Tanager: A Generator of Feasible and Engaging Levels for Angry Birds

Lucas N. Ferreira and Claudio Fabiano Motta Toledo

***Abstract*—Generating feasible levels for Physics-based puzzle games is a complex and time-consuming task. This is because the mechanics of these games are based on realistic physics, so simulations are required to evaluate the playability of the generated levels. Recently, a few generators have been able to produce considerably complex levels in the context of the Angry Birds game, a very famous title of this genre. However, none of these generators are able to guarantee the playability of their produced levels. This paper presents Tanager, a level generator based on a genetic algorithm that is capable of producing feasible levels for the Angry Birds game. Evaluating playability in this game requires checking both the stability of the stacked blocks and the possibility of killing all the pigs with the given amount of birds. These two components are handled by the algorithm through a simulation. The first one is calculated by measuring the overall velocity of the blocks and the second is defined by an intelligent agent which plays the levels. Three sets of experiments are conducted to evaluate Tanager. The first one measures the performance of the genetic algorithm underneath Tanager. The second one explores the expressivity of the generated levels considering their structural characteristics. The third one measures design quality of levels via an on-line user study. Results show that Tanager is capable of generating a considerable variety of feasible levels that are as engaging and enjoyable as those manually designed. However, the generated levels are less challenging than the hand-authored ones.**

***Index Terms*—Procedural content generation, genetic algorithm, optimization, intelligent agent, immersion, physics-based puzzle games.**

I. Introduction

PROCEDURAL level generation (PLG) is the process of designing a game level automatically or semi-automatically via algorithms [1]. It has been studied in the context of several different video game genres such as platformers [2], real-time strategy (RTS) [3], first person shooters (FPS) [4], racing [5], roguelike [6] and stealth [7]. The main goal of a level generator is usually to create an expressive amount (which covers a relevant part of the space of levels) of feasible and interesting levels [1]. In most of these genres, evaluating feasibility is relatively simple and can be solved with path-finding algorithms [8]. For example, it is possible to run A* in Super Mario Bros to discover if the player can traverse from the initial to the final position of a level [9].

Lucas N. Ferreira was with the Institute of Mathematics and Computer Science, University of Sao Paulo, Sao Carlos, SP, 13566-590, Brazil e-mail: (see http://www.lucasnferreira.com).

Manuscript received April 19, 2005; revised August 26, 2015.

Generating feasible levels is a more complex task in the under explored genre of physics-based puzzle games. In this genre, different rigid and soft bodies are used to create puzzles that have to be solved considering forces like gravity and other physics constraints. Thus, generators have to decide where to place such bodies in a given space in order to create interesting feasible puzzles. Moreover, bodies can be either static or dynamic. Static objects do not move and are not affected by forces or collisions. Dynamic objects can move and are affected by the physics of the game. A level is considered feasible if the dynamic bodies are initially stable and if there is a set of valid moves that let the player solve it. Levels must be initially stable to guarantee that the objects will not move before any player action. The difficulty to evaluate the feasibility of a level is directly related to the type (static or dynamic) and amount of game objects forming the geometry and the challenges of the level.

Few papers [10], [11], [12] have explored how to generate levels for physics-based puzzle games, specially for Angry Birds [13]. Although some generators are capable of producing levels with complex structures, they do not guarantee the playability of the levels. Also none of these papers evaluated the design quality of the generated levels. Thus, the main goal of this paper is to generate feasible and interesting levels for the Angry Birds game, evaluating them with players in order to measure their design quality. The proposed generator is called Tanager and it is based on a genetic algorithm (GA).

Tanager is built upon and extends the work described in [10], which introduced a GA focused on creating stable stacks of blocks for the Angry Birds game. Tanager improves the level representation in order to evolve more complex structures as well as the number of birds in the level. The initialization method previously designed is also extended to improve the stability of the structures during the evolutionary process. A new fitness function is introduced to evaluates the playability of the generated levels. Thus, the main contributions of this work are the following:

1) An advanced level representation that supports the evolution of more complex blocks structures as well as the amount of available birds.
2) An initialization method that generates a population of levels with high likelihood of being stable.
3) A simulation-based fitness function, which applies gameplay metrics from an intelligent agent, to penalize infeasible levels.

4) An user study to analyze the design quality of the generated levels.

In this user study, we consider that hand-authored levels from the original game have good design quality, so we discuss the similarity between generated levels and hand-authored ones according to difficulty, engagement and enjoyment. Tanager was also evaluated according to performance and expressivity. Results show that it is capable of generating stable and playable levels that are as engaging and enjoyable as those manually designed.

The remainder of the paper is organized as follows: Section II provides an overview of PLG for physics-based puzzle games, including the previous method [10] that Tanager is built upon. Section III introduces the design of Tanager and how it extends the work of [10]. Section IV reports the results of the experiments, including the user study evaluating the design quality of the generated levels. Finally, Section V concludes this paper with a discussion of the results achieved and future works.

II. Procedural Level Generation in Physics-based Puzzle Games

Most of the work in PLG for physics-based puzzle games has been conducted in the context of the Angry Birds game [10], [14], [15], [11], [12], however a few other works also used Cut the Rope as testbed [16], [17]. An Angry Birds level is composed of a stable pile of blocks which contains pigs inside a shown in Figure 1. The objective of the game is to kill all pigs with a limited amount of birds. Despite the simplicity of the game, generating feasible levels in this case is a quite complex problem, which is composed of two parts: placing a stable structure of blocks with pigs inside and adding a set of birds that allows the player to kill all pigs. The first method that attempted to approach this problem is presented in [10], which is the base of Tanager and it is described next.

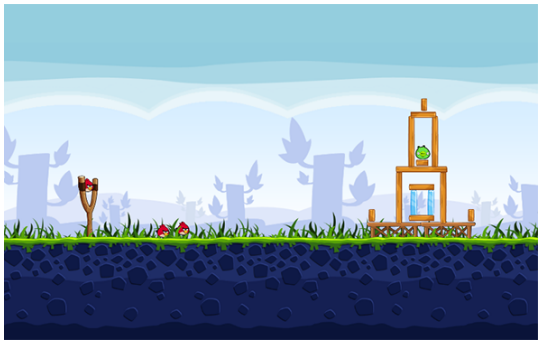


Figure 1: Instance of Angry Birds level extracted from the original game with its main game elements: slingshot, birds, pile of blocks and pigs.

