Artificial Intelligence Association

Technical report

Angry Birds Level Generation

Wolf van der Hert
Bram Grooten
Thomas Molier
Jelle van Kerkvoorde
Tunahan Sari

September 2020

Abstract

Team Amaru[1], consisting of five members from Serpentine, participated in the 2020 Angry Birds Level Generation competition. Serpentine is a student team of the Eindhoven University of Technology which competes in AI programming contests. In this competition, which is part of the IEEE Conference on Games, teams create a bot that can generate levels for the Angry Birds video game. The generated levels had to be: enjoyable, aesthetically pleasing, and challenging. There was a prize for each of these three criteria. Team Amaru programmed a bot that has a small data set of existing structures such as a windmill, a pyramid, or a ship. The bot selected a few of these structures at random to generate levels. With this Amaru finished in first place in the *challenge* category!

Figure 1: An Angry Birds level created by the Amaru-Generator.

[1]Contact us at: amaru2020@serpentineai.nl

Contents

1 Introduction

From June to August, 2020, five members of Student Team Serpentine competed in the 5th Angry Birds Level Generation competition [1], part of the IEEE Conference on Games [2]. For this competition, the team was asked to design an algorithm which could generate levels for the Angry Birds video game. After a quick few weeks, the team submitted their generator on the 7th of August, 2020. Out of the five competing teams, Serpentine Team *Amaru* won the first price for the *Challenge* section. The code can be found on the GitHub page of team Serpentine [4].
In this report, the strategies are first discussed in section 2, after which the implementations of these are discussed in section 3. In section 4, the results of the competition are presented and in sections 5 and 6, the project is discussed and a conclusion is given. The team is named after a flying snake, called *Amaru*.

2 Strategy

The strategy consists of a few key features. The parameter file is used as the input by providing instructions to the program. The information about the number of levels should be created, the number of pigs should be spread along with these levels and the materials used for the structures used in the levels and other details about the level design are provided here. On top of it, hard-coded structures are provided as explained in the implementation. The rest is planned to be done by the algorithms based on the rules that are set by our developers. In Figure 2 some examples are given to support the following information.

2.1 Baseline

The baseline code that was provided by the organisation was already capable of generating random levels. This baseline algorithm would first generate structures on the ground. When ground structures were constructed, the algorithm would try to fit one to three platforms based on their widths and heights, as explained in the following algorithms. Random structures were constructed onto the platforms using the same algorithm as on ground level. Pigs and TNT would be placed by replacing the blocks based on the algorithm that is given in implementation. In this paper, the given baseline algorithm of constructing random structures replaced with custom hard-coded universal structures. The implementation explains how the code is replaced with what intention. The idea of changing the baseline code was to create level designs that are not necessarily random but are rather looking aesthetically pleasing.

2.2 Universal structures

Universal structures form the basis of the strategy and should comply with a few requirements. First of all, universal structures should be diverse. Diversity in this context means that the generated levels will be looking different every time one is generated while it is aesthetically pleasing at the same time.
When all the materials (Wood, Stone, Ice) of a certain block type are restricted, and if all the universal structures are based on the specified block type, then no level can be generated. However, if a relatively large database of diverse universal structures is created, then no matter what restrictions are imposed on the code, the generator will always be able to find a suitable alternative structure.

Another requirement for a universal structures is being able to get 'scaled': Certain structures can be generated at different sizes. These different sizes create variety in the levels, while maintaining efficiency.

2.3 Pigs

Pigs are generated based on a few variables. One of the variables is the amount of pigs distributed over a number of levels, provided by the parameter file, and another variable is the generated structure itself. Based on these variables the amount of pigs used in a certain level can be generated.

2.4 Birds

The amount of birds and their types are determined based on the generated level and the number of pigs. Firstly, the different types of bird are specialised in tackling different kinds of blocks. As a brief introduction, yellow birds are great against wood, blue birds excel against ice and black birds are especially effective against stone blocks. Therefore, the type of bird the level comes with matters. The amount of birds is determined based on the difficulty of the level and the amount of pigs placed in that specific level.

Figure 2: Some structures from the database; a bird sequence.

3 Implementation

In this section, the algorithms that are developed based on the team's strategies are explained. A level is generated in five different steps, as follows: Design the level, merge several different designs, place the pigs, provide different types of birds based on the type of structures and finalise it with the right amount of birds based on the number of pigs.

Level designs are made based on the level-design algorithm where the algorithm takes individually crafted structures that are made based on the preliminary designs that are made by our team, see Figure 3. Thus, there is no algorithm available for the stated purpose since the structures are made individually. However, it is crucial to understand the mathematics behind the level designs since placing structures in a proper way is tougher than it seems due to the sensitivity of the structures. When a structure overlaps with a different structure, unexpected events happen as flying structures, instantly killed pigs and etc. Therefore, the measurements of sub-parts of the structure that is in the development process should be taken into account.

