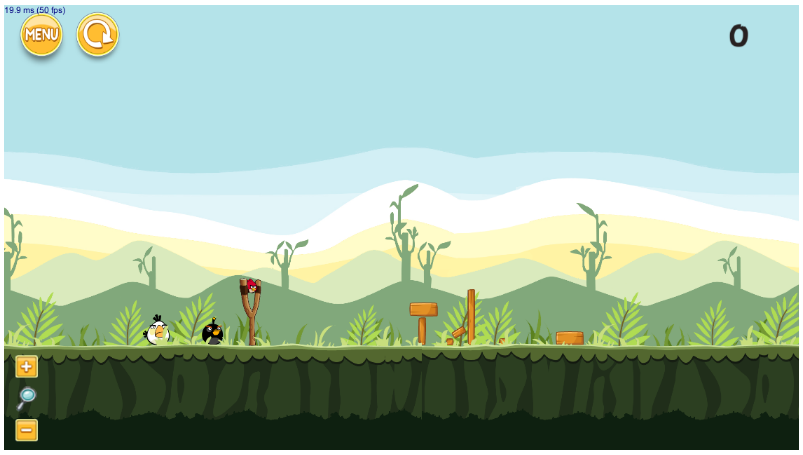

Figure A.1: Fitness = 5.496, nBlocks = 9, G = 100

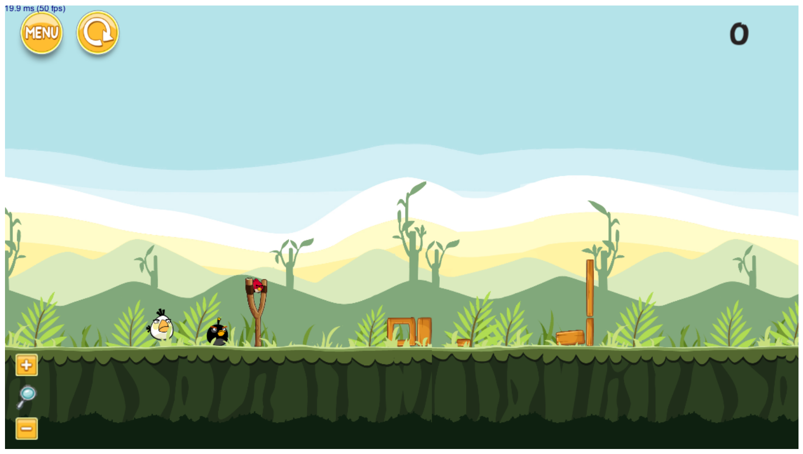

Figure A.2: Fitness = 7.777, nBlocks = 8, G = 100

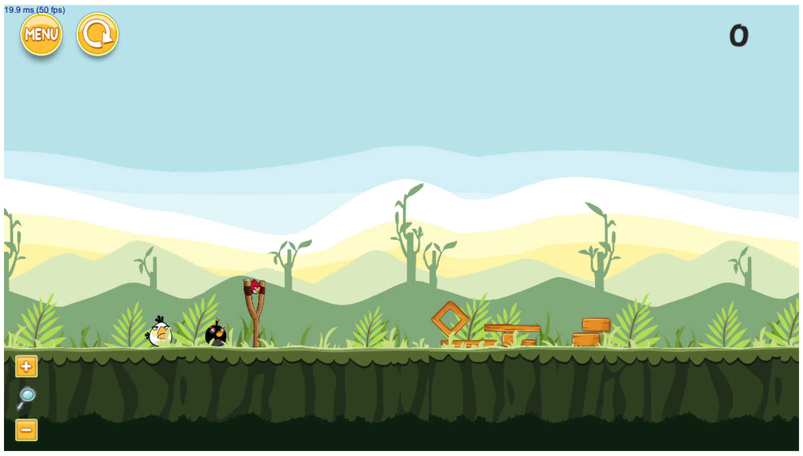

Figure A.3: Fitness = 8.154, nBlocks = 7, G = 100

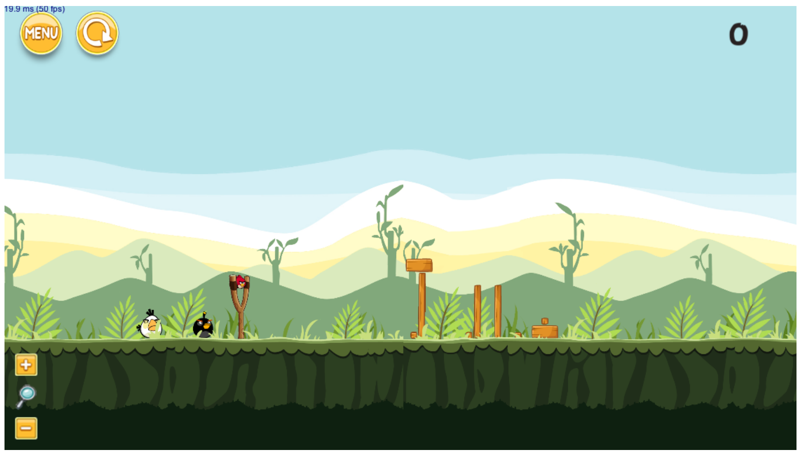

Figure A.4: Fitness = 9.021, nBlocks = 7, G = 100

A.2.2 *Experiment 2*

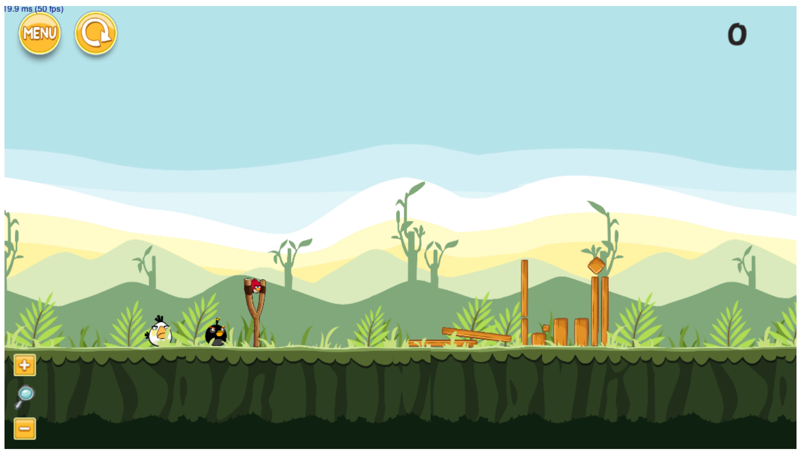

Figure A.5: Fitness = 5.276, nBlocks = 11, G = 383

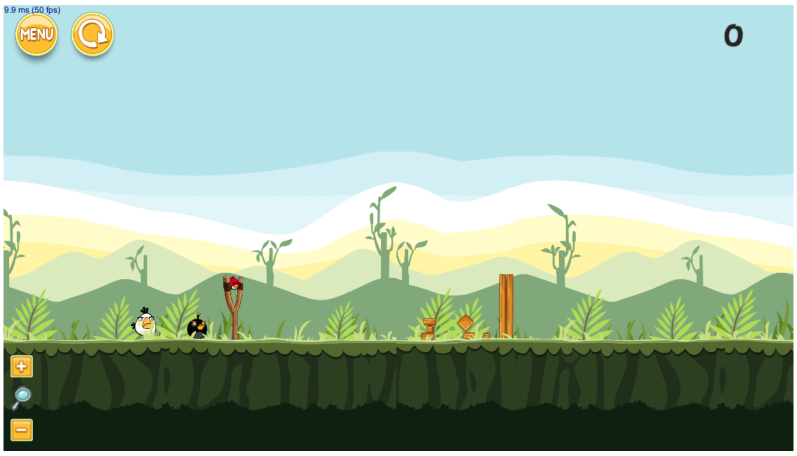

Figure A.6: Fitness = 2.909, nBlocks = 8, G = 364

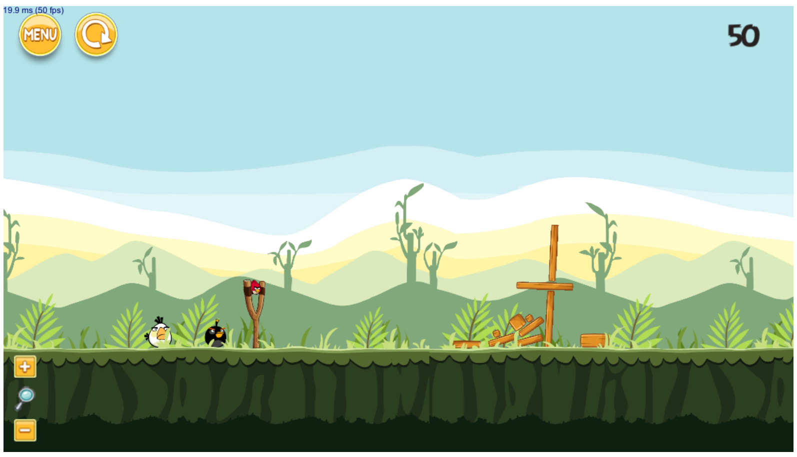

Figure A.7: Fitness = 203.776, nBlocks = 13, G = 44

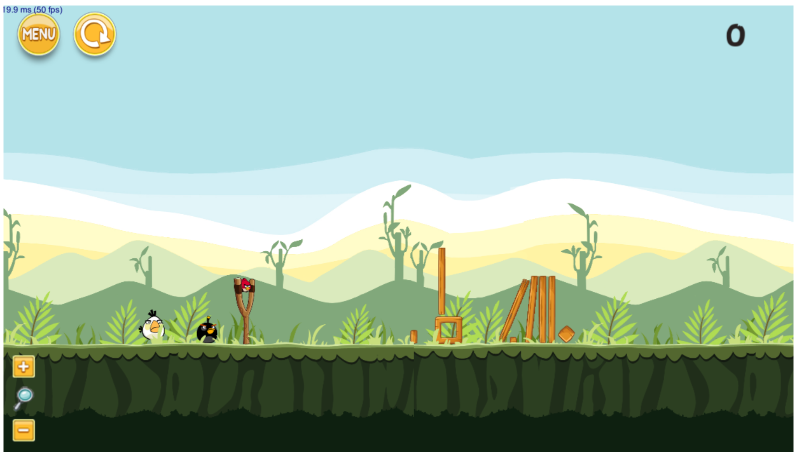

Figure A.8: Fitness = 5.176, nBlocks = 9, G = 767