A. The Previous Generator

The previous generator [10] is a GA where a level is encoded as a genotype composed of stacks of blocks. A block can be either elementary or composed, where elementary blocks are unitary pieces connected to form composed ones. Each block, elementary or not, is represented by an integer value and a stack is structured as an array of blocks (array of integers). The whole level, in turn, is encoded as an array of stacks, i.e., a structure compounded by arrays of blocks. The amount of stacks can be different for each level and the stacks sizes can change within a level. The distances (in pixels) between each stack is also encoded in this representation as shown in Figure 2. The desired amount of birds in the level is a parameter that is not evolved by the GA in [10] and hence it is not encoded in the level representation.

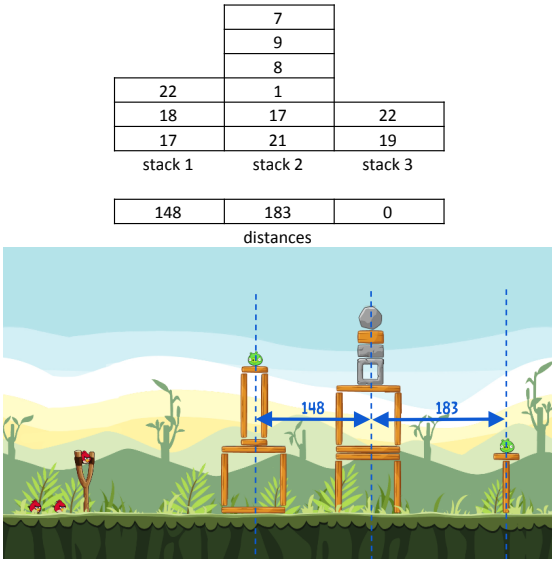


Figure 2: Level representation of the GA that Tanager is built upon [10]. Each block as well as the distances between the stacks are represented by integer numbers. In this example, the first stack has three blocks (17, 18 and 22), the second one has six blocks (21, 17, 1, 8, 9 and 7) and third one has two (22 and 19). The distances from stack 1 to stack 2 is 148 and from stack 2 to stack 3 is 183. Both the distances are measured in pixels.

The initial population is created using a probability table, which defines the probability of a block to be selected in a certain part of the level (ground, middle or top). Table I is an example of such probability table. The idea behind this table is that some blocks easily harm the stability of the stacks when placed in certain parts of a stack. For example, placing a triangular block in the first position of a stack makes it very unstable. The first step of the initialization process consists of randomly defining the amount of stacks to be placed. After that, each element of each stack is randomly sampled using the probability table. Finally, random distances $0 \leq d \leq 300$ (in pixels) are assigned in between the stacks using an uniform distribution.

To decode this representation into an actual level, all the stacks are placed on the ground starting from a given coordinate x to the right of the slingshot. The

Block	Ground	Middle	Top
1	0.2	0.2	0
2	0.2	0	0.1
3	0	0.2	0.0
4	0.1	0.2	0.0
5	0.1	0	0.05
6	0.2	0	0.05
7	0.1	0	0.0
8	0	0.2	0.4
9	0	0.2	0.4
10	0.1	0	0.0

Table I: Probability table used by the previous level generator. The amount of rows in the table depends on the amount of building blocks used by the GA.

x_0 coordinate of the first stack is exactly x and the the x_i coordinate of a stack $i > 0$ is calculated adding the distance from the stack i to the stack $i-1$ to x_{i-1}. A stack is built placing the first block on the ground and then stacking the remaining ones on top of each other using the same x_i position.

The fitness function penalizes unstable levels using the sum of the magnitude of each block's velocity vector (the less the total velocity, the less penalization). The fitness function also rewards the levels with at least one pig as well as an amount of blocks desired by the user. Although this generator is able to create stable levels, there is no guarantee that they are playable. Also, the structures of blocks generated are quite simple and no experiments were conducted to evaluate the design quality of the levels.

B. Other Generators

The authors of [11] extended the GA proposed in [10] by adding the Extended Rectangle Algebra (ERA) [18] to calculate the difficulty of the generate levels. The same level encoding was used, but the fitness function was modified to evaluate how hard the evolved levels are. This allows the user to set previously the desired difficulty of the levels that must be evolved by the GA. The method was validated with human players, but a deep analysis of final design quality was not presented. Tanager is also based on the GA proposed in [10], but it is different from [11] because it has an advanced level representation allowing duplicated blocks. Moreover, Tanager's fitness function uses an agent to define the feasibility of a level.

Another level generator was proposed by [12] which can create complex levels using a constructive approach. The algorithm generates levels by first creating structures composed of elementary blocks and then placing these structures throughout the space of the level. The structures are composed of rows, which are recursively generated using a probability table that determines the likelihood of a block type being selected. The generated structures are then placed either on the ground or on floating platforms. The structures are populated with pigs after being placed in the level. This approach does not guarantee global stability for the generated levels, therefore, it was evaluated according to the frequency of stable levels generated. An evaluation of the design quality is also not conducted by the authors in [12]. The main difference between [12] and Tanager is the level representation. In [12], levels can have structures more general than the ones generated by Tanager. However, Tanager guarantees both the global stability and playability of the levels, which is not guaranteed by [12] .

There are few other works approaching the problem of PLG for physics-based puzzle games using the Cut the Rope game as testbed [16], [17]. In [17], a method based on Grammar Evolution generates levels using a fitness function that tries to find the best placement of game objects according to the rules of the game. The method is evaluated according to expressivity and no design quality analysis is conducted. One contribution of [17] is a first attempt to evaluate playability of procedurally generated levels for physics-based puzzle games. The method presented in [17] can't be directly applied for level generation in the Angry Birds game. However, Tanager's approach for evaluating feasibility is similar to [17] in the sense that both use an agent to evaluate feasibility.

