Procedural Generation of Complex Stable Structures for Angry Birds Levels

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 content generation algorithm for the physics-based puzzle game Angry Birds. The proposed algorithm creates complex stable structures using a variety of 2D objects. These are generated without the aid of pre-defined substructures or composite elements. The structures created are evaluated based on a fitness function which considers several important structural aspects. The results of this analysis in turn affects the likelihood of particular objects being chosen in future generations. Experiments were conducted on the generated structures in order to evaluate the algorithm's expressivity. The results show that the proposed method can generate a wide variety of 2D structures with different attributes and sizes.**

I. Introduction

Procedural content generation (PCG) is a major area of investigation within the video game industry [1]. It is typically defined as the automatic creation of aspects of a game which affect gameplay other than non-player characters (NPCs) and the game engine [2]. PCG is commonly used to create new unique experiences for players without the need to design every possibility manually. This can dramatically cut a game's development time, as well as increasing available content and reducing memory consumption [3]. PCG can also be used to learn about the player's abilities and adapt the game's content accordingly [4].

Previous research has investigated the use of PCG for many different types of game content, including vehicles [5], weapons [6] and rulesets [7]. Level generation, or the generation of certain level aspects, is one of the most popular uses of PCG and has been implemented in many different game types. These include real-time strategy games [8], role-playing games [9], platform games [10], racing games [11] and arcade games [12].

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 PCG. However, as far as we can tell, very little research has been done on this particular area of PCG. A small collection of studies have explored PCG for the physics-based game Cut the Rope [13], [14], as well as the popular mobile game Angry Birds [15], [16], [17].

Physics-based games make PCG more difficult for a variety of reasons. Firstly, there are typically many constraints that dictate the types of content that can be created. Any PCG algorithm must be aware of the physical limitations of its environment and create content that functions as expected, e.g. a procedurally generated car must be able to drive and steer. Secondly, the state and action spaces are typically very large. This makes the task of determining if a procedurally generated level can be completed extremely difficult, especially for increasingly complex levels and content. Lastly, the variety of content that the algorithm can create must not be significantly reduced by any constraints imposed. The main appeal of PCG is that a large and diverse range of content can be created. Designing algorithms with restrictions that are unnecessarily strict will severally limit its PCG capabilities.

Previous research into PCG for Angry Birds has been rather basic in terms of the complexity of the structures they generate. These prior methods create Angry Birds levels by generating columns of either single objects or small predefined structures [16]. These columns are then recombined using simple genetic algorithms in an attempt to maximize structural stability [15], [17]. Whilst this method is suitable for creating primitive structures in Angry Birds levels, it cannot generate anything more complex than an array of single columns.

This paper presents a search-based procedural content generator for the Angry Birds game which can create complex stable structures using a variety of different objects. The structures are evaluated using an improved fitness function which measures various important aspects. These include the structure's block count, pig count, aspect-ratio and pig dispersion. The probability of selecting certain block types during the construction process is evolved over successive generations, using this function as the optimisation criterion.

Several experiments were conducted to analyze the expressivity and of the structure generator. Metrics such as frequency, linearity, density and leniency were calculated to describe the characteristics of the content generated.

II. Angry Birds

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

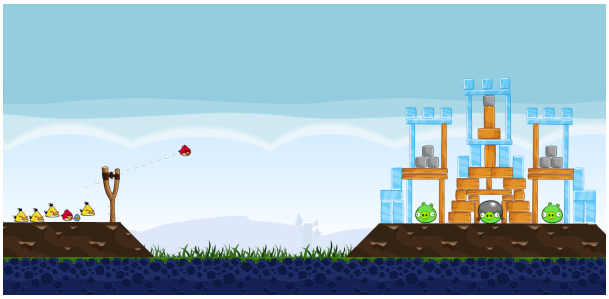


Fig. 1: Screenshot of a level from the Angry Birds game.

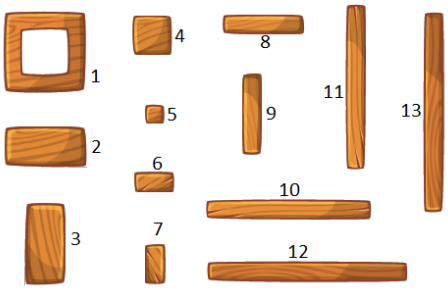


Fig. 2: The thirteen different block types available.

