Proceedings, The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)

# Procedural Generation of Levels for Angry Birds Style Physics Games

**Matthew Stephenson** and **Jochen Renz**
Research School of Computer Science
Australian National University
Canberra, Australia
matthew.stephenson@anu.edu.au, jochen.renz@anu.edu.au

**Abstract**

This paper presents a procedural generation algorithm for levels in physics-based puzzle games similar to Angry Birds. The proposed algorithm creates levels consisting of various self-contained structures placed throughout a 2D area. Each structure can be placed either on the ground or atop floating platforms within the available level space. These structures are created using a variety of different block types and do not require predefined substructures or composite elements. Target object locations are determined based on a combination of factors, including structural protection, occupancy estimation and overall dispersion. Experiments were performed in order to determine the ideal input parameters for generating desirable levels. The expressivity of the generator was also evaluated and the results show that the proposed method can generate a wide variety of interesting levels.

## Introduction

Procedural level generation (PLG) is one of the most popular forms of procedural content generation (PCG) and has been implemented in an extensive assortment of digital games (Hendrikx et al. 2013). PLG is defined as "the automatic creation of game levels without manual interaction" and typically requires multiple different components of a level to be dependently generated (Kerssemakers et al. 2012). PLG can be used to generate a large number of levels in a short period of time. This can greatly reduce a games development cycle and memory requirements (Dahlskog and Togelius 2012), as well as providing unique and original gameplay experiences based on the user's playstyle (Yannakakis and Togelius 2011).

Previous research into PLG has explored its applicability to many different game genres. These include platform (Mourato, dos Santos, and Birra 2011), racing (Cardamone, Loiacono, and Lanzi 2011), role-playing (Valtchanov and Brown 2012), arcade (Cook and Colton 2011), stealth (Xu, Tremblay, and Verbrugge 2014), roguelike (Stammer et al. 2015) and real-time strategy (Lara-Cabrera et al. 2015). Several papers have also explored the use of PLG for physics-based puzzle games, most notably for the Cut the Rope (Shaker, Shaker, and Togelius 2013a; 2013b; Shaker et al. 2015) and Angry Birds games (Ferreira and Toledo 2014a; 2014b; Kaidan et al. 2015; 2016). The physics constraints employed in these types of games create many problems for PLG and makes 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 (Shaker et al. 2013).

This paper presents a procedural level generator for physics-based puzzle games similar to Angry Birds. Although the proposed generator is designed specifically for the Angry Birds elements and environment, the techniques used can be applied to many other similar games. Examples of such games include Crush the Castle, Fragger and Siege Hero, all of which share the same general level design and play style as Angry Birds. Several different level aspects are considered by our generator, including structure generation, structure placement, target placement, support analysis and bird selection.

Previous implementations of PLG for Angry Birds have been very limited in terms of what they have been able to achieve. These prior methods can only generate simple levels, containing columns of either single objects or small predefined structures (Ferreira and Toledo 2014b; 2014a). Several attempts have been made to improve this approach by adapting levels to the player's skill (Kaidan et al. 2015) and increasing the number of composite elements (Kaidan et al. 2016). However, even with these alterations the complexity of the generated levels is still relatively low. In contrast, our proposed PLG can create a broad range of levels, containing a wide assortment of complex and novel structures.

Several experiments were conducted to analyze the expressivity of our level generator and to determine its capabilities. Metrics such as frequency, linearity, density, leniency and playability, were used to describe the characteristics of the generated levels. The stability of generated structures for different widths, heights and compositions was also investigated.