III. The Tanager Level Generator

Tanager is described through this section, where the main points distinguishing it from its previous version in [10] are highlighted. It starts with an initial population composed of levels with randomly sampled stacks of blocks, pigs and birds. A new fitness function evaluates stability, playability and structural characteristics of the levels via game simulations where unplayable levels are penalized. A tournament selects levels for reproduction based on their fitness values. New levels are created by crossover and mutation operators that try to keep the level stability. All new levels compose the population of the next generation, except the worst one that is replaced by the bes level from the current generation (elitism). Tanager stops after a given number of generations or if the fitness of the best level does not improve after some generations.

A. Level Representation

The new level representation introduced here extends that from [10] by encoding two new aspects: amount of birds and duplicated blocks. The first one adds only one integer to the genotype and the second one adds a boolean for each block of the level, as showed in Figure 3. Duplicated blocks are elementary or composed blocks that are added in the stack besides another one exactly like it (for example, in Figure 3, the first three blocks of stack 2 are duplicated blocks). To simplify how gaps between stacks are represented, the distances between stacks are removed in this new genotype and gaps are now added using empty stacks (stacks containing only one element -1). See Figures 2 and 3 for a comparison between the old and the new representations.

In the new genotype, the first element encodes the amount of birds and all the others encode stacks. Each element of a stack encodes information about a block with an integer representing its index and a boolean defining if it is duplicated or not. In this particular example, the level has 4 birds and 8 stacks, where the stacks 3, 5 and 7 are empty (they have only the -1 element). All the non-empty stacks have a pig on the top (designed by 31 values). The second block of stack 1, the first three blocks of stack 2 and the first block of stack 6 are duplicated since they have value 1 on the boolean parameter.

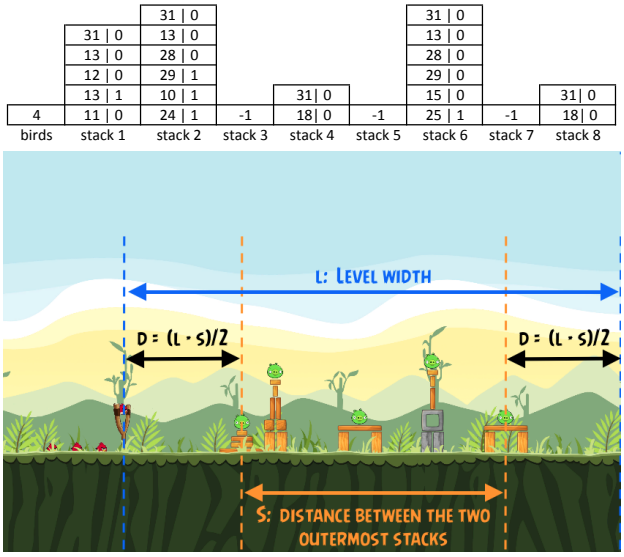
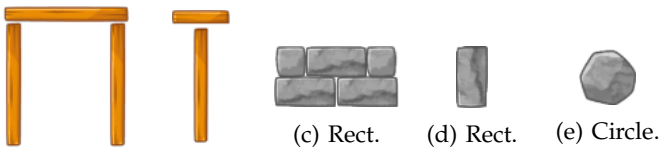


Figure 3: A genotype and its correspondent decoded level.

Decoding this new genotype into a level follows a process different than the one presented in [10] (See Section II for an explanation of the previous process). First, the amount of birds is initially added to the level, where one is placed on the slingshot and the remaining stay behind it. Next, the stacks are placed starting in a x_0 coordinate that is calculated based on both the level width L and the distance S between the two outermost stacks of the level. Both L and S are highlighted in Figure 3. We calculate $D = \frac{L-S}{2}$, which is the distance between the slingshot and the first stack of the level, before determining x_0. The value D is also the distance between the last stack and the end of the level. Thus, the coordinate $x_0 = D + x_s$ of the first stack is assigned by summing the distance D to the x_s coordinate of the slingshot. The coordinate $x_i = x_{i-1} + \frac{w_{i-1}}{2} + \frac{w_i}{2}$ of a stack $i > 0$ is calculated by adding both half the widths w_{i-1} and w_i of the wider block in stack $i-1$ and i, respectively, to the coordinate x_{i-1} of the stack $i-1$. The width w_i is equal to a constant $k > 0$ for all the empty stacks.

B. Blocks Classification System

Another novelty introduced is a classification system for blocks. They are now classified by their shapes, which helps to build more diverse and stable stacks. In this system, a block is necessarily from one of the following classes: R, T, C or B. The R blocks have rectangle shapes, T blocks have podium shapes (shapes similar to the letter "T"), C blocks have circular shapes and B blocks are boxes, i.e. rectangular blocks that can be stacked on top of other blocks. Figure 4 shows examples of blocks from different classes. This classification system based on shapes is introduced to help building stable stacks of blocks. There are blocks that will surely harm the stability of the stacks when placed over another. Thus, each class has a set of "safe relationships" with other classes, as showed by the graph illustrated in Figure 5.



(a) Box. (b) Podium. (c) Rect. (d) Rect. (e) Circle.

Figure 4: Instances of blocks from different classes, the first three are composed and the others are elementary.

A node in such graph represents a class and a directional edge from node a to b represents that a block from class a can be stacked on top of another block of class b. For example, the graph shows that only boxes can be stacked on top of C blocks and any one can be stacked on top of T, R and B blocks. Every time Tanager must place a block over another, it will first check if one has a safe relationship with the other. This relationships graph, by itself, does not guarantee the stability of the stacks hence Tanager still has to account for level stability during the evolutionary process. The graph helps on the search process avoiding the exploration of stacks that surely would not form stable structures.

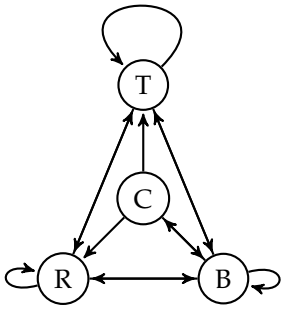


Figure 5: A graph representing safe relationships between different block classes.

C. Random Initialization

The first population of the GA is initialized with levels that have both a random amount (greater than zero) of stacks and birds. Each stack is formed by a random

amount of blocks, which may be zero for empty stacks. Algorithm 1 describes in details the process of generating a random level. Initially, it samples a random integer $1 \leq b \leq B$ using a uniform distribution (line 2), where B is the maximum amount of birds allowed. Next, it samples the width (in the game world space) of the stacks array using a normal distribution with both μ and σ equal to the half of the total level width W (line 4). Stacks are iteratively generated until the defined width has been fulfilled (line 5-21). To build a stack, the algorithm first samples its height (in the game world space) using a normal distribution with both μ and σ equal to the half of the total level height H (line 9). With the height defined, the algorithm places random blocks one by one, always considering the graph of relationships. This process is performed until the stack reaches its height or there are no blocks that can be safely added (line 10).

