Procedural Content Generation in a Physics-Based Puzzle Game

Abhishek Akshat

Abstract

This paper presents an interesting Procedural Level Generation algorithm set up in a physics-based puzzle game that is a cloned version of the popular Angry Birds game. We will be looking at two research questions firstly on Procedural Content Generation secondly on improving the Angry bird agent. In this paper, we explain how the generator generates structures in interesting ways to make the game-play more challenging. Moreover, we also discuss on the agent's engagement with these generated levels while trying to optimize the game-play by scoring more while using less number of birds. Various experiments were conducted and we were able to device some interesting and some fun levels.

1 Introduction

Who doesn't love games? We all do. In our free project we will be working on a simple cloned version of Angry birds[3] [4].The original game is created by a Finnish company called Rovio Entertainment[1]. Which has huge number of downloads makes demanding and challenging levels. Physics based games like these become particularly interesting from the research perspective because problems that need to be solved by AI systems emulate the ones that are intended to interact successfully with the real-world. This ability to be able to estimate the consequences of a physical action based on visual inputs or other forms of perception is essential for the future of ubiquitous AI, and has huge real-world relevance and application. Also, it is among the popular mobile games with more than 6 million downloads on the Android play store. This game is relatively simple to play as well since all the game player needs to do is to destroy all the pigs in the game. The trick is that most of them are protected or covered by various objects like stone, ice, and wood. The game also provides us with various birds that have special powers. Using a catapult, the player is supposed to shoot those birds and destroy all the pigs. This uses real physics considering the firing angle and how back you pull the elastic the bird is going to be projected into the air similarly. We can see from Fig 1 to visualize the game.

The game gets interesting and difficult with an increase in the level. The game also offers some more damage-causing elements like TNT. These cause huge damage and these will help and score more points from the game. Now we know that the game is complex and makes up for a challenging task to build levels and agents. Thus paving way for a competitive space for `AI Birds`[2] where participants can create there own levels and agents. This competition received attention from various

research groups which were interested to win the game but also solve some essential problems for Game AI. AI Birds initially, rolled out a base level generator and a basic agent where participants were able to add more features to the existing version. This competition was held in the various section where users were asked to come up with interesting level generators and agents. Lastly, this competition was held every year.

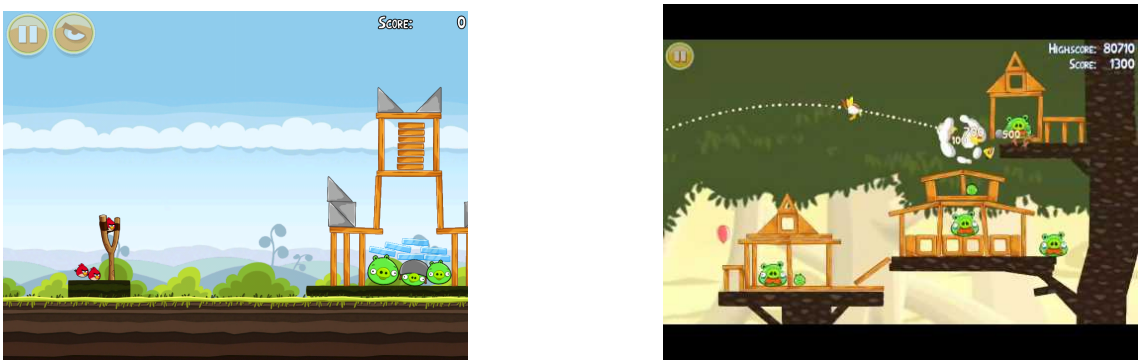


Figure 1: Angry Bird game sample images

In this project, we would like to focus mainly on level generations, on how to build a stable, interesting, and challenging levels. Where we will be comparing the baseline level generator on two-game agents one being Navies agent while the other one being DQN(Deep Q-network) agent. Keeping this as a baseline we will be comparing it against the level generator that is being done by us. We will understand how good our levels are by comparing the scores that agents were able to score. Keeping in mind that if the agent is scoring less score then we can say that the levels were difficult. In the further section we will learn and understand more about the game and agents.