A.2.3 *Experiment 3*

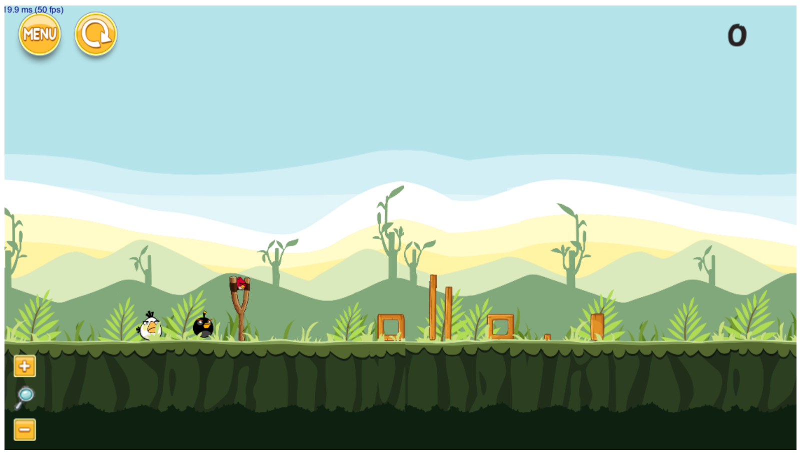

Figure A.9: Fitness = 8.334e-8, nBlocks = 6, G = 52

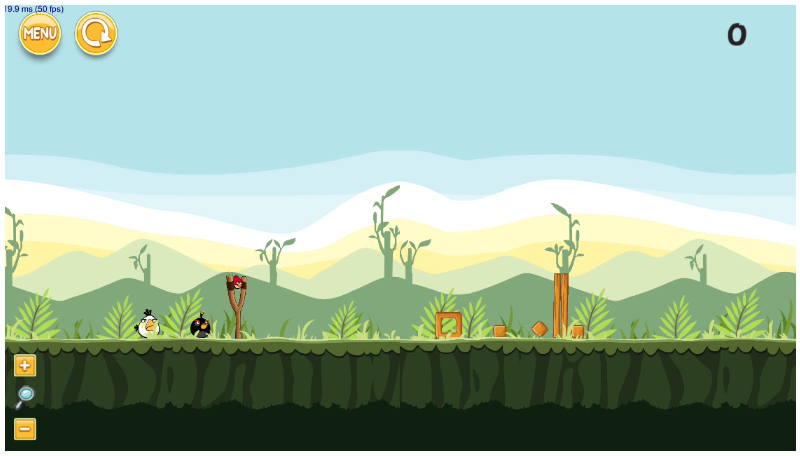

Figure A.10: Fitness = 6.035e-8, nBlocks = 6, G = 69

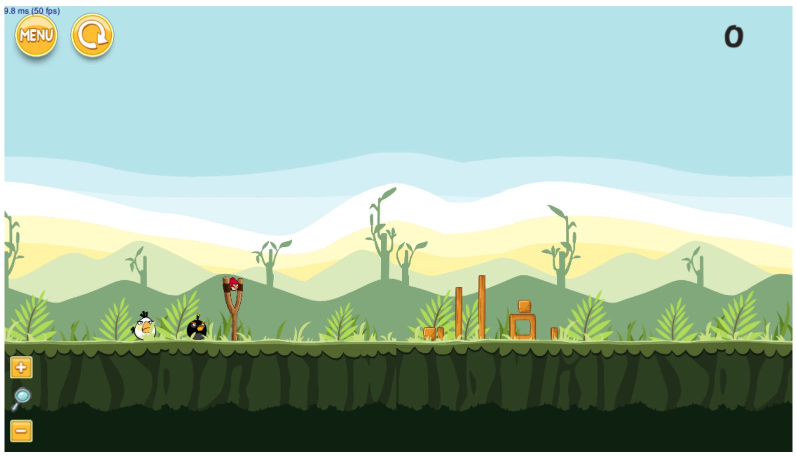

Figure A.11: Fitness = 9.856e-8, nBlocks = 7, G = 76

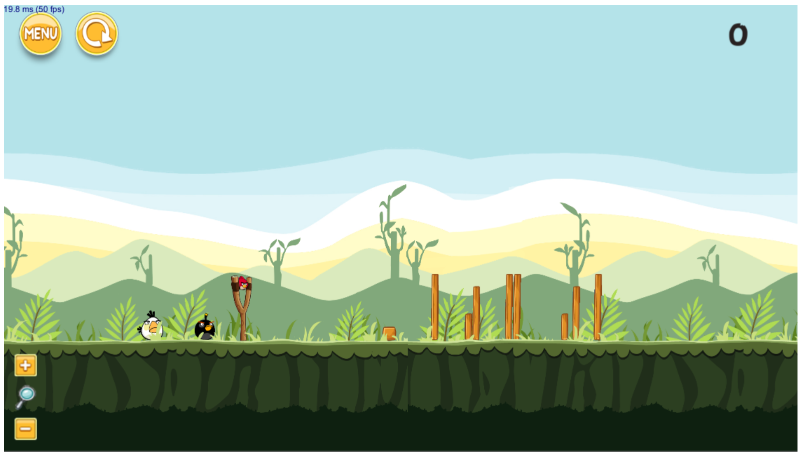

Figure A.12: Fitness = 1.571e-7, nBlocks = 9, G = 194

A.2.4 *Experiment 4*

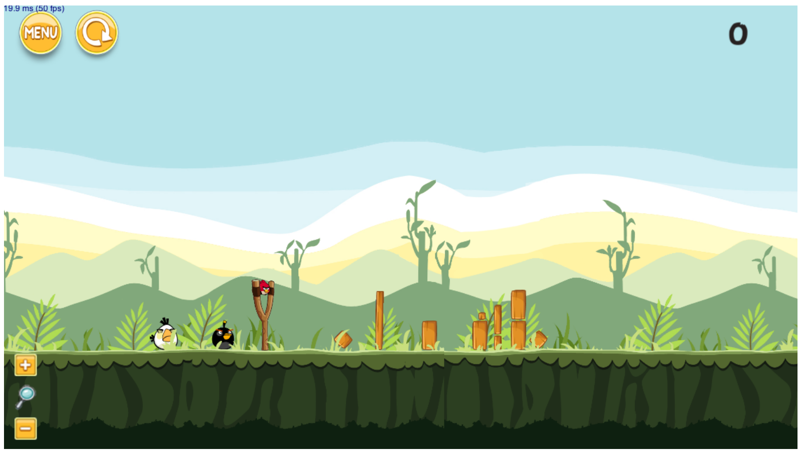

Figure A.13: Fitness = 9.229e-7, nBlocks = 10, G = 198

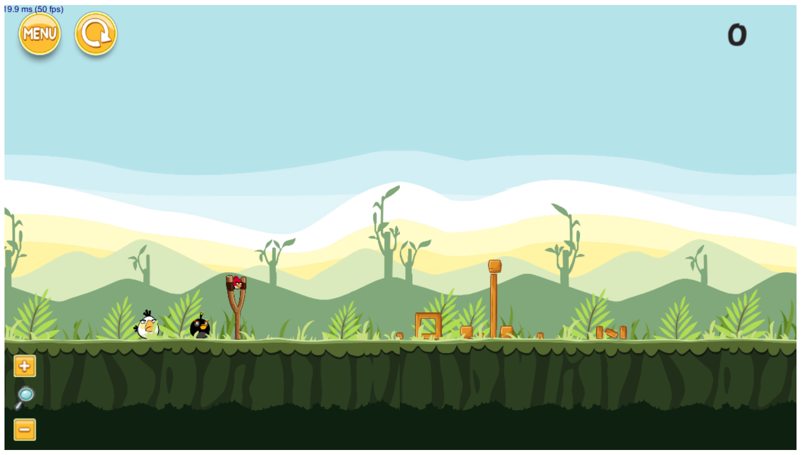

Figure A.14: Fitness = 1.065e-7, nBlocks = 11, G = 344

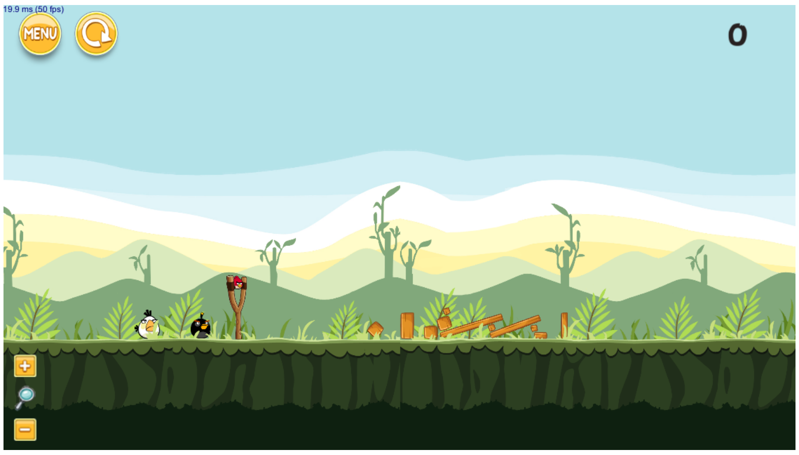

Figure A.15: Fitness = 8.395e-5, nBlocks = 11, G = 375

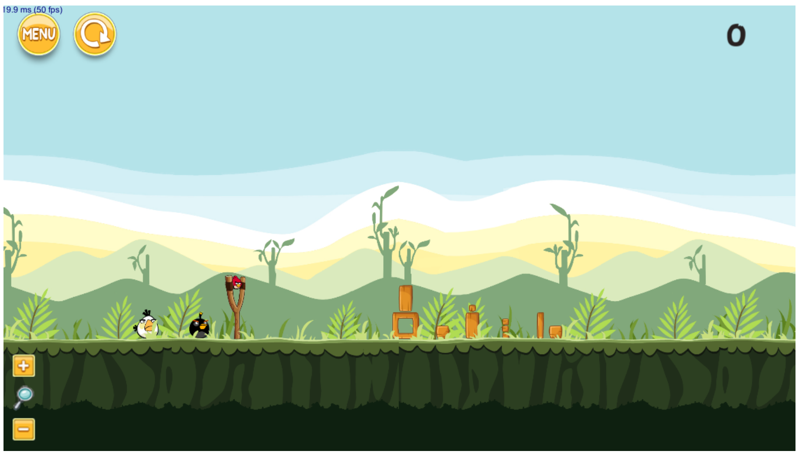

