Generating Varied, Stable and Solvable Levels for Angry Birds Style Physics Games

Matthew Stephenson
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au

Jochen Renz
Research School of Computer Science
Australian National University
Canberra, Australia
jochen.renz@anu.edu.au

***Abstract*—This paper presents a procedural level generation algorithm for physics-based puzzle games similar to Angry Birds. The proposed algorithm is capable of creating varied, stable and solvable levels consisting of multiple self-contained structures placed throughout a 2D area. The work presented in this paper builds and improves upon a previous level generation algorithm, enhancing it in several ways. The structures created are evaluated based on a updated fitness function which considers several key structural aspects, including both robustness and variety. The results of this analysis in turn affects the generation of future structures. Additional improvements such as determining bird types, increased structure diversity, terrain variation, difficulty estimation using agent performance, stability and solvability verification, and intelligent material selection, advance the previous level generator significantly. Experiments were conducted on the levels generated by our updated algorithm in order to evaluate both its optimisation potential and expressivity. The results show that the proposed method can generate a wide range of 2D levels that are both stable and solvable.**

I. Introduction

Procedural level generation (PLG) is the automatic creation of game levels without manual interaction and has become a key area of investigation for video game research [1], [2]. PLG can be used to generate a large number of levels in a short period of time. This can greatly reduce a game's development cycle and memory requirements [3], as well as dramatically increasing the amount of available content. The levels created can also be tailored to the user's playstyle, providing a unique and original gameplay experience [4].

Physics-based puzzle games such as Angry Birds, Bad Piggies, Crayon Physics and World of Goo have increased in popularity in recent years and provide many interesting challenges for PLG. Several papers have previously explored the use of PLG for physics-based puzzle games, most notably for the Cut the Rope [5], [6], [7] and Angry Birds games [8], [9], [10], [11]. The physics constraints employed in these types of games create many problems for PLG and make evaluating the quality of levels difficult. The playability/solvability of generated levels is particularly difficult to confirm, due to the exceptionally large state and action spaces [12]. Although the proposed generator is designed specifically for the Angry Birds elements and environment, the techniques used can be applied to many other games which share similar mechanics and level designs.

This paper presents an enhanced search-based procedural level generator for Angry Birds and other similar physics-based puzzle games. This algorithm is an updated version of that proposed in [13], [14] with several important improvements being made. One of the major changes we propose is to the fitness function, which was originally designed to create structures with specific properties such as height, width and number of blocks. Our revised approach evaluates structures on a much higher level, with new parameters for structure robustness, variety of block types, and pig dispersion, being used instead. Another significant change is that Angry Birds agents are now used to determine the number of birds provided for a generated level, as well as for estimating its difficulty and verifying that it is solvable. Additional improvements to the level generation process, including more varied structure designs, determining suitable placement positions for TNT, intelligent bird type and material selection, and variable terrain shapes, were also implemented. These combined updates significantly advance the capabilities of this level generator beyond that of the original.

Several experiments were conducted to analyse the expressivity of our level generator and to determine its capabilities. Metrics such as frequency, linearity, density and leniency were used to describe the characteristics of the generated levels. The optimisation potential of our algorithm was also investigated, as was the performance of different Angry Birds agents in solving the generated levels.

II. Angry Birds Overview

Angry Birds is a physics-based puzzle game where the player uses a slingshot to shoot birds at structures composed of blocks, with pigs placed within or around them. The player's objective is to kill all the pigs using the birds provided. A typical Angry Birds level, as shown in Figure 1, contains a slingshot, birds, pigs and a collection of blocks arranged in one or more structures. Each bird is assigned one of five different types (red, blue, yellow, black or white) and each block is assigned one of three materials (wood, ice or stone). TNT can also be placed within a level and will explode when hit by another object. The source code for the official Angry Birds game is not currently available, so a Unity-based clone created by Lucas Ferreira was used instead [8].

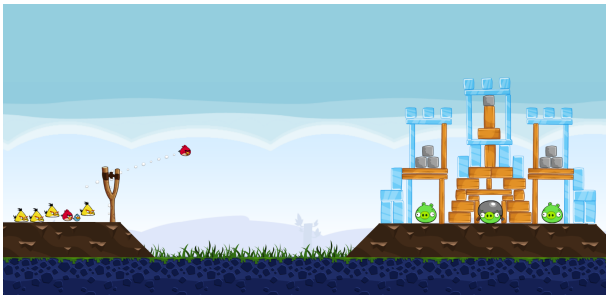


Fig. 1: Screenshot of a level from the Angry Birds game.