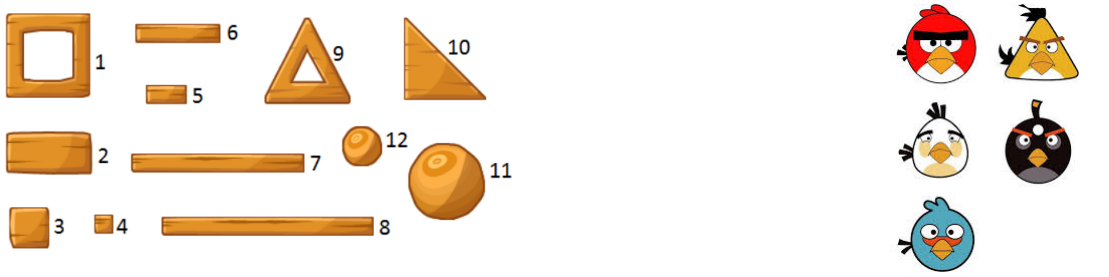
2 Game Environment

We experiment on a physics game developed by *Lucas N. Ferreira*[3] that is an implementation of Angry Birds[1] and is popularly known as 'Science Birds' throughout the AI birds community[2]. We also require a working game environment which provides us a base platform to run various experiments on. In the past few years, they have successfully organized three competitions encouraging people to develop code for creating intelligent game playing agents as well as generating levels for playing the game. This is a platform based on Unity3D and is executable for all Linux, Windows and Mac OS, but is run locally on a game server with no access to the internet. We were able to find one such working environment that provided us with a testing ground to run all our experiments and test our level generator using multiple agents, as we discuss in later sections of this paper.

This environment provided a feasible space to run the game on Windows, Linux and MacOS with some base requirements of Python environment and Java environment. However, although we were able to get our level generator to function well, we wanted to compare our results with other relevant work presenting the level generation approach. In order to do this, we establish a ground truth for our game which remains standard throughout and provides for a framework to build levels from. We define the ground truth in the following subsection.

2.1 Ground Truth

The game comprises of 5 elements, namely, bird, pig, stone, TNT and block. It provides us with 3 kinds of base movable blocks, namely, square, rectangle and circle. However, these blocks are arranged in different sizes and orientations(rotated 90 degrees) thus constituting a total of 12 kinds of blocks as seen in the following Figure 2.



(a) Types of blocks

(b) Types of birds

Figure 2: Elements within the game[2] structure.

These blocks are used in each game to create independent structures that are oriented in various ways in the different levels. These blocks may again belong to a type based on the material i.e. ice, wood or rock and have variable properties of strength and thus are affected differently on impact. Each of these structures are supposed to fit in a particular area with pre-determined dimensions by the defined environment.

Bird Type	**Strength**	**Feature**
Red bird	Nil	Regular
Yellow bird	Wood blocks	High speed
White bird	Nil	Drops an explosive egg
Black bird	Rock blocks	Explodes on contact
Blue bird	Ice blocks	Split into three birds

Table 1: The strengths and features of each bird. The strengths column defines the kind of block each bird is effective against, while the features column defines the bird's characteristic features.

The game environment provides for five different kinds of birds. In the table above, we list down the types of birds and their special characteristic features as defined in the game[2].

Then there are Pigs which exist in three sizes, namely, small, medium and large. The goal of the game is to kill all the pigs by crushing them under various blocks in order to register a win. The game also provides various birds that have special powers and may be identified from their color and shape. The game also comprises of a TNT element block that explodes when a bird hits it. In the original Angry Birds game[1], there are various terrains. Whereas, for our Science Birds game[2] setup, it is restricted to variations in the flat terrain in order to replicate irregular terrains thus making game more challenging.

3 Methods

As previously mentioned, we like to focus our work on developing interesting levels for a player to be able to play. Since the Game AI aspect of our project has much to do with Procedural Level Generation(PLG), in this section we formally define PLG and dive into how we reflect these ideas in our work. PLG is among the most popular and widely used forms of Procedural Content Generation(PCG). The domain has been extensively implemented in many digital games for better and automatic content generation. Our project focuses on PLG that is essentially described as the generation of game content given the game environment's base components and structural properties. Now we start our work with the goal to be able to develop an AI that is able to generate interesting levels for a player. In this approach, we would like to comprehend the levels as more or less unanimous and not definitive of ease of play-ability of a player agent. Thus, out first and primary focus is to develop discrete levels only. Further in our report we attempt on being able to arrange these levels in an orderly fashion based on some score. But here we discuss our approach for generating these levels first.

There are many approaches to generating levels, such as being based on a number of factors like number of structures, stability of structures, overall stability of the levels, probability of different structures appearing in these levels. For our project, we aim at making the game-play more interesting for the player agents, as well as to produce levels that are stable and solvable. We take a look at the level generator that we use for our project in the following subsection.