Algorithm 1 Structure Generation

```
currentRow ← 1
blockType ← SelectBlockType(probabilityTable)
currentStructure ← InitializeFirstRow(blockType)
while currentRow < desiredRow do
    subsets ← SubsetCombinations(currentStructure)
    blockType ← SelectBlockType(probabilityTable)
    currentStructure ← AddRow(blockType, subsets)
    currentRow ← currentRow + 1
end while
PopulateStructure(currentStructure)
EvaluateStructure(currentStructure)
```

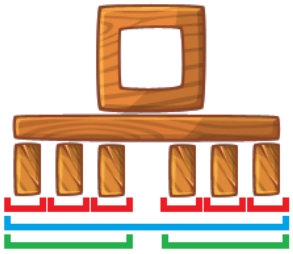


Fig. 3: The bottom row of this structure has three possible subset combinations: each block is in a separate set (red), all blocks are in a single set (blue), and the three left/right blocks are partitioned into two sets (green).

one or more structures. The ground is usually flat but can vary in height for certain difficult levels. Each block in the game can have multiple different shapes as well as being made of several possible materials.

Angry Birds is a commercial game developed by Rovio Entertainment who do not provide an open-source version of their code. Instead we use a Unity-based clone of the Angry Birds game developed by Lucas Ferreira [15], which is open-source and available to download from GitHub. This clone provides many of the necessary elements to simulate our procedurally generated structures in a realistic physics environment. There are currently eight different rectangular blocks available, of which five can be rotated ninety degrees to create a new block type. This gives a total of thirteen different block variants with which to build our structure, see Figure 2. Each block is also assigned one of three materials (wood, ice or stone), bringing the number of possible options to thirty nine.

III. Procedural Structure Generation

The proposed structure generator operates by recursively adding rows of blocks to the bottom of the already generated structure. This process continues until a desired number of rows are reached. Unlike previous methods, our structure is created using only the original block types and does not require any composite elements to be created prior to structure generation. This vastly increases the number of possible structures that can be constructed, whilst also allowing greater algorithm flexibility to satisfy conditions and restrictions which may be imposed. The complexity of a generated structure can be defined in a manner similar to that of Kolmogorov complexity [18]. The extensive amount of variation that can occur within each structure, including the number, size, orientation and positioning of blocks used, allows our generator to create a diverse range of complex structures. Algorithm 1 provides an overview of the proposed generator, with a more detailed explanation given below.

A. Structure Generation

First, a starting block type is selected at random from all possible variants. This block type will become the peak(s) of the structure, beneath which all other blocks will be placed. For our implementation up to three blocks can be placed at the top of the structure at varying distances apart, with the number of peaks being chosen at random. Initially we are only concerned about the local positions of blocks relative to each other with the world positions being calculated after the structure has been fully generated.

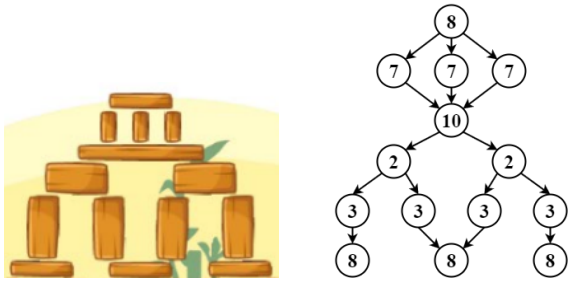
After the first row has been initialized we recursively add more rows of blocks to the bottom of the currently generated structure. The blocks at the base of the structure are split into subsets based on the distances between them. All possible subset combinations are then recorded, see Figure 3. A new block type is then selected at random. For each possible subset combination there are now three possible supporting block placement options:

- Blocks are placed underneath the middle of each subset.
- Blocks are placed underneath the edges of each subset.
- Blocks are placed underneath both the middle and edges of each subset.

All three of these possibilities are shown in Figure 4. Each of these options is created for all subsets using the selected block type, after which they are tested for validity. Any case where blocks overlap each other is deemed invalid and is removed as a possible option. In addition, each object in the structure's bottom row is tested for local support by the new

(a) (b) (c)

Fig. 4: The three possible supporting block placement options for a single block subset: middle (a), edges (b), both middle and edges (c).



(a) (b)

Fig. 5: An example of a generated structure (a) and its corresponding directed acyclic graph representation (b).

row. Each block in the bottom row of the current structure must be supported from below, either at its middle position or both of its edge positions. Any case that does not fulfil this requirement is also deemed invalid. After validity checks have been performed for all possible supporting block locations and subset combinations, one of the valid options is selected at random. If no valid options are available then a new block type is selected and the process repeated. The selected option is then used as the structure's new bottom row. This process is repeated until the desired number of rows is reached. Once the structure is complete each block is assigned a random material.