Before describing our algorithm's methodology, we will define some terms which will be used throughout this paper. A block is any object within the level that can be moved, apart from a bird, pig or TNT. Twelve different blocks are available within the unity clone, see Figure 2. Blocks one to eight are referred to as "regular" blocks, whilst blocks nine to twelve are called "irregular". A platform is any surface, apart from the ground of the level, which has a fixed position.

III. Original Level Generator

The proposed level generator described in this paper, builds upon a previous Angry Birds level generator, originally described in [13], [14]. It creates Angry Birds levels consisting of a collection of independent structures, constructed using the eight regular blocks available. Five of the regular blocks (2, 5, 6, 7 and 8) can also be rotated 90 degrees to give a different block shape. This creates a total of 13 different regular block types. A probability table is used to determine the likelihood of a particular block type being selected. Each block type is given a probability of selection, with all probabilities summing to one. Structures generated using this algorithm are made up of rows, with each row consisting of a single block type. These structures can have multiple peaks and feature a variety of placement methods for each row of blocks. Local stability requirements are enforced and more rows can be added until the structure reaches the desired size. Each block is also randomly assigned one of the three possible materials.

These structures are then distributed throughout the level, either on the ground (ground structures) or atop floating platforms (platform structures). The number of ground and platform structures, as well as their respective width and height limits, can be determined either manually or by random selection.

Once these structures have been placed, the level is then populated with pigs and irregular blocks, distributed on and within the created structures. Possible positions for pigs are identified and ranked based on a combination of structural protection (how much the surrounding blocks shield the pig from incoming shots), location dispersion (how far away this position is from other pig locations) and occupancy estimation (how likely it is for other objects to fall onto the pig). Pigs are then placed using this ranking until a desired number of pigs is reached. Any remaining locations are then substituted with randomly selected irregular blocks.

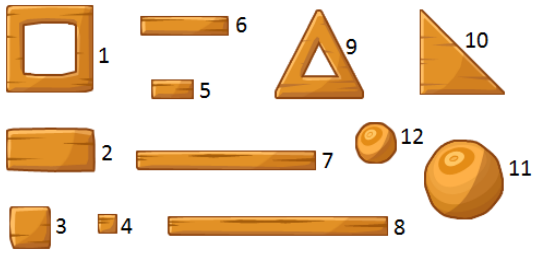


Fig. 2: The twelve different block types available.

Lastly, the generator attempts to identify and protect critical weak points throughout the level. A weak point is defined as a block within a structure that can be hit directly by a player's shot (reachable) and that if removed would affect a large number of other blocks and/or pigs. If a block is identified as a weak point within a structure then it is protected using one of three methods. The first method is to place a column of blocks to the left of the structure, such that the weak point is no longer reachable. The second method is to add more blocks to the structure row that contains the weak point, reducing the number of objects that would be affected by its removal. The third method is to simply set the material of the weak point to stone.

The number of birds that the player is given to solve the level is calculated using a simple formula that takes into account the number of structures and pigs within the level. The types of these birds are not considered by the original level generator.

For a more in depth explanation of the baseline structure and level generation processes, as well as examples of generated levels, please refer to the original papers.

IV. Improved Level Generator

This section of the paper describes the enhancements that have been made to the original level generator, to provide a more varied and robust method for creating levels. This includes processes for creating structures with multiple block types within a single row, terrain variation, TNT placement, global stability analysis, intelligent material and bird type selection, and using AI agents to determine the number of birds. Examples of fully generated levels can be found at the end of this section, demonstrating the enhancements described here.

A. Block Swapping

One of the main limitations of the original level generator was that each row of a structure always contained only one block type, significantly reducing the variety of structures that could be created. We therefore propose a simple modification that allows multiple block types within a single row. After a structure has been generated we attempt to replace some of the blocks in the structure with other block types that have the same height, a process referred to as "block swapping". For each block within a generated structure, we record a list of any other block types that have the same height as it and would still satisfy all local stability requirements if used as

a replacement. Each block then has a random chance (S) of swapping its block type with one from its list. The choice of which block type to swap to is determined using the original probability table from the structure's construction. Examples of structures with swapped blocks can be seen in Figures 3.a (central ground structure), 3.c (central ground structure) and 3.d (leftmost ground structure).

B. Terrain Variation

Another minor update to increase level variety is through the use of varying terrain height and angles. Whilst platform structures can be suspended in the air at varying locations, ground structures were previously always placed at the same height. Instead, we now allow ground structures to have terrain placed below them, resulting in an increased range of vertical positions. For each ground structure within a generated level (starting with the leftmost structure and moving right) there is a random chance (G) that the current height of the ground will increase or decrease by some amount. This amount of variation can be selected randomly but should have fairly small bounds to prevent it from increasing or decreasing too much. The height of the ground can never be lower than the base original ground height. The jumps in height between different ground structures are masked by using angled terrain, resulting in a smoother look. Examples of levels with terrain variation can be seen in Figures 3.a and 3.b.