## Angry Birds Level Overview

Angry Birds levels consist of several different components. On the left side of the level there is a slingshot and a number of birds which can be thrown by it. On the right side there

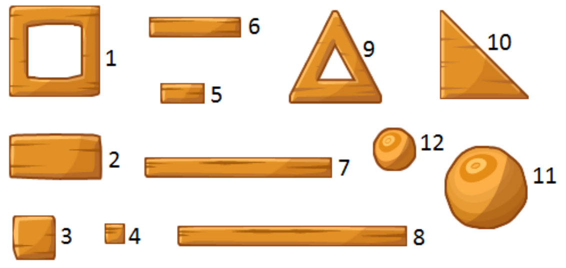

Figure 1: The twelve different blocks available.

are various blocks, platforms and pigs, usually arranged into an interesting design. The objective of any given level is to kill all the pigs using the birds provided. The source code for the official Angry Birds game is not currently available, so a Unity-based clone created by Lucas Ferreira was used (Ferreira and Toledo 2014b).

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 which can be moved apart from a bird or pig. There are currently twelve different blocks available within the unity clone, see Figure 1. 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. We also define the concept of "level space" which is a pre-defined area of the level, within which blocks, platforms and pigs can be placed. This level space is used to prevent objects being placed too close to the slingshot, below the ground, or outside of the camera's view. The positions of the slingshot and ground are fixed within a level and all other objects are placed relative to these two locations.

## Proposed Level Generator

The proposed level generator creates Angry Birds levels consisting of a collection of independent structures. These structures are distributed throughout the available level space, either on the ground or atop floating platforms. The number of ground and platform structures can be decided either manually or by random selection. Before discussing the placement of these structures within a level it is necessary to first explain how these structures are created.

### Structure Generation

Within our level generator there is a self-contained structure generator which creates complex structures 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. Structures generated using our algorithm are made up of rows, with each row consisting of a single block type. Each block is also randomly assigned one of three possible materials (wood, ice and stone).

Each structure is generated to fit within a certain area of the overall level. This area is used to define the maximum width and height values that the generated structure can possess. A probability table is also 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.

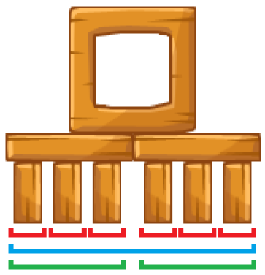
Figure 2: The bottom row of this structure has three possible subset combinations: each block is in a separate set (red), all blocks are in a single set (blue), and the three left/right blocks are partitioned into two sets (green).

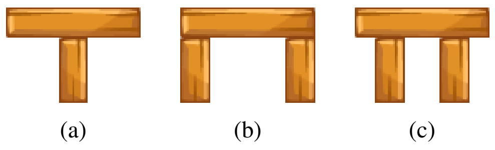

Figure 3: The three supporting block placements for a single block subset: middle (a), edges (b), mid-points (c).

First, an initial block type is selected at random using the probability table. This block type will become the peak(s) of the structure, beneath which all other blocks will be placed. Any number of peaks can be chosen, either manually or randomly, as can the distance between each of them. However, if the area taken up by all of the peak blocks fails to satisfy the structure's maximum width or height limits, then the peak combination will be declared invalid and a new arrangement will be chosen. This process continues until a suitable selection is made.

After this first row has been initialized, additional rows of blocks are recursively created which will be placed underneath the already 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 recorded, see Figure 2. A block type for the new row is then selected using the probability table. For each subset combination there are three possibilities for placing supporting blocks:

- Blocks are placed underneath the middle of each subset.
- Blocks are placed underneath the edges of each subset.
- Blocks are placed underneath the mid-points between the middle and edges of each subset.

All three of these possibilities are shown in Figure 3. These three choices can also be combined to make a total of seven different options. Each of these options is created for all subset combinations using the selected block type, after which they are tested for validity.

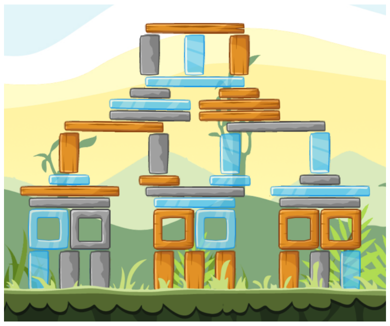

Figure 4: An example of a fully generated structure.