Merging multiple designs is a bit tricky due to the problem given above: Overlapping structures. Thus, the level designs should fit in the borders of the level and since doing this manually is extremely time-consuming, an algorithm to handle the stated progress is desired. However, there are multiple ways of merging the given structures and not all of them are efficient in the sense of playability. One idea is to place all the structures next to each other, making them stand on the same ground level, and another idea is to use the free space available on the top side of the level, making the structures stand at different heights. The use of free spaces offered the team a chance to come up with more detailed level designs. Therefore, an algorithm was developed based on these preferences. The algorithm works as follows:

1. As the input, use structures.

2. Get each structure's total width and height.

   (a) For the width, check the lowest and greatest x-coordinates and subtract them.

   (b) For the height, check the lowest and greatest y-coordinates and subtract them.

3. The difference in structure's width and height allow us to consider the structures as rectangles where their width is the difference in width, height is the difference in height as calculated earlier.

4. If the sum of areas (width x height) is lower than the area of the level, step on. Else, remove the structures that is making the sum greater than the area of the level itself.

5. Place the first rectangle first, then look for a suitable one for the second and continue like this. Since this is an NP-Complete problem, it is necessary to check every possibility one by one.

6. In a case where a rectangle cannot fit in anymore, go back to the previous rectangle and change its location. For a better understanding, check "Backtracking Sudoku Solving Algorithm"[**?**].

7. When step 6 is done, place platforms for the ones that are placed above the ground level of the level. This offers a chance for structures to stand above the ground level without falling down.

In conclusion, the rectangles are placed properly within the borders of the given level. In order to finish the level,placing the pigs, birds and other stuff which will allow the player to play the designed level.

Scalability is an important aspect to consider, in order to maintain diversity in between the levels. Being able to scale the designs allow structures to come up in different sizes.When step 2 is considered, we can have different sizes for the same exact structure, resulting in different values in areas.
The problem with scaling is that it is not possible to change the size of blocks that are used in the structure, it is only possible to make them use more blocks to look bigger. The issue with this idea is the chances of overlapping of blocks is quite high when their location are not recalculated properly and thus, the team should be able to provide enough space for blocks to get re-calculated their locations. The algorithm to handle this task as follows:

1. Take a parameter as a multiplier.

2. Multiply the length of the structures with their sizes

3. Re-calculate the coordinates of the blocks where no overlapping will happen as explained above

   To give an example, let two squares going to be get x3 bigger in size by adding additional blocks accordingly. Let their width and height is 1, and located on $x_1 = 0.5$, $x_2 = 1.5$ and $y_{1,2} = 0.5$, indicating that they are not overlapping since $1.5 - 1 \geq 1$. When they get multiplied, their width and height are now 3 but their coordinates are still the same, indicating that now they are overlapping. Thus, we should be changing their coordinates. Take 1 from x1, resulting in -0.5 and add 1 on x_2, resulting in 2.5, resulting in a scenario where they do not overlap, the maths holds as, $2.5 - 3 \geq -0.5$.

4. Do this recursively for each structure.

5. Feed the merging algorithm with the current output.

In conclusion, sizing your structures can be quickly done with the given algorithm.

The signature character of the game, green pig should be placed in such a way that the level design would look playable. After assembling the structures into levels, placing pigs is easily handled with an algorithm that works similarly to structure merging algorithm as explained above. In order to place pigs, it is desired to check for available places for pigs and after spotting all the free spots for the pigs to be placed, the algorithm gets terminated. However, the pigs should be placed in a reasonable way, indicating that they cannot be placed fully randomly. To avoid such problems, set of rules should be declared as input. The algorithm words as follows:

```
0: Get the number of available spots based on the given constraints.
// To give an example, it is intended to have pigs
// to be at a distance of at least 5 units from each other.
// This can be changed and manipulated extensively.

1: for each available spot in Level Designs
2: |     if constraints allow current spot == true :
3: |     |  Place a pig
   |     |  // Look for available spots and
   |     |  //  place your pigs one by one.
4: |     else:
5: |     |  pass
```

The last step of the level designs is to give the player some birds to play with. The number of birds depends on the number of pigs and their type depends on the block types used in the level. To give an example, yellow is great against the wood, black is great against stone. Thus, different birds are desired for different types of blocks. The algorithm works as follows:

1. Get the merged level design with pigs.
2. Decide on the number of birds with some multiplier on the number of pigs.
3. Decide on the type of birds depending on the amount of each block type.

After calling the birds with the given algorithm, the level is fully ready to be played. This process is done recursively for every level. The randomisation on selecting structures is important to avoid similarity between levels. A well crafted randomisation function is desired to provide diversity, to achieve the intended effect, modulus equations/functions are useful in such scenarios.