C. TNT Placement

Whilst the original generator did not attempt to place TNT throughout the level, the proposed generator does. TNT is a small square shaped box that will explode when hit by another object, damaging and pushing away other nearby objects. Possible TNT locations are identified throughout the level, using the same approach as for pig positions, and are then ranked based on a combination of three factors.

The first factor (f_1) is how many pigs and structural weak points are within the TNT's blast radius. TNT boxes that are placed near to vulnerable targets will maximise the impact of their explosions and typically provide the player with alternative methods for solving the level. The potential damage that a TNT box has is calculated simply as the number of pigs (p_d) and weak points (b_d) within its blast radius. This value is then multiplied by a set weighting (A).

$$f_1 = A(p_d + b_d) \tag{1}$$

The second factor (f_2) is the overall dispersion of TNT throughout the level. Levels with TNT spread throughout them are typically preferable to levels with TNT grouped together, as setting off one of the TNT boxes will likely cause the others to explode as well. The dispersion value for a TNT location (t_l) is calculated as the product of the Euclidean distances between itself and all the TNT locations which have already been selected (t_s). This value is then multiplied by a set weighting (B).

$$f_2 = B \prod_{t_x \in t_s} \overline{t_l t_x} \tag{2}$$

The final factor (f_3) is occupancy estimation and is based on a technique called occupancy-regulated extension [15]. If a TNT location is lower than a platform and within a set distance (D) of that platform's edges then f_3 is equal to a set weighting (C) (otherwise $f_3 = 0$). This is because one of the key features within Angry Birds is the ability to cause TNT to explode with falling blocks, rather than with birds alone. TNT that is placed below or near other blocks which may potentially fall and hit it provides the user with this alternative choice of action.

The sum of all three of these factors gives a score for each TNT location. The location with the highest ranking is chosen and a TNT box is placed at the specified position. Any previously valid TNT locations that would overlap the newly placed TNT are removed. The remaining TNT locations are then re-evaluated and the highest ranked position is again selected. This process continues until either a maximum number of TNT boxes (T_m) is reached, there are no more valid TNT locations, or the score for the highest ranked location falls below some value (S_m). Examples of levels that contain TNT can be seen in Figures 3.b, 3.c and 3.d.

D. Global Stability Analysis

Another one of the major weaknesses with the original level generator was that it did not feature a reliable method for testing the global stability of the structures created. Although local stability requirements are enforced when adding blocks, the global stability of a structure must be determined after its construction. Whilst it is possible to guarantee that the structures generated would be stable by implementing stricter stability requirements when adding rows of blocks, this reduces the overall variety of content that can be produced. Qualitative stability methods, such as those described in [16], would provide a quick way of estimating stability, but they lack the robustness required for larger and more complex structures.

Instead, as all the relevant physics parameters (mass, density, friction and location) of objects are known beforehand, we can use the quantitative method described in [17] to calculate the global stability of the structures within our generated levels. Using this quantitative method is still not 100% accurate, as the Unity Engine upon which the Angry Birds clone is based suffers from simulation inaccuracies. However, we found that assuming zero friction for our quantitative stability calculations produced no false positives (i.e., all structures classified as stable by our quantitative analysis were also stable within the Unity Engine). Effectively, after each structure has been generated it is tested for global stability using this quantitative method. If the structure is deemed unstable then it is abandoned and a new structure is generated instead.

E. Material and Bird Type Selection

The original level generator did not address the use of multiple bird types and selected the material of blocks randomly. Both of these are limitations that heavily reduce the variety and enjoyment of the levels created. These two points are also highly interconnected, as many of the bird types in Angry Birds react differently to specific block materials.

There are three different materials that are available in Angry Birds; wood, ice and stone. These materials form a natural hierarchy within themselves, with stone being the heaviest and strongest material, and ice being the weakest and lightest material. The material for each block within a generated level is selected using one of several systems, described below. The trajectory analysis system is carried out first, after which each structure in the level is randomly allocated one of the remaining systems. Blocks that have already been set as stone due to them being weak points are exempt from this material selection process.

- Trajectory analysis: Two possible trajectories (low and high) are identified to each pig and TNT within the level. Each of these trajectories then has a random chance (p_t) of being selected. For each trajectory selected, set all blocks that intersect this trajectory to the same material (specifically either wood or ice) unless already set prior. Trajectories to pigs have higher preference than those to TNT and the highest ranked locations are done first. This results in interesting material paths for specific birds to follow in order to reach important or useful objects.
- Clustering: Pick a random block and set it to a random material. Find the next closest block that hasn't already had its material selected and set this block to the same material. Each time a block's material is set (including the very first block) there is a random chance (p_c) that the material will change. If this happens then the next selected block is used from now on when determining the next closest block. This continues until all blocks in the structure have had their material set. This results in a cluster like pattern of materials throughout the structure, as each block has a high likelihood of being the same material as the blocks around it.
- Row grouping: For each row within the structure, set all blocks to a random material
- Structure grouping: Set all the blocks within the structure to a random material. This material selection system only occurs in structures that have fewer than n blocks.
- Random selection: Set each block within the structure to a random material (original method).