Any case where blocks overlap is deemed invalid and removed as a possible selection. It is also important that the blocks at the bottom of the already generated structure are supported by this new row. The level of required support can be set to one of three settings. The first is that each block must be supported either at its middle position or both of its edge positions. The second is that each block must be supported at both of its edge positions. The third is that each block must be supported at its middle position and both of its edge positions. Any case that does not fulfil the chosen support requirement is deemed invalid.

After validity checks have been performed for all subset combinations and supporting block placements, a valid option is selected at random from all possibilities. If no valid options are available then a new block type is chosen and the process repeated. The selected option is then used as the structure's new bottom row. This process continues until the width or height of the structure is greater than its maximum width or height values respectively. Once this occurs the last row that was added is removed, after which the structure is complete. This process ensures that the generated structure will fit within the dimensions specified. An example of a fully generated structure is shown in Figure 4.

## Structure Placement

Structures within an Angry Birds level can be placed either on the ground or on a platform. The desired number of structures for each of these options can be defined by the user but size restrictions mean that this may not always be possible. Ground structures are placed first followed by platform structures.

The available ground space within a level is divided into randomly sized sections, with the number of sections equal to the desired number of ground structures. Whilst these sections can theoretically be any size it is useful to employ a minimum size limit. This prevents sections from being too small which restricts the complexity of the generated structures. A structure is then generated for each ground section. The maximum width of each structure is equal to the width of its section and the maximum height of the structure is set to two thirds the total height of the level space.

After all ground structures have been generated the platforms are placed within the level. Platforms are made up of square blocks which are not subject to the same physics as other objects and are instead fixed in place. The size of a platform is determined by the number of blocks that are used to create it. All platforms have a height of exactly one block but the width of a platform can vary. Like the ground sections, it is useful to set minimum and maximum size limits on the platforms created. Each platform's location is determined randomly within the level space. The location is deemed valid if the following holds true: the platform does not overlap any other platform or ground structure, the platform is not too close to the top of the level space, if the platform is placed above or below another platform then it should not be too close to that platform. These last two requirements ensure that all platforms have enough space above them to generate a complex and interesting structure. Additional checks are also performed to ensure that platforms do not block off any sections of the level. Depending on the desired number of platform structures and the size of the ground structures, it may not be possible to fit all the necessary platforms within the available level space. Each platform is therefore given a maximum number of placement attempts. If a suitable location for a platform cannot be found after this many attempts then it is disregarded. This means that the actual number of platforms within a level may be lower than what was originally requested.

A structure is then generated for each successfully placed platform. The maximum width of each structure is equal to the width of its platform and the maximum height of the structure is the vertical distance between the platform and either the top of the level space or any platform located above it (whichever is smaller).

## Pig Placement

Once all structures have been placed within the level they can be populated with pigs. Each structure is analyzed for possible pig locations using the following method. First, the spaces directly above the middle and edges of each block within the structure are analyzed to see if there is space for a pig to fit such that it doesn't overlap any blocks or platforms. Any positions that are deemed large enough to support a pig are recorded as valid pig locations.

Next, the positions that are either on the ground or on platforms, which are also within a structure (to a set precision), are tested. A position is defined as within a structure if there are blocks to its left and right that both belong to the same structure. Again, a check for any overlap with nearby blocks is carried out and valid locations recorded.

Once all valid pig locations have been identified they are ranked based on a combination of factors. The first factor ($f_1$) is the structural protection that the pig is offered with respect to the blocks surrounding it. Pigs that are placed within a structure have greater protection from incoming shots than those outside it. The degree of protection that a pig location has is calculated as the minimum number of blocks to its left ($b_l$), right ($b_r$) or above ($b_a$), that are all associated with the same structure as the pig location. This value is then multiplied by a set weighting ($X$).

$$f_1 = X(min(b_l, b_r, b_a)) \tag{1}$$

The second factor ($f_2$) is the overall dispersion of pigs throughout the level. Levels with pigs spread throughout them are typically preferable to levels with pigs grouped together. The dispersion value for a pig location ($p_l$) is calculated as the product of the Euclidean distances between itself and all the pig locations which have already been selected ($p_s$). This value is then multiplied by a set weighting ($Y$).