Algorithm 1: Sampling random levels.

```
1  A ← {};
2  b ← U(1, B);
3  w_fulfilled ← 0;
4  w_A ← N(W * 0.5, W * 0.5);
5  while w_fulfilled < w_A do
6      S ← {};
7      V ← all blocks;
8      h_fulfilled ← 0;
9      h_s ← N(H * 0.5, H * 0.5);
10     while h_fulfilled < h_i and V is not empty do
11         c_next ← PickRandomSafeClass(c_prev);
12         l_next ←
             PickRandomBlockFromClass(c_next, V);
13         if c_next is not a Box then
14             if c_next is not a Podium and U(0, 1) < 0.5
                 then
15                 DuplicateBlock(l_next);
16             if BoundingBoxArea_prev >
                 BoundingBoxArea_next then
17                 add l_next to S;
18                 h_fulfilled ← h_fulfilled + h_next;
19         else if blocks in the top of S fit inside l_next then
20             add l_next to S;
21             h_fulfilled ← h_fulfilled + h_next;
22     InsertPigs(S);
23     if h_fulfilled > 0 then
24         w_fulfilled ← w_fulfilled + w_s;
25     else
26         w_fulfilled ← w_fulfilled + k;
27     add S to A;
```

To add blocks in a stack, the initialization algorithm randomly selects a class c_{next} based on its relationships (see Figure 5) with the previously stacked block c_{prev} (line 11). The new block l_{next} is randomly chosen from the array V among all those from class c_{next} (line 12). Function **PickRandomBlockFromClass** removes the selected block from V, which ensures that the selection is done without replacement. If this is the first block of the stack, any class can be selected and, consequently, any block either. If the selected block is not a box (line 13), the next step is to check if it is also not a podium and "flip a coin" to define if the block will be duplicated or not (line 14). Podiums cannot be duplicated because they can easily collapse the stack.

After defining if the selected block will be duplicated, the algorithm compares the bounding box areas of the previous block and the next one (if it is not the first). This comparison is a simplified solution to the problem of discovering if two non-squared stacked blocks will fall when submitted to the gravity force [19]. Instead of analyzing the real shapes of the blocks, the algorithm assumes that they have a square shape. Therefore, if the bounding box area of the previous block is greater than the next one (line 16), l_{next} is added to the current stack S (line 17) and the height counter $h_{fulfilled}$ is incremented with the block height h_{next} (line 18). On the other hand, if the selected block is a box, the generator tries to cover the blocks in the top of the stack that fits inside it (line 19). Depending on the selected box and on the stack state, it will not be able to cover anyone. In this case, the selected box is not added. Otherwise, it is added (line 20) and the height counter is incremented (line 21).

The last step for creating a stack is to add pigs in available spaces, which in this case are either empty spaces inside boxes or the top of the stack. The function **InsertPigs** (line 22) adds a random amount of pigs to the stack by iterating among all its blocks looking for available spaces. If this function receives an empty stack as input, it simply "flips a coin" to decide if the pig will be placed on the ground or not. Otherwise, the function looks for an available space in the stack and, if such space is found, it "flips a coin" to decide if the pig will be placed there or not. If the procedure reaches the top of the stack without having added any pig, it automatically adds a pig in that place. In this case, if the top block has a circular shape, the algorithm substitutes it by a pig.

Once a stack is completely built, the algorithm increments the width counter $w_{fulfilled}$ with either the width of that stack w_s (line 24) or with a constant width k (line 26). The second option only happens if the created stack is empty. Finally, the algorithm adds the current stack S to the array A(line 27) and it tries to generate the next one. If all the stacks are empty at the end, the algorithm randomly selects one of them and substitute it by another one with height equal to $H * 0.5$, composed only by random rectangular blocks and a pig on the top.

D. Fitness Function

The initialization algorithm described in the last section does not ensure the levels are fully stable and

neither the amount of birds is enough to kill all pigs. The fitness function measures if a level satisfies these two constraints using a game simulation with an intelligent agent. Stability is measured by the total velocity of the blocks during the beginning of the simulation. When the simulation starts, the agent waits to throw the first bird until the blocks stay stable. This means to wait until the magnitude of their velocity vectors becomes zero. During this period, the algorithm measures the velocity of all blocks 10 times per second and accumulates these values in a variable s.

After measuring stability, the level feasibility is evaluated. A level is only considered feasible if the amount of pigs p_f at the end of the simulation is equal to zero. This solution is strongly dependent on the strategy used by the agent to kill pigs. In the applied strategy, the agent initially selects a random pig of the level and tries to shot a bird on it. If the agent doesn't succeed, a random pig is selected again. If the same pig is selected twice in a row, the agent adds a noise in the pig's position and throws the bird toward such position. The idea is to break blocks around that may be protecting the pig. On the other hand, if the selected pig p_i in the i_{st} shot is different from the previous pig p_{i-1}, the agent will shot directly at the direction of the pig p_i. When throwing a bird toward a pig, the agent does not consider if there are blocks in front of it. The bird will be always thrown with the highest possible velocity.

As mentioned before, besides evaluating stability and feasibility, the fitness function also evaluates some structural characteristics of the levels. More specifically, it measures the percentage of blocks in the level and the percentage of birds that is needed to kill all pigs. These metrics are used to control the algorithm output and they are defined, respectively, by parameters $0 \leq l_n \leq 1$ and $0 \leq b_n \leq 1$. The fitness function is mathematically described by Equation 1.

$$fitness(x) = |\lfloor b_n * B \rfloor - B_u| + |\lfloor l_n * L \rfloor - L_b| + p_f + s \quad (1)$$

In this function, B is a constant that defines the maximum amount of birds allowed in a level, B_u is the amount of birds used during the simulation. The constant L defines the maximum amount of blocks allowed in a level and L_b is the amount of blocks in the beginning of the simulation. The first term of the equation calculates the distance between the number of birds that should be used to kill all the pigs and the number that was actually used. The second term calculates the distance between the number of blocks desired in the level and the number of blocks that the level started with. If these terms are both zero, the level has all the desired characteristics. The remainder terms of the equation evaluate the number of pigs in the end of the simulation p_f and the stability s of the blocks in the level, respectively. The level is considered stable and feasible when these two variables are both zero. The design of Equation 1 shows that the objective of the proposed GA is to find its minimum value.