There are also five different bird types that are available; red, blue, yellow, black and white. The special abilities of each of these birds are described below, along with the materials that they are strongest/weakest against.

- Red bird: No special ability, neither strong nor weak against any specific material.
- Blue bird: Splits into three birds when tapped, strong against ice blocks, weak against stone blocks.
- Yellow bird: Shoots forward in a straight line with increased speed when tapped, strong against wood blocks, weak against ice blocks.
- Black bird: Explodes either when tapped or after hitting an object, strong against stone blocks.
- White bird: Drops an egg directly downwards when tapped, this egg explodes after hitting another object.

For each generated level, we calculate the following scores:

- Red score = # reachable pigs / # pigs
- Blue score = # ice blocks / # blocks
- Yellow score = # wood blocks / # blocks
- Black score = # stone blocks / # blocks
- White score = # protected pigs / # pigs

(A pig is reachable if there is a trajectory to it that does not pass through any other objects, and a pig is protected if all trajectories to it pass through platforms/terrain.)

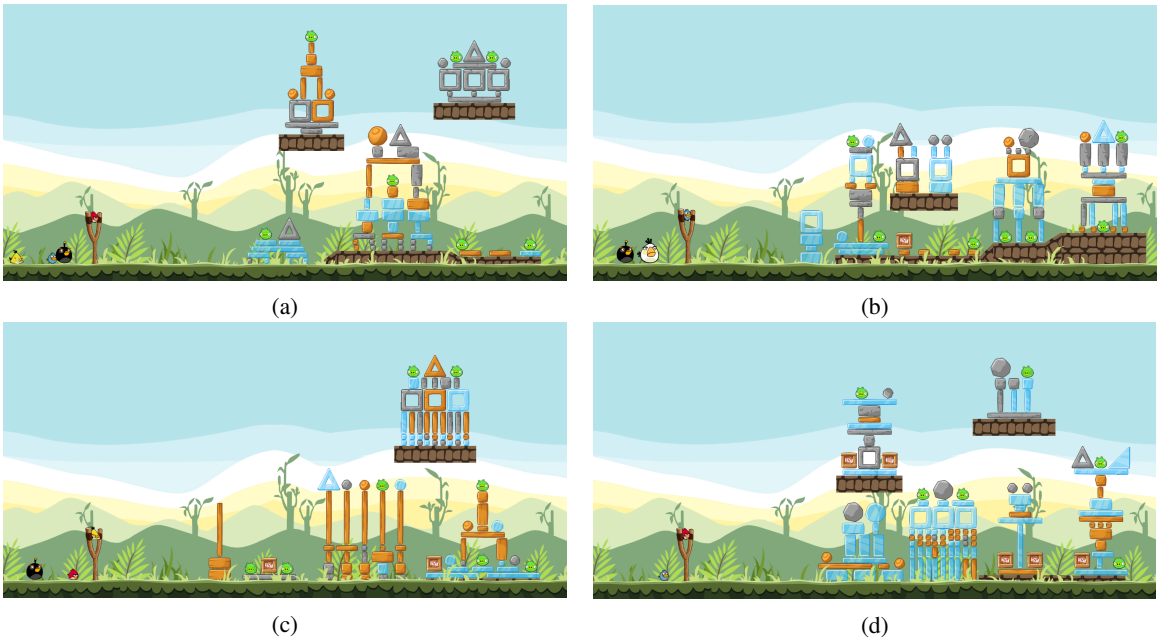
Each of these scores are then normalised so that they all sum to one, giving the desired ratio of bird types for that level. Bird types are then selected one at a time, always attempting to keep the ratio of selected bird types as close as possible to that of the desired ratio, until the desired number of birds is reached. If the ratio error is equal for multiple choices, then the bird type that is least present in the current selection is chosen. If this is also equal then the bird type is selected at random from these choices. This process can therefore be used to determine not only the types of birds that are available to the player but also their ordering.

F. Bird Number Selection

A critical, possibly even game-breaking, issue with the original generator was that it had no way of establishing whether a level it had created was solvable. The number of birds provided to the player was calculated using a very simple formula and was based only on the number of structures and birds within the level. Not only is this estimation of the number of birds required to solve a level exceptionally primitive, it cannot guarantee that the level is even solvable, let alone provide an effective measure of difficulty. Many of the levels generated were either far too easy or extremely difficult, perhaps even impossible to solve. To improve upon this, we propose the use of AI agents to both verify that a level is solvable and to select the number of birds that would provide a suitable level of difficulty for the player.