$$f_2 = Y \prod_{p_x \in p_s} \overline{p_l p_x} \tag{2}$$

The final factor ($f_3$) is occupancy estimation and is based on a technique called occupancy-regulated extension (Mawhorter and Mateas 2010). If a pig 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 ($Z$) (otherwise $f_3 = 0$). This is because one of the key features within Angry Birds is the ability to kill pigs with falling blocks, rather than with birds alone. Pigs that are placed below or near other blocks which may potentially fall and kill them provide the user with this alternative choice of action. Pigs that are situated below the edges of platforms are particularly vulnerable to this kind of attack.

The sum of all three of these factors gives a fitness value for each pig location. All pig locations are ranked using their fitness values, with a higher fitness value indicating a more desirable location.

After all valid pig locations have been ranked the pigs are placed within the level. The desired number of pigs within the level can be decided either manually or by random selection. The location with the highest ranking is chosen and a pig is placed at the specified position. Any previously valid pig locations that would overlap the newly placed pig are removed. The remaining pig locations are then re-evaluated and the highest ranked position is again selected. This process continues until the desired number of pigs is reached or there are no more valid pig locations.

If the desired number of pigs has still not been reached, even after exhausting all valid pig locations, then additional pigs are added as follows. A ground position is chosen at random and analyzed to see if there is space for a pig to be placed there. If there is then the pig is placed, otherwise a new location is randomly selected. This continues until the desired number of pigs is reached or a maximum number of attempts is reached.

## Irregular Block Placement

After pig locations have been finalised, attempts are made to place irregular blocks throughout the level. Block 10 can be rotated 90 degrees to form a new block shape, bringing the total number of irregular block types to five. These blocks are placed in a similar fashion to the pigs. Valid locations are determined for each of the block types, both on top of blocks and on the ground or platforms within a structure. As block 10 is not vertically symmetrical it must also be supported such that it will not fall over. It can therefore only be placed on blocks that are wide enough to support it. After all valid locations have been identified for all block types, a specific block type and location is selected at random. Much like the regular blocks, the chance of selecting each block type is specified in a probability table. Any remaining locations that would overlap this selected block are removed as valid possibilities. This continues until no more valid options remain.

## Structural Weak Points

The concept of a weak point for a structure is any block that, if removed, would cause a large number of other objects (blocks or pigs) to be affected (Zhang and Renz 2014). Levels that are intended to be difficult to solve can attempt to shield these particular blocks from the user's shots. This protection can also reduce the effectiveness of greedy shots and requires the user to plan their actions carefully.

To identify weak points, every block within the generated level is first tested to see if it is ”reachable”, i.e. it can be hit directly with a bird fired from the slingshot. Every reachable block is then given a score based on the number of objects that will be affected by its removal. If the removal of a block violates another object's local support requirements then we say that this object has been affected. An object is also affected if its local support requirements would be violated by the removal of any other affected objects. Affected blocks add one to the score, whilst affected pigs add ten. If the score for any reachable block is greater than a set threshold ($W$), then the block is classified as a weak point.

The proposed level generator can attempt to protect a weak point using a variety of methods. Firstly, if the weak point is part of a ground structure and there is sufficient space to the left of the structure, then a stack of randomly chosen blocks (selected using the probability table) is placed to the left of the structure. This stack is recursively built one block at a time until either the weak point is no longer reachable, or any of the blocks in the stack overlap other objects, at which point the last added block is removed. Secondly, if the weak point is part of a supporting block arrangement where positions for other support blocks are available, then additional support blocks are added if there is sufficient space. Lastly, the material of the weak point can be set to stone, as this increases the block's overall durability.