The physics engine used during the simulations does not work deterministically. The strategy used by the intelligent agent during the simulation is not deterministic as well. Thus, it is not guaranteed that the same level encoding (individual) will have the same fitness when evaluated many times. It is possible to reduce this problem by running the simulation several times for the same level and then considering the average of calculated values as the final fitness of that level [20]. This approach increases considerably the total evolution time, so another one was adopted in this paper.

If a level l is evaluated once and the intelligent agent was able to finish it, we know a human player will be able to finish that level as well. This is true because the agent uses exactly the same controls as the human player. Based on this argument, the proposed GA never recalculates the fitness value for a level. This is done by caching the fitness calculations with a hash table that stores the levels as keys and their fitness as values. Every time the GA must evaluate a level, it searches for the level inside the table. If it finds the level, the stored fitness is returned, otherwise, a simulation is executed and a new instance is added in the table.

This solution has two main advantages: considerable reduction on the amount of simulations and freedom to design the agent strategy, allowing random components. On the other hand, the hash table may consume a lot of extra memory, depending on the accuracy of the algorithm when exploring the search space. Moreover, levels that are considered unplayable and/or unstable, but actually are, will be penalized and never reevaluated again. Thus, their chance of surviving the evolution will be low and hence their features won't be propagated.

E. Genetic Operators

The fitness values are used to select n individuals for reproduction, where n is the size of the population. This process is performed iteratively and, in each of the n iterations, an individual is selected via a tournament composed of two different random parents. A new generation is then formed by applying a crossover operation in each pair of selected individuals, which in turn generates two new children. This operation is followed by a mutation that is performed separated on each child.

Both genetic operators are composed of two steps, the first one is applied on the birds and the second one on the stacks. Crossover recombines two individuals (parents) applying an arithmetic crossover on the amount of birds and an uniform crossover on the stacks. Considering two parents p_1 and p_2 with x_1 and x_2 birds, respectively. The arithmetic crossover produce two new amounts of birds y_1 and y_2, as shown in Equation 2. In this Equation, $0 \leq r \leq 1$ is a random weighting factor chosen before each crossover operation. Since the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCIAIG.2017.2766218, IEEE Transactions on Computational Intelligence and AI in Games

amount of birds is an integer, the crossover results are rounded using the floor of their values.

$$y_1 = \lfloor r * x_1 + (1 - r) * x_2) \rfloor$$
$$y_2 = \lfloor (1 - r) * x_1 + r * x_2) \rfloor \quad (2)$$

The other part of the crossover is intended for generating two new arrays of stacks from two parents p_1 and p_2 with m and n stacks, respectively. The uniform crossover generates the stack $i \in [1, max(m, n)]$ of a new individual (child) randomly selecting a stack either from p_1 or from p_2 with same chance. If the size of the two parents are different ($m \neq n$) and assuming $m > n$, each one of the $m - n$ last columns from p_1 will have 50% of chance to be included in the new individual. Otherwise ($n > m$), each one of the $n - m$ last columns from p_2 will have 50% of chance to be selected. Figure 6 shows an example of this crossover operation.

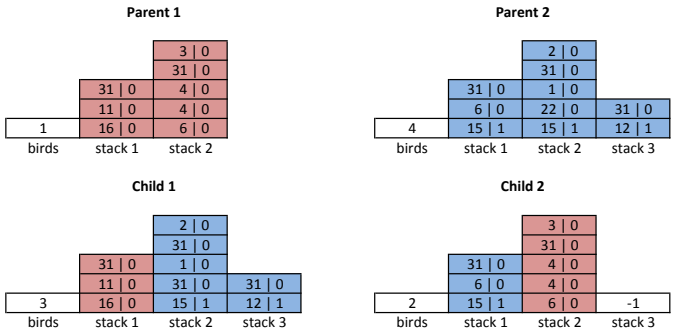


Figure 6: A crossover example. The two individuals on the top are the parents and the others on the bottom are the generated children.

In this example, the weighting factor of the arithmetic crossover is 0.2 and the birds amounts are 1 and 4, so the generated values are 3 for the first child and 2 for the second one. To generate the stacks of the child 1, the uniform crossover selected the first stack of the parent 1 and the last 2 stacks of the parent 2. To generate the stacks of the child 2, it selected the first stack of the parent 2 and the second stack of the parent 1. In this case, the crossover did not selected the third stack of the parent 2, so the third stack of the child 2 is empty.

There is a probability (mutation rate) to apply the mutation operator over the two new individuals generated after crossover. This mutation randomly changes the amount of birds and each stack of the individual with a certain likelihood. The operation on the amount of birds randomly generates a new integer value $1 \leq b \leq B$. On the other hand, the mutation on the stacks randomly generates a new stack following the same steps described on the initialization algorithm. Figure 7 shows an example of the mutation operation applied over the amount of birds and the stacks. The amount of birds of the first child of Figure 6 changes from 3 to 5. The mutation operation also changes stack 2 of child 1 and stack 3 of child 2.

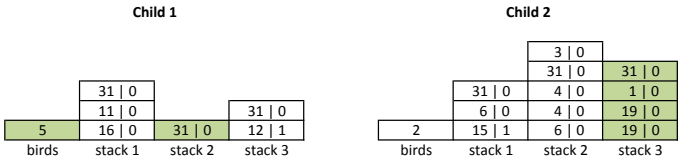


Figure 7: A mutation example on the two individuals generated in Figure 6.