The Angry Birds AI Competition [18] was initiated in 2012, and for the past five years participants from all over the world have been submitting agents to take part in this competition. These agents are designed to solve Angry Birds levels using the fewest number of birds possible. The most recent competition was in 2016, where eight different agents competed. Once a level has been fully generated we let each of these eight agents play the level using a very large number of birds (e.g. 20). The exact number of birds used doesn't matter, just so long as there are enough that the agent could be reasonably expected to solve the level in this many shots. The type and order of each of these birds is determined using the method described in the previous section. The number of shots taken by each agent is then recorded and the fewest number of birds that was required by any agent is the number that is given to the player. If none of the agents can solve the level using all the birds provided, then that level is abandoned and a new level is generated instead.

Using these agents to evaluate our generated levels allows us to gain a more accurate estimation of a suitable number of birds for solving it, as well as confirming that the levels are indeed feasible with the birds provided. This, combined with



(a) (b) (c) (d)

Fig. 3: Four example generated levels using our new improved algorithm.

the other advancements described, mean that not only can our new level generator create a more varied and enjoyable set of levels, but that these levels are also be guaranteed to be both stable and solvable.

V. Fitness Function

One of the original papers [13] also proposed a fitness function that could be used to tailor the content generated over time. This was achieved by updating the values in a probability table used for selecting block types during the structure construction process. Whilst this method is an effective way of generating highly specific content, the fitness function used was fairly basic. This original function took into account the number of possible pig locations within the structure, the number of blocks within the structure, the aspect ratio of the structure, and the dispersion of possible pig locations within the structure. The problem with this is that while these factors allow for the user to heavily tailor the type of structure generated, all these values are either highly subjective or dependent on the size boundaries for the structure. Structures that are larger will typically have more blocks and more viable locations for pigs, as well as a lower average distribution. While the concept of ranking structures based on aspect ratio may allow for more user liberties, it does not typically result in more interesting structures.

We therefore propose a new fitness function that evaluates each structure on more objective factors, increasing the overall quality of the levels created without severely reducing the variety of generated content. This new fitness function takes into account three distinct factors, pig placement potential, block type variety and structure robustness, with a lower fitness value indicating a more desirable level.

A. Pig Placement Potential

This component of the fitness function is a merger of two of the previous fitness function factors, specifically the factors regarding the number and dispersion of possible pig locations. Instead of simply rewarding structures that have a lot of possible places to put pigs, we will now reward structures that have a large number of well dispersed possible pig locations. $|p|$ is defined as the total number of possible pig locations in the structure and d defines the dispersion value calculated using the same dispersion measurement technique proposed in [13]. To give a quick summary, this dispersion method divides the width and height of the structure by the square root of the number of possible pig locations, and then places a rectangle with this new width and height at every possible pig location. The total area that these rectangles cover gives an indication of how well dispersed these locations are (less dispersion means more overlapping rectangles and so less area is covered). The total area covered is then normalised by dividing it by the area of the structure's bounding box, to giving the value of d. The set factor X is used to adjust how much of an impact this component has on the structure's overall fitness value. This section of the fitness function is described by equation (3):

$$X\frac{1-d}{1+|p|} \tag{3}$$

B. Block Type Variety

One of the new components that we have added to our fitness function is the variety of blocks within the structure relative to the number of rows it contains. Instead of simply rewarding structures with more blocks (as the original method tended to do) which would highly favour smaller block types, we instead favour structures that are constructed using a wide variety of different block types. v is defined as the number

of different block types in the structure and n is defined as the number of rows within the structure. The set factor Y is used to adjust how much of an impact this component has on the structure's overall fitness value. This section of the fitness function is described by equation (4):

$$Y\frac{n}{n+v} \tag{4}$$

C. Structure Robustness

The other new component that we have added to our fitness function is the overall robustness of the structure against rotation. Although we will only accept a structure if our quantitative stability analysis method deems it globally stable, this does not tell us anything about how stable or robust the structure really is. In order to estimate this, we favour structures that will remain stable even when rotated. The structure being evaluated is rotated both clockwise and anticlockwise with angle intervals of five degrees, until the structure hits 45-degree rotation in each direction or becomes unstable. This gives a total of 18 possible angles at which the structure could be stable. r is defined as the number of these angles at which the structure was deemed stable. The set factor Z is used to adjust how much of an impact this component has on the structure's overall fitness value. This section of the fitness function is described by equation (5):

$$Z(1-\frac{r}{18}) \tag{5}$$

D. Complete Fitness Function

The sum of all these separate components for pig placement potential, block type variety and structure robustness makes up the complete fitness function, described by equation (6):

$$F = X\frac{1-d}{1+|p|} + Y\frac{n}{n+v} + Z(1-\frac{r}{18}) \tag{6}$$

VI. Experiments and Results

Several experiments were carried out to test different components of the structure generator and fitness function.