Any structure generated using this method can be depicted as one or more directed acyclic graphs, with each node representing a specific block. Each block is a descendant of the blocks that it supports (supportees) and subsequently an ancestor of the blocks that support it (supporters), see Figure 5. This can be extremely useful for other stability analysis techniques, such as identifying structural weak points [19].

B. Pig Placement

Once the structure has been fully created it is populated with pigs. First, the space directly above the middle of each block is analyzed to see if there is space for a pig to fit such that it doesn't overlap any other blocks. If this is not possible for a particular block then the positions directly above the edges of the block are checked as well. Any positions that are found to be big enough to place a pig are recorded. Next, we test all the possible ground positions that are within the structure (to a set precision). Again we check for any overlap with nearby blocks and valid positions are recorded. We then randomly choose a position from all the valid possibilities and place a pig at the given location. Any remaining pig locations that would overlap the newly placed pig are removed and another location is chosen at random. This continues until there are no more valid locations or a desired number of pigs is reached.

Fig. 6: An example structure that has local stability but is globally unstable.

This process ensures that the structure will always contain at least one pig, as a pig can always be placed on top of the structure's peak block(s).

C. Global Stability Analysis

Whilst our structure generation method ensures that each block has local stability, the global stability of the structure must be determined after its construction, see Figure 6. As all the relevant physics parameters (mass, density, friction and location) of blocks and pigs are known beforehand we can calculate the global stability of our structure exactly [20]. It is also possible to use qualitative stability analysis techniques to estimate the stability of the structure more quickly, whilst sacrificing some accuracy [21] [22]. Unfortunately, the Unity Engine upon which the Angry Birds clone is based suffers from simulation inaccuracies. These minor discrepancies cause structures which are theoretically stable to collapse within the simulation if given enough time. Currently, the only way to be certain that the structure will not collapse in this environment is to place the structure within a level and record if any blocks move a significant distance from their origin point [15]. If the structure is deemed unstable using the chosen approach then it is abandoned and a new structure is generated.

D. Structure Placement

Once the structure has been fully generated it can be placed within the Angry Birds level. For the clone implementation, levels are specified as xml files with the block and pig locations given as coordinates in world space. First, we take the bottom row of our structure and place it on top of the level's ground (the location of the ground is fixed within the level). We then continue adding additional rows on top of the structure's base until all rows have been placed. Pig locations are then converted to their corresponding world coordinates and placed within the level as well. It is also possible to place multiple structures within the same level at different locations.

IV. Fitness Function

In order to evaluate individual structures against each other we define a fitness function to measure certain desirable properties. This fitness function calculates a fitness value for a given structure, with a lower fitness value indicating a more desirable structure. A fitness function has been proposed in previous Angry Birds papers [15], [16] for a similar reason but we believe it has several limitations in its current form. The original fitness function takes into account the structure's

simulated velocity over time (used to measure the stability of the structure) as well as the number of blocks and pigs used. Our method analyzes stability outside of the fitness function, automatically rejecting a structure if it is deemed unstable. This provides the user with more freedom over which approach to use and will allow any new stability estimation techniques to integrate seamlessly with our algorithm. Our fitness function also improves upon the previous implementation by updating the analysis of certain parameters, as well as proposing some new ones of our own. These can be separated into four distinct factors, number of pigs, number of blocks, structure aspect ratio and pig dispersion; each of which can affect the fitness value of a structure. We believe that this new function provides a broader and more sophisticated analysis of the structures generated by our algorithm.

A. Number of Pigs

This is the only component of the original fitness function that has not been altered. Simply put, the more pigs that are present within a structure the more desirable the structure. $|p|$ is defined as the total number of pigs in the structure. This section of the fitness function is described by equation (1):

$$\frac{1}{1+|p|} \tag{1}$$

B. Number of Blocks

The original fitness function defined this component as the difference between the desired and actual number of blocks, divided by the difference between the maximum and actual number of blocks. While this was appropriate for simple columns of blocks it becomes very impractical when used for more complex structures. This is because the maximum number of blocks that a structure could theoretically contain grows exponentially as the number of rows increases. For example, a ten row structure generated using our method typically contains between twenty and sixty blocks, but the maximum number it could theoretically contain is 88,572 (structure with three peak blocks and each block having three supporting blocks). This means that the value for this component of the fitness function will become insignificant for any structures with a medium to high number of rows. Instead, we suggest a more suitable calculation, where the difference between the desired number of blocks B and the actual number of blocks $|b|$ is multiplied by a set factor X. This factor is used to adjust how much of an impact the difference between the desired and actual number of blocks has on the structure's overall fitness value. This section of the fitness function is described by equation (2):

