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

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

June 6, 2018

ABSTRACT

ACKNOWLEDGMENTS

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

PCG Procedural Content Generation

SBPCG Search-Based Procedural Content Generation

NPC Non-Playable Character

GUI Graphic User Interface

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 [4].

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.[3]

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 expressivity 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).[8]

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* [10].

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.[5]

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: [8]

- *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 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 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[9]:

- *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, replacement operator decides which individuals

make it to the next generation. In some strategies, offspring completely replace the previous generation, while others maintain best individuals through several generations.

Part II

THE STATE OF THE ART

You can put some informational part preamble text here.
PLACEHOLDER TEXT

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 [7] [6] 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 complete. The second solution also tries to add variety by replacing some blocks with others of the same hight.

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 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[2] 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:

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

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% 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 and 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.

Part III

METHODOLOGY

4 OPEN SOURCE GAME ADAPTATION

As a part of their solution, Lucas Ferreira and Claudio Toledo[2] 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.

4.1 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 with *Assets/StreamingAssets/Levels* directory. Each level is described in a different file, similar to listing 4.1

```
<?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 4.1: 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

- Birds list, 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 [2] 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.

4.2 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 is 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.

4.2.1 *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 it 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 4.2 (note that encoding was also changed from original):

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <Level width="2">
3   <Camera x="0" y="0" minWidth="20" maxWidth="25" />
4   <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 4.2: 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).

4.2.2 *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 places 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 if 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 10*s* 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.

Part IV

APPENDIX

BIBLIOGRAPHY

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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 6, 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 6, 2018 (`classicthesis` versiono).