A. Probability Table Optimisation

As previously mentioned, a probability table for block type selection can be optimised over many generations using our specified fitness function. The training algorithm used for updating this fitness function is the same as described in [13]. To summarise, for each training generation nine separate structures are created. These nine structures are then ranked using the fitness function previously described. The frequency of block types in each structure is then used to update the corresponding sections of the probability table, with the highest ranked structures having the greatest impact and the lowest ranked structures having the least impact. The probability table values are then renormalised so that they again sum to one. Please see the original paper for full details on the probability table optimisation algorithm.

For our experiment, we initialised the probability table with equal values for all block types (1/13) and then repeatedly generated structures of random sizes (width limits between 3.0 and 10.0). For our fitness function, we defined: $X = 1.0$, $Y = 0.5$, $Z = 0.5$. This gives roughly equal weighting to the pig placement potential and block type variety components of the fitness function, with a slightly higher emphasis on the structure robustness component. The probability table was then updated over 300 generations of training (total of 2700 structures) with the current state of the probability table recorded after each generation. The result of this experiment is illustrated in Figure 4 (block types with an r-subscript indicate blocks that have been rotated ninety degrees).

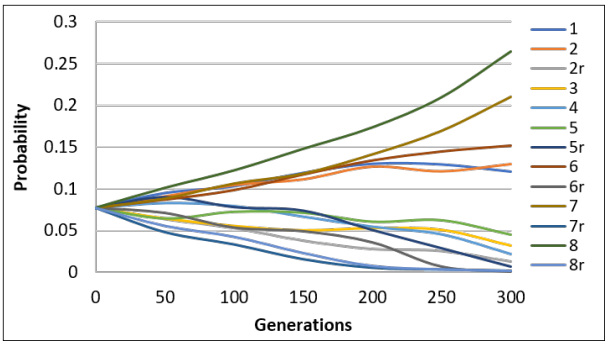


Fig. 4: Probability table values for each block type over multiple generations.

From this graph, we can see that over time the probability values for block types 1, 2, 6, 7 and 8 tended to increase (although the probability for block type 1 appeared to be decreasing towards the end), whilst the values for block types 2r, 3, 4, 5, 5r, 6r, 7r and 8r all decreased. This indicates that our function tends to favour wider blocks, as they provide a larger and more disperse set of possible pig locations, and also likely increase the overall robustness of the structure. The block type variety component of the fitness function keeps structures that contain only these desirable blocks from becoming too dominant, as we can see that the probability values tended to fluctuate over time.

Whilst this training process could carry on indefinitely, this would likely result in very small probability values for a significant number of block types. We therefore decided to cease training once the probability of selection for any block type dropped below 2.0%. In our case, this occurred for block type 7r after 131 generations. This optimised probability table was then used when analysing the expressivity of our new level generator.

B. Generator Runtime

The majority of our procedural level generation algorithm was coded using Python 3.4. The only exceptions being our quantitative stability analysis program, coded in C++, and our collection of Agents from the 2016 AIBirds competition, each of which was coded in Java. This software was all run on an Ubuntu 14.04 desktop PC with an i7-4790 CPU and 16GB RAM. For our experiments we generated 200 levels using our optimised probability table, each containing between two and four ground structures, between one and three platform structures, and between 6 and 10 pigs (all values selected randomly for each new level generated). For block swapping we defined $S = 0.5$. For terrain variation we defined $G = 0.5$.

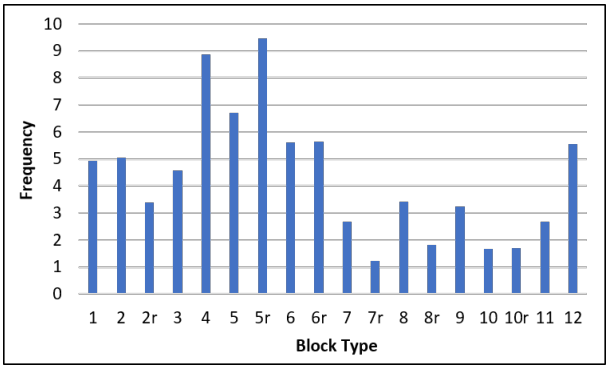


Fig. 5: Average frequency for each block type.