3.1 Baseline Level Generator

The main idea behind generating levels for the game is to be able to use the various block available in the environment and use them to create structures as shown in the Figure3 below. These structures are created by arranging the blocks in a row-wise manner, originating from a peak at the top and recursively placing blocks below it in different ways in each of the different levels. This selection of blocks is based on selection probabilities for each block type. This process is recursively repeated in order to create rows of blocks beneath the previously created row of blocks. These blocks on the base row of the to-be-structure are split into subsets on the basis of distances between them in order to form a set of all possible subsets. The position of a new block underneath an existing block is decided on the basis of a probability table for this set of all possible subset combinations. There are three possible positions derived from this process:

- under the center of each subset.
- under the edges of each subset.
- under the mid-points of the center and each edge of each subset.

Thus, these three positions combine to provide for 7 different positions to each possible subset. These are then checked for validity, ensuring no overlap between blocks. Finally, the generated level comprises of structures made up from constituent blocks and is represented in the Figure3 below. After which, the generated level is populated with pigs as well as some additional blocks.

Figure 3: A sample structure formed within our level generator.

While the positions for pigs is determined on the basis of a rank determined on how much a pig is shielded from the structures. Whereas, the additional blocks are selected at random and placed on the remaining positions in the level. Finally, the level generator checks whether or not there is an easily reachable position of the pig by the player. If so, then there are three ways that the generator tries of deal with this:

- place vertical blocks on the left of this position so that it isn't easily reachable.
- add more blocks on the same row of the structure of this position.
- simply swap this position with a stone element.

The baseline level generator is essentially about creating these simple levels that might not even have structural integrity under certain conditions. In the generated levels from this generator, there is only one simple red bird described as such for the agent to choose. Only one kind of block is used in order to create structures for the levels. As a result of which instability of the generated levels was a huge problem. We strive to bring some overall changes to the generated levels which result in more stable and solvable levels. This is where we look into some interesting implementations of PLG in order to introduce structure stability, by diversifying the developed structures. We adapt the approach from the paper[5] for our experiments and compare with the baseline. In the following subsection we shed some light on this PLG algorithm for creating stable and solvable structures.

3.2 PLG algorithm

The main idea behind improving on the baseline level generator was to be able to generate more stable and more comprehensive levels. As discussed in detail in the previous section, we needed to optimize three parameters, firstly, number and types of blocks used to create structures in the level. Secondly, the placement and number of pigs placed within these created structures. Thirdly, the idea of overall stability of a structure is optimized. This is to say that we have an in-built structure evaluation checker, but we optimize it by checking for 5 degree rotations of blocks, both clockwise and anti-clockwise until an angle of 45 degrees. This creates 18 possibilities of rotations in which these structures would be stable. In order to encompass all three parameters we define a fitness function as defined in [5] as the following:

$$F = X\frac{1-d}{1+|d|} + Y\frac{n}{n+v} + Z(1-\frac{r}{18})$$

Apart from this, we also introduce the ability to provide multiple kinds of birds from the environment. The generator also defines the number of birds that may be used by a agent while playing the generated levels.

3.3 Agents

We need to test our generated levels for various parameters, such as stability, difficulty or playability, etc. For this, we test the generated levels using two agents. This is done in order to be able to compare the performance of two different agents with different abilities on the same levels that were generated. We first set a baseline, and choose a Naive agent and then proceed onto testing on a slightly more complex and advanced agent. This is the Deep Q-Network agent that uses Deep Reinforcement Learning strategies for game play. We describe both of our agents in detail below.

Naive Agent

The naive agent uses a simple strategy to use the in-built components to create relevant structures. It only considers the birds, pigs and the slingshot in order to play the game. It chooses a bird randomly from the available set of five birds, an then catapults it in a trajectory in the direction of the detected pigs. The direction of this slingshot is chosen at random. This agent provides a baseline for any game playing agent and helps us determine the stability and playability of the levels generated by our generator.