Figure A.16: Fitness = 8.946e-6, nBlocks = 10, G = 320

BIBLIOGRAPHY

[1] Rovio Entertainment Corporation. *Angry Birds official site*.

[2] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[3] Lucas Ferreira and Claudio Toledo. "A search-based approach for generating angry birds levels." In: *Computational intelligence and games (cig), 2014 ieee conference on*. IEEE. 2014, pp. 1–8.

[4] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. "DEAP: Evolutionary algorithms made easy." In: vol. 13. Jul. 2012, pp. 2171–2175.

[5] Gregory S Hornby and Jordan B Pollack. "The advantages of generative grammatical encodings for physical design." In: *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*. Vol. 1. IEEE. 2001, pp. 600–607.

[6] *IEEE Conference on Computational Intelligence and Games*. `http://www.ieee-cig.org/`. Accessed: 2018-05-22.

[7] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. "SGE: a structured representation for grammatical evolution." In: *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer. 2015, pp. 136–148.

[8] Nils J Nilsson. *Artificial intelligence: a new synthesis*. Elsevier, 1998.

[9] Christian S Perone. "Pyevolve: a Python open-source framework for genetic algorithms." In: vol. 4. 1. ACM, 2009, pp. 12–20.

[10] Thomas P. Runarsson and Xin Yao. "Stochastic ranking for constrained evolutionary optimization." In: vol. 4. 3. IEEE, 2000, pp. 284–294.

[11] Thomas Philip Runarsson and Xin Yao. "Evolutionary search and constraint violations." In: *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*. Vol. 2. IEEE. 2003, pp. 1414–1419.

[12] Gillian Smith. "An Analog History of Procedural Content Generation." In: *FDG*. 2015.

[13] Matthew Stephenson and Jochen Renz. "Procedural generation of complex stable structures for angry birds levels." In: *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*. IEEE. 2016, pp. 1–8.

[14] Matthew Stephenson and Jochen Renz. "Generating varied, stable and solvable levels for angry birds style physics games." In: *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*. IEEE. 2017, pp. 288–295.

[15] Julian Togelius, Noor Shaker, and Mark J. Nelson. "Introduction." In: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Ed. by Noor Shaker, Julian Togelius, and Mark J. Nelson. Springer, 2016, pp. 1–15.

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

[17] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. "What is procedural content generation?: Mario on the borderline." In: *Proceedings of the 2nd international workshop on procedural content generation in games*. ACM. 2011, p. 3.

DECLARATION

Yo, Laura Calle Caraballo, alumno del Grado en Ingeniería Informática de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada con DNI *, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Granada, June 16, 2018

Laura Calle Caraballo

colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Thank you very much for your feedback and contribution.

Final Version as of June 16, 2018 (`classicthesis` versiono).