For TNT placement we defined $A = 1.0$, $B = 0.005$, $C = 2.0$, $D = 0.8$, $T_m = 5$ and $S_m = 6.0$. For material selection we defined $p_t = 0.3$, $p_c = 0.2$ and $n = 10$. All other generator variables were defined the same as those used in [14]. With the exception of the Angry Birds agents which will be discussed later, the average combined runtime of all other level generator components was 54.1 seconds. Additionally, our quantitative stability analysis program found a structure stable 68.2% of the time and produced no false positives.

C. Expressivity Analysis

The expressivity of a level generator is the space of all levels it can generate and is measured by evaluating various aspects of a level to identify its strengths and weaknesses. Several metrics have been proposed to analyse a generator's expressivity [19], [20]: frequency, linearity, density and leniency.

1) Frequency: Frequency evaluates the number of times that a block type occurs within a level. Figure 5 shows the average frequency of each block type within a level. Unsurprisingly, we can see that smaller block types such as 4, 5 and 5r have a much higher average frequency than larger block types. This is despite the fact that all three of these block types had their probability of selection reduced from their original values in the optimised probability table. This would suggest that the frequency of these block types would be even higher had we not optimised the probability table. The wider thinner block types favoured by our fitness function also appear much more frequently than their rotated counterparts. However, the difference is not as large as one would expect based purely on our optimised probability values. For example, the probability of selecting block type 8 is nearly five times as large as selecting block type 8r, yet its frequency is not even double that of 8r. This is likely due to the fact that wider block types are more likely to fulfil the necessary support requirements with a fewer number of blocks. Thinner block types require more blocks to satisfy these conditions and so are placed more frequently. However, overall, we can see that no block type has a restrictively small frequency value, meaning that the variety of structures created by our generator has not been severely reduced by the use of our fitness function to optimise the probability table.

2) Linearity: Linearity measures the "profile" of generated levels. Levels with objects placed at multiple heights throughout the level space will have a low linearity, while levels where the objects follow a straight line will have a high linearity. Linearity is measured by performing a linear regression, taking the centre points of all blocks, platforms, pigs and TNT boxes as our data points. Each level is then scored based on its R^2 value. The average linearity of a generated level is 0.0581, with a standard deviation of 0.0652. This result shows that our levels are highly non-linear, with objects being distributed throughout the entire level space.

3) Density: The density of a level represents the compactness of the objects placed within it. Density is measured by calculating the total area of all blocks, platforms, pigs and TNT boxes within the level space. This is then divided by the total size of the level space to give a value indicating how much of the level's area was taken up by these objects. The average density of a generated level is 28.7%, with a standard deviation of 5.23%. We believe this density percentage is suitable, as levels with a low density are likely to be sparse and uninteresting, whilst levels with a high density are likely to be too congested.

4) Leniency: Leniency is used to express how difficult a level is to successfully complete (i.e., kill all pigs with the birds provided). The difficulty of an Angry Birds level is therefore almost solely dependent on the number of birds provided to the player. This is in turn heavily dependent on the skill of the Angry Birds agents used to decide this number.

We therefore propose a new measure of leniency for games where agents are employed to verify that the levels created are solvable. We utilise a Naive agent that makes each of its shots at a randomly selected pig as the base method for determining the number of birds required to solve a level. Each of the other Agents is then compared against this. An agent that performs much better than the naive approach would indicate that not only is this agent very skilled, but that it is better at selecting the minimum number of birds required to solve a level. The difference between the best performing agent and this Naive agent is therefore a suitable measure of Leniency.

Eight agents from the 2016 AIBirds competition (including the Naive agent) were used to play the generated levels. Each of these agents has a different strategy for solving Angry Birds levels, with some using heuristic approaches, logic programming, or even simulations, in an attempt to solve them. All these agents were given three attempts to solve each level, and the number of birds that it took was recorded. The average number of birds (μ_B) required by each agent, as well as the standard deviation (σ) and runtime (seconds), to solve each of the levels is provided in Table I.

From this we can see that the Datalab agent performed the best, with an average of 3.40 birds used for each level. The Naive agent took 4.79 birds on average, giving us a leniency measure of -1.39 for the levels generated. This measure could be reduced even further (increased level difficulty) by developing either agents that can solve levels better, or levels that are harder for the Naive agent to solve. We can also see that the combined runtime for testing a level using all these agents is very high. It would therefore make more practical sense to only use a smaller subset of very good agents to

TABLE I:
AVERAGE NUMBER OF SHOTS REQUIRED BY EACH AGENT

Agent	**Shots** (μ_W | σ)		**Runtime** (seconds)
HeartyTian	4.35	2.32	93.5
AngryHex	6.32	3.28	151.4
Datalab	3.40	1.42	78.5
SEABirds	3.85	2.11	125.1
S-Birds	4.04	1.89	213.0
Naive	4.79	2.54	132.7
IHSEV	7.20	2.98	250.6
BamBirds	4.67	1.77	124.2

determine the bird number for a level. This means that by just using the Datalab agent we are able to generate a stable and solvable level on average every 132.6 seconds.