$$X(\sqrt{(B-|b|)^2}) \tag{2}$$

C. Structure Aspect Ratio

One of the new components that we have added to our fitness function is the structure's width to height ratio (aspect ratio). Similar to the previous component, the maximum aspect ratio for any structure can be extremely large depending on the number of rows. This means that any attempt to normalize the ratio by dividing by the maximum would severely reduce the effectiveness of this component. Instead, we simply multiply the difference between the desired ratio R and the actual ratio $|r|$ by a set factor Y. This factor is used to adjust how much of an impact the difference between the desired and actual structure aspect ratio has on the structure's overall fitness value. This section of the fitness function is described by equation (3):

$$Y(\sqrt{(R-|r|)^2}) \tag{3}$$

D. Pig Dispersion

The other component that we have added to our fitness function is the dispersion, or spread, of pigs throughout the structure. The theory here is that structures with pigs located throughout them will be more desirable than structures with the pigs all grouped together. There are several methods that are currently available for measuring the spread of points (or in ours case pigs) throughout a 2D space.

1) Variance from center point: This method estimates the dispersion of pigs by calculating the variance for the Euclidean distance between each pig's position and the mean position of all pigs. This value is then normalized by dividing it by the length of the diagonal of the structure's bounding box.

2) Mean nearest neighbor distance: This method estimates the dispersion of pigs by calculating the mean of the nearest neighbor distances for each pig [23]. This value is then normalized by dividing it by the length of the diagonal of the structure's bounding box.

3) Morisita's index of dispersion: This method first divides the structure's bounding box into a set number Q of equally sized quadrats. The number of pigs in each quadrat n_i is then counted and used together with the total number of pigs N to calculate Morisita's index of dispersion [24], described by equation (4):

$$MI = Q(\frac{\sum_{i=1}^{Q} n_i(n_i-1)}{N(N-1)}) \tag{4}$$

4) Pig surrounding area overlap: This method was created specifically to address limitations which were identified in the previous methods and so provides a robust estimation of pig dispersion. First, the total width and height of the structure is divided by the square root of the number of pigs. A rectangle with this new width and height is then placed at the location of each pig within the structure. If none of these rectangles overlap then their total area would equal the area of the structure's bounding box. However, it is likely that some of these rectangles will overlap those that are nearby, resulting in a lesser value. The total area that all the rectangles cover is then calculated and normalized by dividing it by the area of the structure's bounding box (maximum possible area).

5) Comparison of methods: Whilst all the methods described above give suitable estimations of pig dispersion for the majority of generated structures, there are several cases where they can give unreliable results. To compare all the methods, each was tested on four different structures, see Figure 7, and the results are given in Table I.

TABLE I
COMPARISON OF PIG DISPERSION ESTIMATION METHODS

	Mean Variance	Mean Nearest Neighbor	Morisita's Index of Dispersion	Surrounding Area Overlap
Structure a	0.7314	0.0763	0.3333	0.5782
Structure b	0.3613	0.2568	0.6667	0.8908
Structure c	0.1592	0.0763	0.2778	0.3263
Structure d	0.5092	0.0763	0.5556	0.5958

In Figure 7, we can see that although the pigs are more dispersed in (b) than in (a) the mean variance from center point was higher for (a) than (b). This is because this method essentially rewards structures with pigs placed away from the center point, rather than structures with pigs dispersed throughout. A single grouping (c) would correctly give a very low dispersion value but two separate groupings results in an incorrect estimation.

In Figure 7, we can also see that although the pigs are more dispersed in (d) than in (c) the mean nearest neighbor distance is the same for both. This is because this method only uses the distance between each pig and its nearest neighbor to estimate pig dispersion. Having groupings of two pigs at multiple locations gives the same value as having all pigs at one location.

The problem with Morisita's index of dispersion is that although it gave good estimations for the structures tested, it relies on the number of quadrats to be chosen effectively. For this comparison, we created nine quadrats (3x3) but a different number of quadrats would have yielded quite a different result. This means that this method is only accurate when there are a large number of pigs available, so that each quadrat contains a sufficient number of pigs to be representationally accurate.