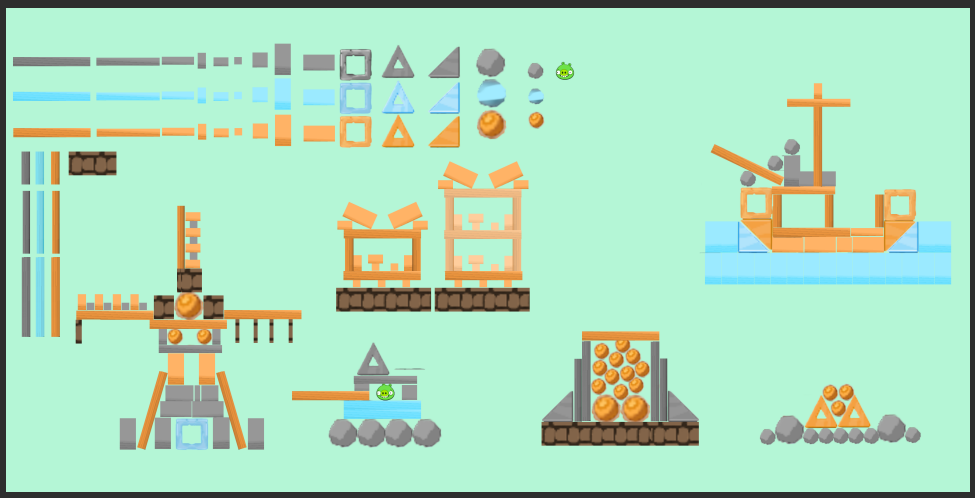

Figure 3: Some of the preliminary structure designs.

4 Results

The international Angry Birds Level Generation competition that was held during the IEEE Conference on Games (24-27 August 2020) [2] consisted of four entries in total, with other teams from universities in England and Australia.

Each team had to generate 100 different levels. If the levels were too similar, points were subtracted. The jury judged five levels from each team on three criteria: enjoyment, aesthetics, and challenge. See Table 1 for the results per category. As shown in the right most column, team Amaru won the challenge category! Amaru finished in second place in total, as shown in Table 2.

Table 1: Results per category.

Enjoyment		**Aesthetics**		**Challenge**	
MelodyGen	70 points	IratusAves+	65 points	Amaru	67 points
IratusAves+	68 points	MelodyGen	62 points	IratusAves+	52 points
Amaru	58 points	Amaru	56 points	MelodyGen	47 points

Table 2: Total results

Team	**Score**
IratusAves+	185 points
Amaru	181 points
MelodyGen	179 points
Naive	123 points

5 Discussion

Some major improvements can be made to the Amaru-Generator. First of all, the generator only uses pre-defined structures and does not actually 'generate' structures itself. This limits the variety in the levels. Therefore, possible innovations to the generator can include the generation of various sub-structures such as towers of varying height and width. Creating these additional structures also adds the features that the levels become more challenging and exciting.

Secondly, the position of the structures can be improved. Currently, the generator is not efficient because a rectangular no-spawn area is constructed around the structure. The size of the rectangular box is determined by the maximum width and maximum height of the structure. Consequently, lots of space is wasted and levels become stale and uneventful. Therefore, better levels can be generated when the collision box around a structure is reduced to the (convex) structure itself, as more structures can be generated in the same level without colliding.

Another possible improvement is the pig placement. At the moment pigs can spawn on random positions, for instance on the inclined part of a pyramid, and die instantly, granting the player free points while not utilising a single bird. The second problem is that too many pigs are generated on the same structure, meaning that a multitude of pigs can be destroyed by a single bird. This destruction might seem rewarding at first, but eventually the levels are too easy to complete. Therefore, the new pig-placing code should try to solve these problems.

Further improvements in several aspects of this competition can be made. However, as a first time team we believe that the knowledge obtained during this competition can be valuable for the next Angry Birds Level Generation competition.

6 Conclusion

As a team of Serpentine we are proud to have won the challenge track of the 5th Angry Birds Level Generation competition. The generator is not nearly complete towards our satisfaction, and can be improved in many ways. We hope to take on the 6th competition of this series, bringing the knowledge we gained from this competition. Combining pre-defined structures helped us this time around, but we look forward to bring deep-learning into the level generation next time.

Working on this project was a fun experience. Level Generation really forces another angle on gaming AI's, compared to the AI's that *play* the games. We are proud to have helped set the first few steps for future Serpentine teams.

Acknowledgements

This project was done within Serpentine, the student team of the Eindhoven University of Technology (TU/e) that competes in AI programming contests. We would like to thank the TU/e, as well as our industry partners VBTI and Flowserve, for supporting our work.

References

[1] AIBIRDS COG 2020 Level Generation Competition. `http://aibirds.org/level-generation-competition.html` Accessed on: 23 July 2020.

[2] IEEE Conference on Games. `http://ieee-cog.org/2020/competitions_conference`. Accessed on: 24 July 2020.

[3] Level Generation Competition - Basic Instructions. `http://aibirds.org/level-generation-competition/basic-instructions.html` Accessed on: 26 July 2020.

[4] Team-Serpentine GitHub Repository: Amaru-Generator `https://github.com/TeamSerpentine/angry-birds-level-gen-2020`

[5] Back-Tracking Sudoku Solving Algorithm, Geeksforgeeks. `https://www.geeksforgeeks.org/sudoku-backtracking-7/`

Appendix

Examples of exported levels by the Amaru Generator

Figure 4: Generated level with a train and table.

Figure 5: Generated level with a pyramid and windmill.

Figure 6: Generated level with a house and train.

Figure 7: Generated level with a ship.