VII. CONCLUSIONS AND FUTURE WORK

This paper has presented a procedural level generation algorithm for Angry Birds style physics games, which can guarantee that the levels it creates are both stable and solvable. This generator builds upon a previously proposed algorithm to substantially increase the variety and validity of the levels created. Improvements not only to structures and terrain within these levels, but also to their evaluation and optimisation, produces levels greatly superior to those previously generated. We have also utilised Angry Birds agents to ensure that the levels created are solvable, arguably the most important requirement for level generation algorithms.

Each of the structures we generate is evaluated using a high-level fitness function, which considers pig placement potential, block type variety and structural robustness. This function can then be used to evolve the probability of selecting certain block types over multiple generations, resulting in a more fine-tuned and enjoyable set of levels. Each section of this fitness function can also be weighted independently, allowing the user to define which aspects of the generated levels are most important.

Our proposed level generator was evaluated in terms of its expressivity using a wide assortment of metrics: frequency, linearity, density, and leniency. These metrics were calculated using not only the type of objects within each level, but also their position and quantity. The results of this analysis demonstrated that our structure generator can create a broad range of levels with many desirable attributes.

There is an extensive range of future possibilities for this research. One example could be to develop a structure generation method that can create structures that are no longer segmented into distinct rows, or perhaps angled structures for sloping terrain. Another obvious improvement would be not to the level generator itself, but rather to the AI agents. The difficulty of procedurally generated content is particularly troublesome to measure, especially with games such as this which contain a near continuous state and action space. Improving the skill of these AI agents may also help improve the interest or quality of the levels we create, as it could be that levels which can only be completed by more advanced agents require a certain degree of skill or ingenuity to solve. Agents could also be used to alter the content of generated levels more than just the number of birds, perhaps by testing out different combinations of fitness values during training, or removing certain TNT boxes or bird types that were not used effectively. Performing a user study that compares these levels against those of the original Angry Birds would also give a good indication of the overall quality of the levels generated, as well as how the performance of our suggested agents compares to that of typical players.

REFERENCES

[1] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1–22, 2013.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[3] S. Dahlskog and J. Togelius, "Patterns and procedural content generation: Revisiting mario in world 1 level 1," in *Proceedings of the First Workshop on Design Patterns in Games*. ACM, 2012, pp. 1:1–1:8.

[4] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[5] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach," in *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 72–78.

[6] ——, "Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 215–216.

[7] M. Shaker, N. Shaker, J. Togelius, and M. Abou-Zleikha, "A progressive approach to content generation," in *18th European Conference on the Applications of Evolutionary Computation, EvoApplications 2015*, 2015, pp. 381–393.

[8] L. Ferreira and C. Toledo, "A search-based approach for generating angry birds levels," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 2014, pp. 1–8.

[9] ——, "Generating levels for physics-based puzzle games with estimation of distribution algorithms," in *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*. ACM, 2014, pp. 25:1–25:6.

[10] M. Kaidan, T. Harada, C. Y. Chu, and R. Thawonmas, "Procedural generation of angry birds levels with adjustable difficulty," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, 2016, pp. 1311–1316.

[11] L. T. Pereira, C. Toledo, L. N. Ferreira, and L. H. S. Lelis, "Learning to speed up evolutionary content generation in physics-based puzzle games," in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2016, pp. 901–907.

[12] M. Shaker, M. H. Sarhan, O. A. Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 2013, pp. 1–8.

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

[14] ——, "Procedural generation of levels for angry birds style physics games," in *Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 2016, pp. 225–231.

[15] P. Mawhorter and M. Mateas, "Procedural level generation using occupancy-regulated extension," in *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG)*, 2010, pp. 351–358.

[16] Z. Jia, A. Gallagher, A. Saxena, and T. Chen, "3D-based reasoning with blocks, support, and stability," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1–8.

[17] A. G. M. Blum and B. Neumann, "A stability test for configurations of blocks," Massachusetts Institute of Technology, Tech. Rep., 1970.

[18] J. Renz, "AIBIRDS: The angry birds artificial intelligence competition," in *Proceedings of the 29th AAAI Conference*, 2015, pp. 4326–4327.

[19] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, pp. 4:1–4:7.

[20] B. Horn, S. Dahlskog, N. Shaker, G. Smith, and J. Togelius, "A comparative evaluation of procedural level generators in the mario AI framework," in *Foundations of Digital Games 2014*, 2014, pp. 1–8.