IV. EXPERIMENTS

Tanager was evaluated by three sets of experiments. The first one is an evaluation of the GA to show its performance and how the fitness values improve over generations. In the second experiment, the expressivity of Tanager is analyzed by generating a high quantity of levels and analyzing their structural characteristics. The third experiment is an user study to measure the design quality of these levels considering difficulty, immersion and fun. In all the experiments, the GA is set with the following parameter values: population size of 100, tournament size of 2, crossover rate of 95% and mutation rate of 10%. The method runs for 25 generations at most or for 10 generations without improvement over the best individual. The maximum amount of birds (B) is set to 5 because the hand-authored levels of the original game do not usually have more than 5 birds. The maximum amount of blocks (L) is set to 100 since it makes a level practically full of blocks. Moreover, with these values, Tanager should be able to generate a wide variety of levels.

A. Performance of the Genetic Algorithm

Tanager's performance is measured based on the evolution of the best fitness through generations. Figures 8 and 9 show the evolution of the average of the best fitness within 30 independent executions. It is also depicted the average (μ) and standard deviation (σ) values for each component of the fitness function: stability s of blocks, amount of pigs p_f at the end of the simulation (Figure 8), and the distance to desired amount of birds and blocks (Figure 9). Thus, these results allow to evaluate the characteristics of the best levels through the generations. During the experiment, the percentage of blocks is set to $l_n = 0.25$ and the percentage of birds needed to kill all pigs is set to $b_n = 0.25$, which means the generated levels should have 25 blocks and 1 out of 4 birds should be required to complete them.

Analyzing the first generation in Figure 8, one can understand the performance of the initialization method on generating stable levels for the first population of the GA. The μ and σ values are approximately 0.25 and 0.4 for the stability component s, respectively. This means that 95% of the initial levels have $s \leq 1.05$ since, as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCIAIG.2017.2766218, IEEE Transactions on Computational Intelligence and AI in Games

described in Section III-D, the stability is the sum of the velocity vectors of all the blocks before the first bird shot, measured 10 times per second. In the Angry Birds clone, an object with velocity magnitude of 1 moves 100 pixels/s and the playable area of the level (to the right of the slingshot) has approximately 600 pixels. Figure 9 shows that the average amount of blocks in the first generation is approximately $25 - 4.6 = 20.4$. It is hard to know how the stability of 1.05 is divided among these 20.4 blocks. However, if the stability is divided equally among them, then a block moves on average $\frac{1.05 \times 100}{20.4} = 5.14$ pixels/s during the stability evaluation. This value represents $\frac{5.4}{600} = 0.009$ times the playable area of the level. Thus, 1.05 is a low value and hence the initialization method is able to generate a great amount (95%) of levels with high stability.

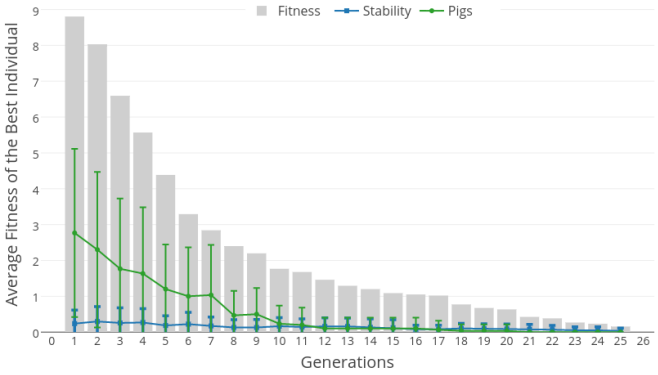


Figure 8: Fitness improvement over generations, highlighting the evolution of the two last components of the fitness function: stability s and pigs at the end of the simulation p_f.

The average stability was improved through all generations in Figure 8, reaching values very close to 0 from the 19th generation on. This indicates that the proposed genetic operators are not only capable of keeping the stability of the initial population of levels, but also improving it to be almost absolutely stable. The initialization algorithm does not reason about the levels feasibility, but from the $\mu = 2.7$ and $\sigma = 2.3$ values of the pigs fitness component (p_f) at the first generation in Figure 8, one can see that approximately 85% percent of the levels are infeasible. This means that at least one pig survived the simulation. However, along the first 10 generations, Tanager strongly improved p_f components by reducing it to almost 0. Moreover, this value was reduced to exactly 0 from the 21st generation until the last one. This shows that, besides generating stable levels, Tanager also guarantees their feasibility.

The evolution of the remaining two fitness components are shown in Figure 9. While the last two components s and p_f are related to stability and feasibility, these ones represent the desired characteristics of the levels in terms of birds and blocks. In the first generation, $\mu = 4.6$ and $\sigma = 4$ values for blocks shows that approximately

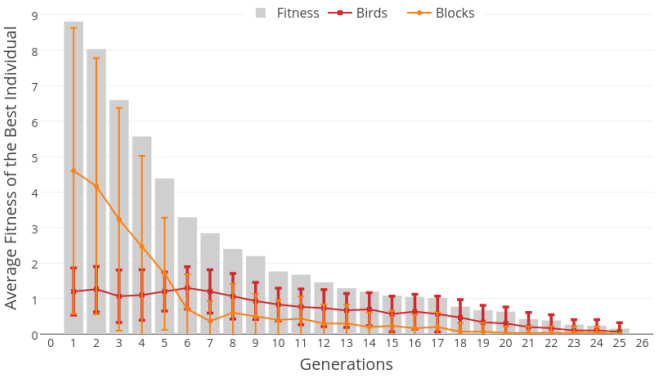


Figure 9: Fitness improvement over generations, highlighting the evolution of the two first components of the fitness function: distance to desired amount of birds and blocks, respectively.

85% of the levels do not have the desired amount of blocks. Among all the four fitness components, this is the one that starts furthest from its optimal value. However, Tanager quickly optimized such value to less then 1 in the first 7 generations. It used the remaining 18 generations to optimize this component to its optimal 0 value. Regarding the birds component, the $\mu = 1.2$ and $\sigma = 0.8$ point out that 85% of the initial levels also do not have the desired amount of birds. Different from other components, this one gets slightly worst in the first 8 generations. However after that it constantly get optimized, reaching its optimal value at the end of the evolutionary process. These results show that in average Tanager successfully finds the optimal value of the fitness function. Thus, it is capable of generating stable and feasible levels with a desired amount of blocks and birds.