## Bird Number Selection

The number of birds that are provided is very important to a level's integrity, as this determines how difficult the level will be to complete. If the number of birds is too low then the level will be extremely challenging, perhaps even impossible. Conversely, if the number of birds to too high then the level will be too easy.

Selecting the number of birds ($b$) is based on a formula which takes into account the number of pigs ($p$) and structures ($s$) that are present within the level:

$$b = \begin{cases} \lceil \frac{p}{2} \rceil & \text{if } s < \lfloor \frac{p}{2} \rfloor \\ \lceil \frac{p}{2} \rceil + 1 & \text{otherwise} \end{cases} \tag{3}$$

In words, this means that the number of birds is equal to half the number of pigs (rounding up) plus an additional bird if the number of structures is greater than or equal to

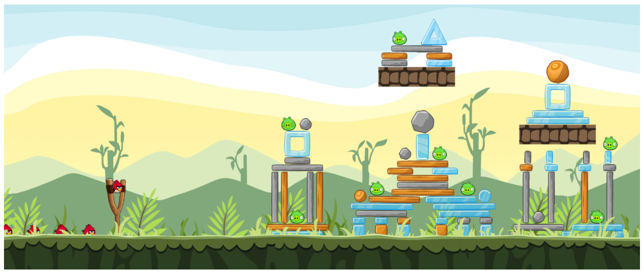

Figure 5: An example of a fully generated level.

this value. An additional bird can then be added again if the level is intended to be easy, or removed if the level is intended to be difficult.

After selecting the number of birds the level is complete. An example of a fully generated level is shown in Figure 5.

## Experiments and Results

Two studies were conducted to analyze the stability of the generated structures and evaluate the overall expressivity of our level generator.

### Stability

The stability of the structures created by our generator is a critical factor that influences the quality of the levels produced. Structures that cannot support themselves will fall down once the level is initialized and severely reduce its overall appeal. There are currently three different support options that can be used to alter the stability requirements for the structures created. The option chosen determines the level of support that is needed by each block within the structure. Several tests were carried out to determine if the support requirement, as well as the width and height, of a structure was a good indication of its stability.

The first test was carried out using the requirement that each block must be supported either in its middle position or both of its edge positions. 100 structures were generated, with the width and height limits for each structure selected randomly. All blocks had an equal chance of being selected and blocks with two possible block types (different rotations) had their selection probability split evenly between them. The results of this experiment are illustrated in Figure 6. The average width and height of each stable structure was 4.65 and 4.39 respectively. The average width and height of each unstable structure was 3.73 and 5.37 respectively. This result demonstrates that structures which are short and wide are more likely to be stable than structures which are tall and thin. Of the 100 generated structures 74 of them were stable and 26 were not.

Whilst it is possible to increase the likelihood of a generated structure being stable by implementing a separate stability analysis method, the engine within which the level is eventually placed will likely suffer from simulation inaccuracies. It is therefore not possible to guarantee the stability of a generated structure using this support requirement.

The remaining two support requirements generated no unstable structures, but had different effects on the qualities of

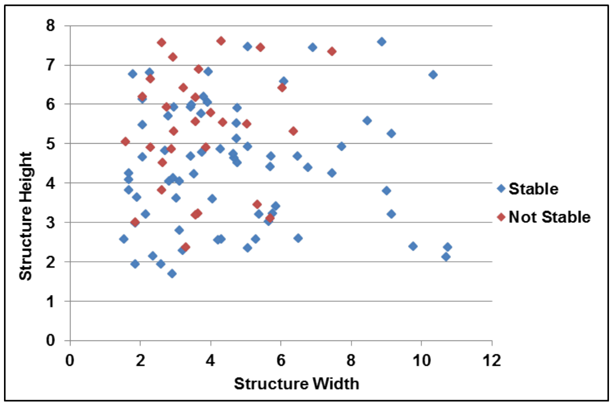

Figure 6: Width and Height values for 100 generated structures.

the structures generated. The second of the three options has the requirment that each block must be supported at both of its edge positions. This requirement guarantees that the generated structure will be stable but results in a lower number of structure possibilities than the first option. The third option has the requirment that each block must be supported at its middle position and both of its edge positions. This additional restriction increases the overall robustness of the generated structures but further decreases the number of structure possibilities.