Our own method for estimating pig dispersion, based on measuring the overlap of each pig's surrounding area, performed well in all cases and can be normalized effectively. This method was therefore chosen to be used in our fitness function, where d defines the dispersion value. The set factor Z is used to adjust how much of an impact the dispersion of pigs has on the structure's overall fitness value. This section of the fitness function is described by equation (5).

$$Z(1-d) \tag{5}$$

E. Complete Fitness Function

The sum of all these separate components for number of pigs, number of blocks, structure aspect ratio and pig dispersion makes up the complete fitness function, described by equation (6):

$$F = \tfrac{1}{1+|p|} + X(\sqrt{(B-|b|)^2}) + Y(\sqrt{(R-|r|)^2}) + Z(1-d) \tag{6}$$

V. PROBABILITY TABLE

Instead of randomly selecting a block type during structure generation in an unbiased manner, a probability table can be used to alter the chance of a particular block type being selected. Each of the block types available is allocated a probability of being selected, with all probabilities summing to one. Whilst this probability table allows for more designer control, it can also be optimized automatically using a training algorithm and our fitness function. The training algorithm attempts to find structures which minimise the fitness function for the given parameters. Each training algorithm iteration creates nine different structures (a single generation) and uses the fitness function to rank them from most desirable ($R = 9$) to least desirable ($R = 1$). The frequency of block types in each structure is then used to update the corresponding sections of the probability table using equation (7):

$$P_i = P_i + \frac{\sum_{R=1}^{9}(S_{Ri})(R-5)}{n\sum_{R=1}^{9}(S_R)} \tag{7}$$

P_i represents the probability table value for block i, S_{Ri} represents the number of i blocks that the structure with rank R contains, S_R represents the total number of blocks that the structure with rank R contains, and n is an update factor which influences the speed at which the probability table values converge. If the probability table value for any block type is more than one then it is reduced to one. Likewise, any probability table value less than zero is increased to zero. After the probability table has been fully updated the values are renormalized so that they again sum to one. The probability table can be updated recursively over many generations using this technique.

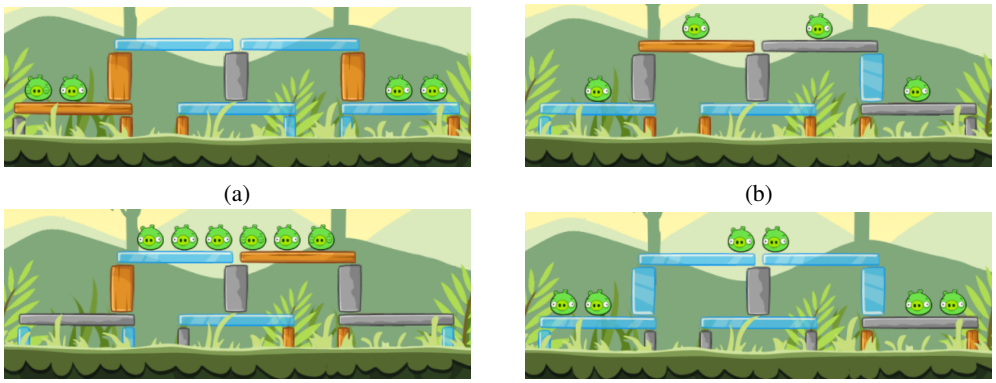
The ability to update the probability table with the fitness function can be used to provide greater direction over what types of structures are created. Each component of the fitness function can be weighted to indicate how much emphasis should be placed on each factor. This allows the user to alter the parameters of the fitness function and hence tailor the output of the structure generator to suit their needs. For example, if the user prefers structures that are tall and thin, rather than wide and short, then the desired structure aspect ratio is set very low and the corresponding section of the fitness function weighted to give more of an impact on the structure's overall fitness value. The probability table is then repeatedly updated using this fitness function, after which the mean aspect ratio of structures generated using this new probability table will be less than before. Whilst this method does not guarantee that certain requirements will be met (e.g. the structure's height must be greater than its width) it can be used to improve the probability of such a structure being created without severely restricting the generator's expressivity.

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 discussed, a probability table for block type selection can be optimized over many generations using our specified fitness function. We therefore updated our probability table over 200 separate generations, with nine structures in each generation, for a total of 1800 structures. Each structure had ten rows and for our fitness function we defined: $B = 40$,



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

Fig. 7: Four structures with the same block placement but with different pig dispersions.