DQN Agent

Deep Q-Network agent, as the name suggests use Deep learning with the Q-learning approach to develop a good policy for playing the levels generated by our Level Generator. The 'Q' in the abbreviation represents quality and defines the quality of an action taken by the agent in a particular state. This quality measure is then used to compute the maximum discounted reward for taking an action in a particular state down to the end of the game. This reward is based on the score that the agent gets from the game during game-play. It seems un-intuitive to be able to compute these rewards up to the end of the game but that's what the beauty of Deep Neural

level generator	birds weight	blue birds weight	black birds weight	min platforms	max platforms	max TNT	min pigs	max pigs	levels stability
0	1.0	1.0	1.0	0	2	3	4	8	Stable
1	1.5	1.5	1.5	1	3	4	3	9	Stable
2	2.0	1.5	1.0	2	4	6	4	10	Stable
3	1.5	1.0	2.0	3	6	7	5	10	Stable
4	1.0	2.0	1.0	4	8	10	8	15	Stable

Table 2: Level generators and corresponding varied parameters

Networks is. It helps us define the entire search space on the basis of the Q-function and chooses the best option by learning through trial and error. Thus the agent develops an optimal policy over time and uses it to play the levels our the game. This optimal policy π, is essentially choosing the action(a) in a state(s) with the maximum Q-value from the Q-Function which is represented below.

$$\pi(s) = \underset{a}{argmax}\ Q(s, a)$$

We describe the algorithm in more detail to understand how well this agent maps the state-action space and thus being very efficient at solving the levels of the game. Initially our Q-function would be very inefficient in computing converging Q-values for performing any action by the agent. Thus a balance between Exploration and Exploitation is required.

Exploration serves to enable the network to explore the search space of all possible actions in a given state and eventually update the Q-function on the basis of the corresponding reward of the taken action. Essentially, the algorithm takes a random action thus performing 'crude exploration'. Whereas, **Exploitation** follows the procedure of accessing Q-values from the available up-to-date Q-function. This balance of exploration and exploitation could be conducted by various techniques such as the simple implementation of an exploration rate which depreciates with each iteration. Thus enabling the algorithm to initiate with an exploration strategy while returning converging Q-values until eventually it starts to exploit the developed Q-function. Consequently, it may be inferred that the Q-Learning algorithm integrates exploration as a part of the algorithm making it a rigorous and strong candidate for our game-play.

4 Experiments

For this section of the report we discuss our empirical analysis on the level generator discussed earlier in this report. In order to test our DQN and Naive Agent, we first varied few parameters in the baseline generator to generate stable levels to test and compare our agents' performances. The list of parameters that were varied and their values are given in Table 2.
We tested both our agents in the levels generated by all the level generators mentioned in table 2. The results are summarized in tables 3-7.
To get a clear picture of which agent performed better in which level generator, we look at the average scores of both these agents in all 10 levels generated by each level generator. These scores

Level	Naive Agent Score	DQN Agent Score
1	0	0
2	0	0
3	0	0
4	0	91620
5	0	0
6	0	60460
7	0	0
8	42360	0
9	0	0
10	0	0

Table 3: Scores for Naive Agent and DQN Agent in levels generated by level generator-0

Level	Naive Agent Score	DQN Agent Score
1	0	0
2	0	0
3	0	0
4	0	78480
5	0	51960
6	40540	54540
7	0	107670
8	0	0
9	0	65020
10	0	0

Table 4: Scores for Naive Agent and DQN Agent in levels generated by level generator-1

Level	Naive Agent Score	DQN Agent Score
1	0	0
2	0	0
3	118610	101570
4	0	0
5	0	0
6	109580	109660
7	87120	0
8	0	100190
9	0	0
10	0	0

Table 5: Scores for Naive Agent and DQN Agent in levels generated by level generator-2

Level	Naive Agent Score	DQN Agent Score
1	62200	0
2	0	0
3	0	112770
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	91360

Table 6: Scores for Naive Agent and DQN Agent in levels generated by level generator-3

Level	Naive Agent Score	DQN Agent Score
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	113260	0
10	0	0

Table 7: Scores for Naive Agent and DQN Agent in levels generated by level generator-4

Level Generator	Naive Agent Average Score	DQN Agent Average Score
0	4236	15208
1	4054	35767
2	31531	31142
3	6220	20413
4	11326	0

Table 8: Average scores of Naive and DQN Agents in 10 levels generated by different level generators