Out of all three of these support options we would therefore recommend the second. The first option provides the most variety in structure generation but cannot guarantee the stability of the structures created. The third option (like the second) guarantees the stability of the generated structures, but restricts the algorithms expressivity and reduces the amount of free space within each structure. As a result of this analysis, the second support option was used when evaluating the level generator's expressivity.

### Expressivity Analysis

The expressivity of a level generator is the space of all levels it can generate and is measured by evaluating different aspects of a level to identify its strengths and weaknesses. Several metrics have been proposed to analyze a generator's expressivity (Smith and Whitehead 2010; Smith et al. 2011; Horn et al. 2014; Snodgrass and Ontanon 2015): frequency, linearity, density, leniency and playability. For our experiments we generated 200 levels, each containing three ground structures, two platform structures and eight pigs. For pig placement we defined $X$=3.0, $Y$=0.002, $Z$=1.0 and $D$=0.8. For identifying structural weak points we defined $W$=30. All blocks had an equal chance of being selected and blocks with two possible block types (different rotations) had their selection probability split evenly between them.

**Frequency** Frequency evaluates the number of times that a block type occurs within a level. Figure 7 shows the average frequency of each block type within a level (block types with an r-subscript indicate blocks that have been rotated ninety degrees). Even though each block had an equal

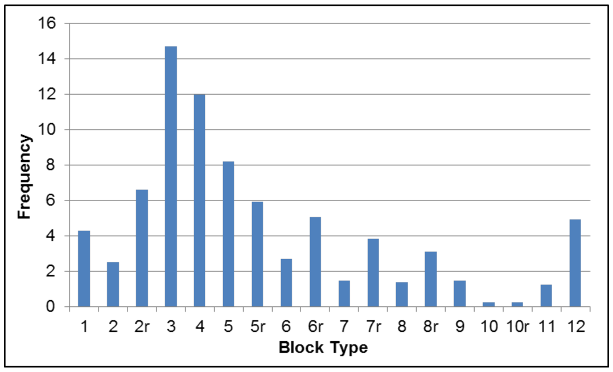

Figure 7: Average frequency for each block type.

chance of being selected we can see that wide blocks appeared less frequently than thin blocks. The same can also be said about most of the regular block types and their rotated counterparts. 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 fulfil these conditions and so are placed more frequently. It is also apparent that short block types are chosen more frequently than tall ones. This is likely due to the size restrictions imposed on the structures created. Once a structure exceeds its maximum width or height, the last row that was added is removed. Selecting tall or wide blocks are more likely to push the structure's dimensions past these limits and so are less likely to be included in the final structure. Both of these issues could be easily rectified by increasing the probability of larger block types being selected.

**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 center points of all blocks, platforms and pigs as our data points. Each level is then scored based on its $R^2$ value. The average linearity of a generated level is 0.0462, with a standard deviation of 0.0439. This result shows that our levels are highly non-linear, with objects being distributed throughout the entire level space.

**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 and pigs within the level space. This is then divided by the total size of the level space to give a percentage indicating how much of the level's area was taken up by objects. The average density of a generated level is 24.3%, with a standard deviation of 4.26%. 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.

**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 a level is estimated using the number of pigs and structures that are present. This is then used to determine the number of birds that are provided to the player. Therefore, the leniency of a level is entirely dependent on the genertor's input parameters.

**Playability** Playability is used to represent whether a level is solvable. Due to the exceptionally large state and action space, it is difficult to determine if a level can be completed. Several AI agents that are designed for playing Angry Birds were employed, but the results proved unreliable. An AI agent can be used to confirm that a level is solvable but not that it is unsolvable. Although every generated level should be solvable using an infinite number of birds, whether or not a level can be solved using the birds provided remains unknown.