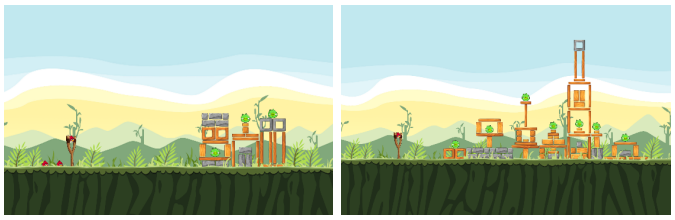
B. Expressivity Analysis

Expressivity analysis was proposed by [21] as a way to evaluate the variety of levels that a generator is capable of creating as well as the impact of changing input parameters. The majority of Angry Birds generators proposed so far was evaluated according to three metrics: *frequency*, *linearity* and *density* [10], [11], [12]. The levels generated by Tanager are also evaluated according to these metrics. Frequency represents the average amount of relevant elements in the level. This metric is calculated by counting the amount of a certain game object and dividing it by the total amount of objects in the level. In this particular case, we are interested in measuring the frequency of birds and pigs. The amount of blocks is not measured because it is a parameter of the algorithm.

Linearity metric is calculated by finding the variance of the stack heights in the level (including the empty ones). Results are normalized to [0,1], where 1 is highly linear and 0 is highly non-linear. Therefore, stacks with different heights present low linearity, while stacks with similar heights have high linearity. Thus, the linearity

of a level is measured using Equation 3, where h_i is the height of the stack i, $\bar{h}$ is the average height of the stacks and H is the total level height. Figure 10 illustrates two levels with different linearities.

$$linearity = 1 - \frac{\sqrt{\sum_{i=1}(h_i - \bar{h})^2}}{\sqrt{\sum_{i=1}(h_i - H)^2}} \quad (3)$$

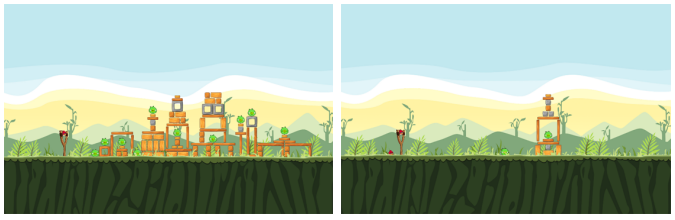


(a) A level with high linearity (0.98). (b) A level with low linearity (0.14).

Figure 10: Example of levels with high difference of linearity. Level (a) has high linearity because all of its stacks have a similar height. In the other hand, level (b) has low linearity because its stacks heights are very different.

Density measures the stacks arrangement among the playable area of the level. This metric is calculated by adding the width of all **non-empty** stacks in the level and dividing it by the total level width. Density values are normalized to [0,1], where 1 is the highest and 0 is the lowest density measure. This means that levels with many stacks and wider blocks have a high density, while levels with a few stacks and narrower blocks have low density. Equation 4 gives the density of a generated level, where w_i is the width of the non-empty stack i and W is the total level width. Figure 11 illustrates two levels with different densities.

$$density = \frac{\sum_{i=1} w_i}{W} \quad (4)$$



(a) A level with high density (0.98). (b) A level with low density (0.14).

Figure 11: Example of levels with high difference of density. Level (a) has high density because it occupies almost the whole playable area. In the other hand, level (b) has low density because its stacks occupies a small area.

A total of 500 levels is used to evaluate the expressivity considering different combinations of parameters b_n and l_n. First, it is set $b_n = 0.5$ and 100 levels are generated for l_n= 0.25, 0.5 and 1.0. Next, it is set $l_n = 0.5$ and another 100 levels are generated for b_n = 0.25 and 1.0. To visually describe how the GA outputs look like, Figure 12 shows some levels that came out during this process.

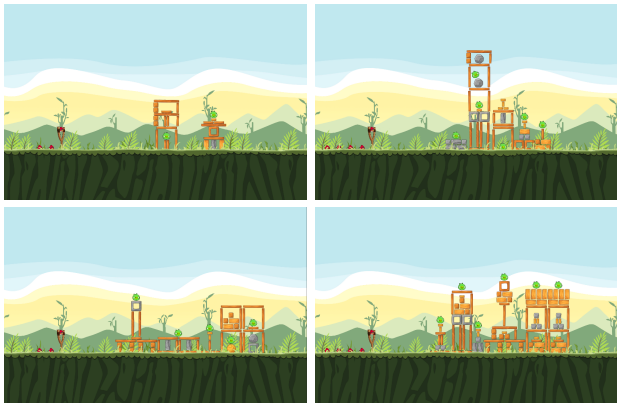


Figure 12: Example of levels generated with the GA using different values of l_n and b_n.

The first analysis evaluates the frequency of pigs and birds in the generated levels. Figure 13 shows charts with the average and standard deviation of these values for each variation of b_n and l_n. In all results, the frequency of pigs stays between 10% and 20% of the total amount of blocks in the level. Changing the characteristics of the levels in terms of b_n and l_n does not dramatically affect the amount of pigs in the levels. This can be explained by how pigs are inserted in the levels (see Section III-C). If there are more boxes with empty spaces, there is more chance to insert pigs into them. This process is not related with the amount of birds in the level, but it is affected by the type of blocks (boxes with empty spaces). Thus, this result also shows that the frequency of boxes is not considerably affected by parameters b_n and l_n.

Frequency of birds is ranging in average from 0.6 to 0.9 in Figure 13. These values are normalized by the maximum amount of birds B, which is defined as 5 for all the experiments. Thus, levels are likely to have between 3 and 5 birds. The standard deviation values show that, in all the variations, approximately 95% of the levels have more than 1 bird. Therefore, Tanager rarely generates a level with only one bird. This can be explained by the components of the fitness function that affect the amount of birds. Birds are inserted in a level in order to meet the requirements defined by the user $\lfloor b_n * B \rfloor$ and to make the level feasible p_f. In the experiments, the lowest bird's requirements considered is $\lfloor 0.25 * 5 \rfloor = 1$. In this case, to minimize the fitness function, B_u has to be equal 1 and p_f equal 0. The results in in Figure 13 showed that the amount of pigs in average is at least 10% of the amount of blocks. The lowest amount of blocks considered in the experiments

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCIAIG.2017.2766218, IEEE Transactions on Computational Intelligence and AI in Games

is $l_n = 0.25$. Thus, the simplest generated levels have in average $|0.10 * 0.25 * L| = 2$ pigs. Using exactly one bird ($B_u = 1$) to kill 2 pigs ($p_f = 0$) is not easy, specially in levels with a low amount of blocks. For values of $b_n > 0.25$, Tanager will generate levels with more than one bird in order meet the constraints added by the user. Thus, it is very uncommon for Tanager to generate levels with only one bird.