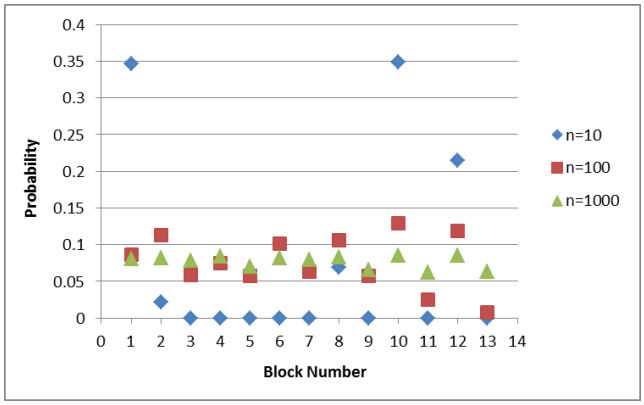


Fig. 8: Probability table values for each block type after 200 generations.

$R = 2.0$, $X = 0.01$, $Y = 0.2$ and $Z = 1.0$. We then compared three different update factors of $n = 10$, $n = 100$ and $n = 1000$, with the probability for each block type initially set to 1/13. The result of this experiment is illustrated in Figure 8.

For $n = 10$, only five block types had a probability greater than zero. These were block types 1, 2, 8, 10 and 12, with block types 1 and 10 taking almost 70% of the probability between them. This is a clear indication that the update factor is set too low, as once the probability for a block type is near zero it is very difficult for it to increase again. This places an overemphasis on the fitness function, increasing the likelihood of creating a desirable structure, but greatly reducing the range of structures that can be generated.

For $n = 1000$, the probability values changed very little even after 200 generations. This suggests that the update factor is set too high and that the probability table values are not being updated by a significant amount for each generation.

For $n = 100$, the probability values have been updated a reasonable amount but the change is not so large as to significantly reduce the structure generator's expressivity. The probability values for block types 1, 2, 6, 8, 10 and 12 increased, whilst the values for block types 3, 4, 5, 7, 9, 11 and 13 decreased.

As a result of this experiment, an optimized probability table was created using 200 generations and $n = 100$ for each of three different row values, five, ten and fifteen. These probability tables were then used when analyzing the generator's expressivity.

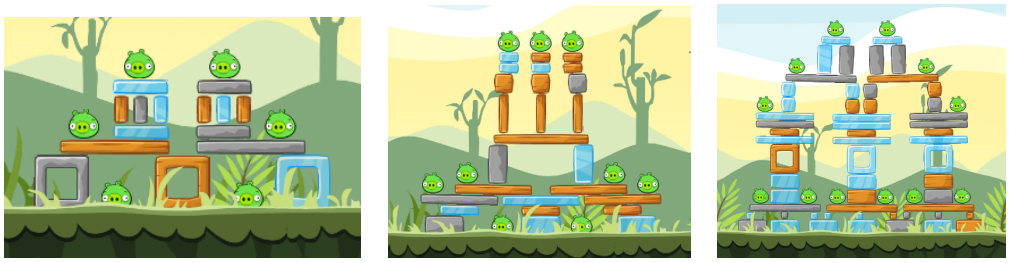
B. *Expressivity analysis*

Expressivity analysis has been described and implemented in many previous content generation papers as a means of comparing and contrasting different techniques. This is typically expressed as a metric which indicates the generator's strengths and weaknesses in various capacities. In this paper we define four measures based on metrics used in previous research [14], [15], [25]: frequency, linearity, density and leniency. Frequency evaluates the number of times that a block occurs within a structure. Linearity measures the width and height of each structure. Density provides a measure for the amount of 'free space' within a structure. Leniency estimates the difficulty of a structure, taking into account pig and block numbers. These metrics will allow our structure generator to be compared against any future methods. Presently however, there are no suitable prior algorithms with which to compare ours against.

For our experiments we generated 200 stable structures for each of three different row values, five, ten and fifteen. Each of these 200 structure groups was then sampled to find the average and standard deviation for the frequency, linearity, density and leniency. Example structures created using our generation algorithm are displayed in Figure 9.

Figure 10 shows the results of frequency sampling for structures with five rows. The average number of blocks is 12.72 with a standard deviation of 7.08. The average number of pigs is 3.07 with a standard deviation of 1.92. Figure 11 shows the frequency results for structures with ten rows. The average number of blocks is 27.39 with a standard deviation of 14.07. The average number of pigs is 4.93 with a standard deviation of 3.28. Figure 12 shows the frequency results for structures with fifteen rows. The average number of blocks is 47.07 with a standard deviation of 24.59. The average number of pigs is 7.54 with a standard deviation of 5.44.

The increase in pig numbers for structures with more rows is likely due to the increased number of blocks and hence the increased availability of viable pig locations. However, the pig



(a) (b) (c)

Fig. 9: Three example generated structures with five rows (a), ten rows (b) and fifteen rows (c).