## Conclusions and Future Work

This paper has presented a procedural generation algorithm for creating complex and interesting levels in physics-based puzzle games similar to Angry Birds. The algorithm constructs these levels by generating a collection of independent structures and arranging them throughout the available level space. These structures are created using a variety of different block types and can be demonstrated to be structurally stable. Additional 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. The levels are then populated with target objects (pigs) and other additional block types. Structural weak points are identified and can be protected using a variety of methods. The number of attempts to solve the level (number of birds) is then chosen based on a combination of factors.

The proposed level generator is also highly customisable. Many different aspects can be defined by the user, such as the number of ground and platform structures, number of pigs, block selection probabilities, structural support requirements, pig placement parameters and many others. This allows the level generator to be tailored to any purpose and it can be used to create levels for a variety of situations. The generator is also flexible enough that it can be applied to many other games apart from Angry Birds.

Our proposed level generator was evaluated in terms of its expressivity using a wide variety of metrics: frequency, linearity, density, leniency and playability. 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 variety of future possibilities for this research. One example could be to develop more sophisticated methods for structure generation, creating structures that can contain multiple block types and angles within each row. Additional studies could also be carried out into intelligent material selection or playability analysis. Work could also be performed on creating an algorithm that can generate levels using a limited supply of objects or other similar restrictions.

## References

Cardamone, L.; Loiacono, D.; and Lanzi, P. L. 2011. 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*, 395–402. ACM.

Cook, M., and Colton, S. 2011. Multi-faceted evolution of simple arcade games. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 289–296.

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

Ferreira, L., and Toledo, C. 2014a. Generating levels for physics-based puzzle games with estimation of distribution algorithms. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, 25:1–25:6. ACM.

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

Hendrikx, M.; Meijer, S.; Velden, J. V. D.; and Iosup, A. 2013. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.* 9(1):1–22.

Horn, B.; Dahlskog, S.; Shaker, N.; Smith, G.; and Togelius, J. 2014. A comparative evaluation of procedural level generators in the mario ai framework. In *Foundations of Digital Games 2014*, 1–8.

Kaidan, M.; Chu, C. Y.; Harada, T.; and Thawonmas, R. 2015. 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)*, 535–536.

Kaidan, M.; Harada, T.; Chu, C. Y.; and Thawonmas, R. 2016. Procedural generation of angry birds levels with adjustable difficulty. In *Proceedings of the IEEE World Congress on Computational Intelligence*.

Kerssemakers, M.; Tuxen, J.; Togelius, J.; and Yannakakis, G. N. 2012. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 335–341.

Lara-Cabrera, R.; Nogueira-Collazo, M.; Cotta, C.; and Fernndez-Leiva, A. J. 2015. Procedural content generation for real-time strategy games. *International Journal of Interactive Multimedia and Artificial Intelligence* 40–48.

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

Mourato, F.; dos Santos, M. P.; and Birra, F. 2011. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 8:1–8:8. ACM.

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

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

Shaker, N.; Shaker, M.; and Togelius, J. 2013a. 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*, 72–78.

Shaker, N.; Shaker, M.; and Togelius, J. 2013b. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 215–216.

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

Smith, G.; Whitehead, J.; Mateas, M.; Treanor, M.; March, J.; and Cha, M. 2011. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on Computational Intelligence and AI in Games* 3(1):1–16.

Snodgrass, S., and Ontanon, S. 2015. A hierarchical mdmc approach to 2d video game map generation. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 205–211.

Stammer, D.; Mannheim, H.; Gnther, T.; and Preuss, M. 2015. Player-adaptive spelunky level generation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 130–137.

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

Xu, Q.; Tremblay, J.; and Verbrugge, C. 2014. Generative methods for guard and camera placement in stealth games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 87–93.

Yannakakis, G. N., and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3):147–161.

Zhang, P., and Renz, J. 2014. Qualitative spatial representation and reasoning in angry birds: The extended rectangle algebra. In *Knowledge Representation and Reasoning Conference*.