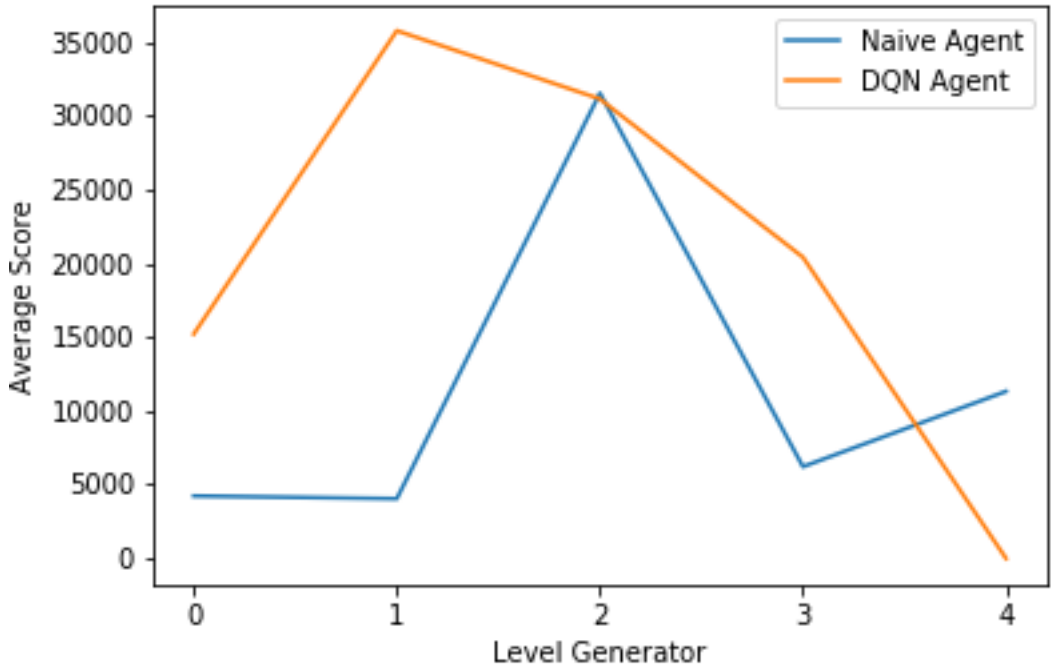


Figure 4: Average scores versus Level generators

are mentioned in table 8.
From table 8 it is evident that the DQN Agent performs far better than the Naive Agent in most of the levels. To examine and visualize how better the DQN agent performs as compared to the Naive agent, we try to plot a graph of average scores versus level generators for both the agents. The graph is depicted in figure 4.

5 Challenges

This project was certainly a very vast and consuming project, with which it presented some fun but interesting challenges. Even though we found ourselves stuck up on very primal things such as being able to setup a base working environment for our game, we learned some essentials about PCG. The most crucial one being that the re-usability of a PLG system is very limited and is primarily limited by the game environment we work in. One of the bigger challenges however was

posed by the limited number of features provided in the environment, leading to making way for only few ways to make levels more interesting. We did consider introducing more features like the addition of more core elements, however the game environment did not allow us to test these generated levels rigorously. We also had a hard time in being able to train our agents on these generated levels. The primary reason for this was that initially there was no way to ensure that only stable levels were generated and that the structures would not fall off before the game-play could begin. We did overcome this at a later stage by implementing a more robust generator that uses multiple blocks types in a more dynamic manner.

6 Conclusions and Future Work

After having conducted a plethora of experiments during the course of our project we were able to make some interesting observations. These observations were based on the null hypothesis that generated levels by our generator would not produce stable and solvable levels. In order to prove this wrong we tested the various versions of generated levels using our level generator to compare whether or not they were stable. Through our experiments we came to the conclusion that our level generator produces a lot more stable and solvable levels for playing the game Science Birds. We also deduced that the generated levels were much more difficult to play than compare to the baseline level generator we used. These conclusions are backed by the results of our empirical analysis on the two generator. We experimented with various tuned parameters for the generators and then play the generated levels using the two agents. Through the means of this project we were really inspired by the idea of being able to generate and suggest levels by an AI, based on the game-play style of the player agent. This would be particularly interesting for the domain of experience modelling. It would be very interesting to model a player's game-play by simple measures such as player score, and suggest a particular level based on difficulty of the level. Moreover, with physics-games like these, we can have some interesting real world applications of the same concept. But then that's an idea for a more ambitious project. Nevertheless, PCG techniques implemented in this project provided very interesting insights from the perspective of research student.

References

[1] Rovio entertainment. `https://www.rovio.com/`.

[2] Science birds competition. `http://aibirds.org/`.

[3] Lucas N. Ferreira et. al. Science birds. `https://github.com/lucasnfe/Science-Birds`.

[4] Lucas Ferreira and Claudio Toledo. A search-based approach for generating angry birds levels. In *Proceedings of the 9th IEEE International Conference on Computational Intelligence in Games*, CIG'14, 2014.

[5] Matthew Stephenson and Jochen Renz. Generating varied, stable and solvable levels for angry birds style physics games. In *Proceedings of the 13th Conference on Computational Intelligence and Games*, pages 288–295, 2017.