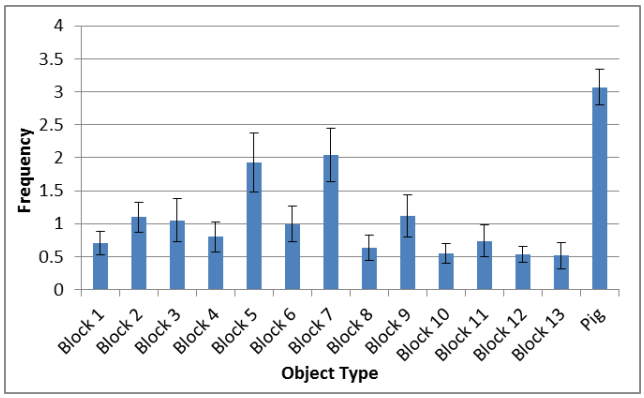


Fig. 10: Average and 95% confidence interval for block type frequency in structures with five rows.

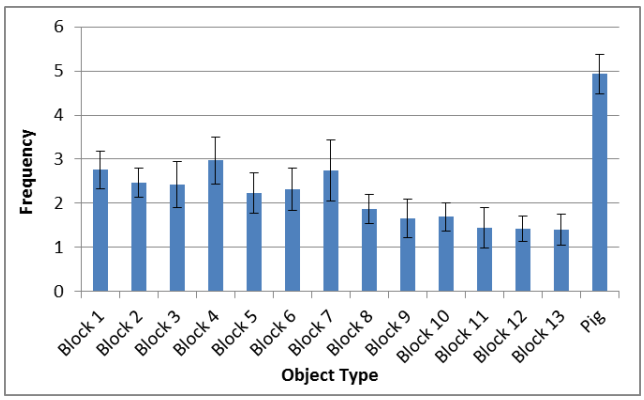


Fig. 11: Average and 95% confidence interval for block type frequency in structures with ten rows.

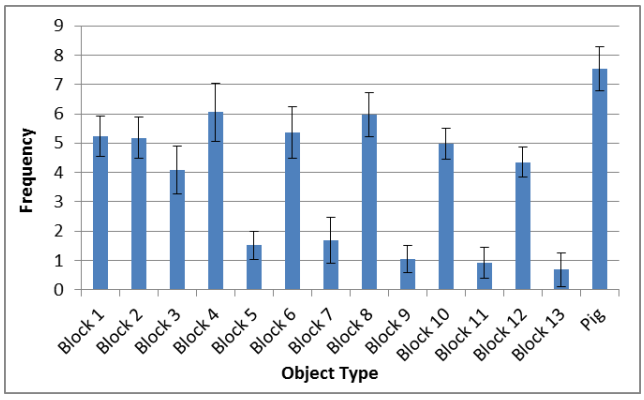


Fig. 12: Average and 95% confidence interval for block type frequency in structures with fifteen rows.

frequency relative to the block frequency was much greater for smaller structures than the larger ones. This is probably caused by the fact that the total number of pigs within a structure has a much greater impact on the fitness function for structures with a low number of blocks.

The relative frequencies of each block type also varied for structures of different sizes. Structures with fewer rows tended to favour smaller block types such as 5 and 7. This was likely due to the fact that their small width allowed more of them to fit within each row, which increased the total block count, and their small height meant that they did not decrease the structure's aspect ratio as much as taller blocks. Structures with more rows tended to favour the wider block types, as these both decreased the total block count and increased the structure's aspect ratio.

Linearity was measured using both the average width (μ_W) and height (μ_H) of all generated structures for each row amount, see Table II. The large standard deviation (σ) shows that the structures created can differ greatly in terms of their width and height, indicating a large variation in the block arrangement of the generated structures.

The density of a structure was measured by summing the areas of all blocks within the structure and dividing this by the total area of the structure itself, including all sections of empty space that it contains. The average density (μ_D) for each row amount, as well as the standard deviation (σ), is provided in Table II. The density of a structure appears to decrease as the number of rows increases, meaning that larger structures are likely to have more empty space within them and are therefore less robust than their smaller counterparts.

For many prior and current content generation methods, leniency is measured by analyzing the presence of certain objects within the subject [25], [26]. For this experiment we defined leniency using the number of pigs $|p|$ and blocks $|b|$ that are present within the structure, described by equation (8):

$$Leniency = -2|p| - |b| \tag{8}$$

Although primitive, this formula gives a rough estimate of how difficult it will be to kill all the pigs located within the given structure. The average leniency (μ_L) for each row amount, as well as the standard deviation (σ), is provided in Table II. The leniency of a structure can be seen to increase with the number of rows, due to the expanded number of blocks and pigs that are present within the structure. This

TABLE II
LINEARITY, DENSITY AND LENIENCY FOR STRUCTURES WITH 5, 10 AND 15 ROWS

Rows	Width ($\mu_W \vert \sigma$)	Height ($\mu_H \vert \sigma$)	Density ($\mu_D \vert \sigma$)	Leniency ($\mu_L \vert \sigma$)
5	2.651|1.727	2.841|0.995	0.701|0.186	−18.86|10.22
10	3.631|1.765	5.749|1.563	0.653|0.169	−37.25|17.14
15	6.349|2.450	6.353|1.274	0.612|0.126	−62.15|25.92

information can be used to influence other important aspects within the Angry Birds game, such as the number of birds provided or the ordering of certain levels.

VII. CONCLUSIONS AND FUTURE WORK

This paper has presented a search-based procedural content generation algorithm for creating complex stable structures within the video game Angry Birds. The algorithm builds structures using a top-down approach, with block types selected using a specified probability table. Each generated structure is symmetrical and can be represented as a directed acyclic graph. The structures created are populated with pig targets and analyzed for global stability. Other factors such as a varying number of peaks, multiple locations for support block placement and several possible materials, ensure that the range of possible structures is extensive and diverse.

Each generated structure is evaluated using a fitness function which considers the pig number, block number, aspect ratio and pig dispersion. This function can also be used to evolve the probability table by updating each block's chance of selection over many different generations. Each section of the fitness function can also be given a different weighting, allowing the user to define which aspects of the structure are most important.

Our structure generator was evaluated in terms of its expressivity and optimization potential. Four metrics were defined to investigate important aspects of the generated structures: frequency, linearity, density and leniency. The results of this analysis demonstrated that our structure generator can create a wide range of structures with many different attributes.

Future work could be to develop algorithms which create structures that can contain multiple block types and angles within each row. Additional research could also be conducted into estimating the number of birds required to kill all pigs within a given structure. This information could then be combined with our structure generation algorithm to create a full procedural level generator for Angry Birds.

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] A. Liapis, G. N. Yannakakis, and J. Togelius, "Optimizing visual properties of game content through neuroevolution," in *Artificial Intelligence for Interactive Digital Entertainment Conference*, 2011.

[6] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 2009, pp. 241–248.

[7] C. Browne, "Automatic generation and evaluation of recombination games," Thesis, Queensland University of Technology, 2008.

[8] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelback, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[9] V. Valtchanov and J. A. Brown, "Evolving dungeon crawler levels with relative placement," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM, 2012, pp. 27–35.

[10] L. Ferreira, L. Pereira, and C. Toledo, "A multi-population genetic algorithm for procedural generation of levels for platform games," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 45–46.

[11] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Interactive evolution for the procedural generation of tracks in a high-end racing game," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2011, pp. 395–402.

[12] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 2011, Conference Proceedings, pp. 289–296.

[13] 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.

[14] 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.

[15] 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.

[16] ——, "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.

[17] M. Kaidan, C. Y. Chu, T. Harada, and R. Thawonmas, "Procedural generation of angry birds levels that adapt to the player's skills using genetic algorithm," in *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, 2015, pp. 535–536.

[18] A. Kolmogorov, "Three approaches to the quantitative definition of information," *Problems Inform. Transmission*, vol. 1, no. 1, pp. 1–7, 1965.

[19] P. Zhang and J. Renz, "Qualitative spatial representation and reasoning in angry birds: The extended rectangle algebra," *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

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

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

[22] X. Ge, J. Renz, and P. Zhang, "Visual detection of unknown objects in video games using qualitative stability analysis," *IEEE Transactions on Computational Intelligence and AI in Games*, 2015.

[23] M. Dry, K. Preiss, and J. Wagemans, "Clustering, randomness, and regularity: Spatial distributions and human performance on the traveling salesperson problem and minimum spanning tree problem," *The Journal of Problem Solving*, vol. 4, no. 1, 2012.

[24] M. Morisita, "Measuring the dispersion of individuals and analysis of the distribution pattern," Thesis, Kyushu University, 1959.

[25] 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.

[26] D. Wheat, M. Masek, C. P. Lam, and P. Hingston, "Modeling perceived difficulty in game levels," in *Proceedings of the Australasian Computer Science Week Multiconference*. ACM, 2016, pp. 74:1–74:8.