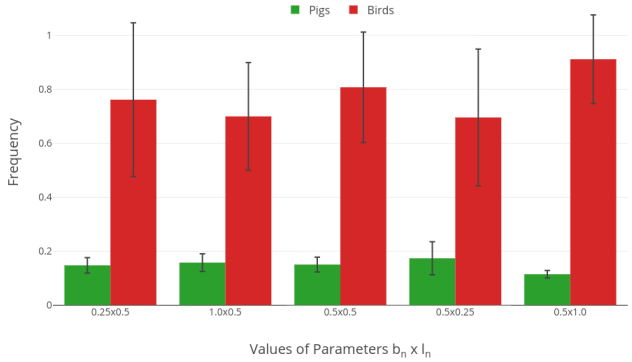


Figure 13: Impact in frequency when variating the parameters b_n and l_n.

The average frequency of birds increases when $b_n = 0.5$ and l_n is raised, as shown by the three rightmost columns. The non-parametric Kruskal-Wallis test was applied to compare these results, since the hypothesis of normal distribution is not fulfilled by samples for a parametric test. Kruskal-Wallis test returned a p-value < 0.05 showing a statistical significant difference among the results.This leads to the expected conclusion that levels with more blocks need more birds to be finished.

When parameter l_n is fixed and b_n changes (three leftmost columns), the amount of birds does not increase monotonically as a function of b_n. The three leftmost columns indicate that Tanager generated levels with the same amount of blocks ($l_n = 0.5$) requiring a different amount of birds (from 3 to 5). It is important to highlight that the total amount of birds will affect the level difficulty, once it can be harder to play a level with a given amount of blocks using fewer birds.

Levels with a large amount of blocks are likely to have higher stacks, which can increase the fragility of the whole structure. If the height difference between these stacks is very high, the chance of a thrown bird to destroy blocks and pigs is high. It increases even more if there are empty spaces between stacks. However, results show that levels with more blocks need more birds to kill all pigs. This indicates that levels' stacks have similar heights and there are few empty spaces between them. It is hard to conclude about structural characteristics of levels based only on frequency results. These characteristics can be deeply discussed from linearity and density results.

Figure 14 illustrates how parameter variation impacts on the linearity and density of the levels. Changing the parameter b_n does not impact considerably the linearity and density of the levels, once levels with $l_n = 0.5$ are not complex enough due to constraint on the amount of blocks. Thus, the proposed agent can solve them most of the time, independent of the b_n value. In this case, Tanager does not have to (or can't, due to the constraint on the amount of blocks) search for very distinct structures, leading the levels with similar values for linearity and density.

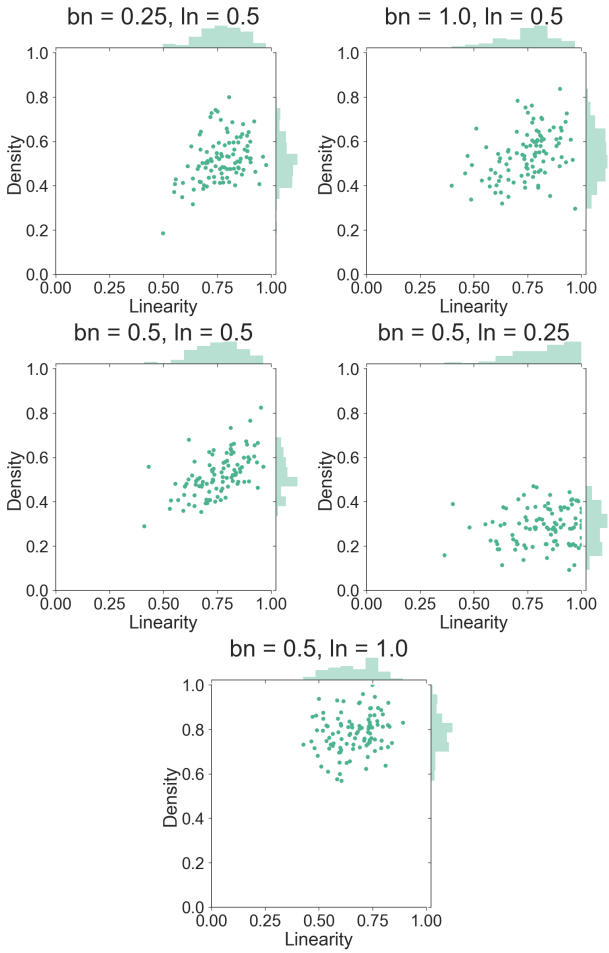


Figure 14: Impact in expressivity when variating the parameters b_n and l_n.

Parameter l_n, on the other hand, has a relevant impact over linearity and density. Most of the levels generated with 25 blocks ($l_n = 0.25$) have high linearity (between 0.8 and 1) and low density (between 0.2 and 0.4). The majority of levels with 50 blocks ($l_n = 0.5$) has high linearity (between 0.7 and 9) and a medium density (between 0.4 and 0.6), while levels with 100 blocks ($l_n = 1$) have intermediary linearity and high density. Linearity values confirm that the height variation among levels' stacks is low. Density values show that Tanager has a bias for distributing blocks among few stacks, instead of creating more stacks with smaller heights.

Overall, increasing the amount of blocks directly increases the level density, but slightly decreases their linearity. This is an expected result once levels with many

blocks require more space to be placed. Moreover, there are spaces in the expressivity range that the generator does not cover, mainly levels with low linearity (less than 0.5). Levels with linearity less than 0.5 have both tall and short stacks, which requires a high amount of blocks. This level configuration is feasible but is likely to be unstable (mainly because of the tall stacks). In order to increase the stability of the levels, Tanager tends to generate stacks with similar heights, so they can support themselves side-by-side.

The expressivity results showed that Tanager can generate a considerable variety of levels, with linearity values ranging from 0.5 to 1 and density from 0 to 1. Considering this variety and the fact that all these levels are stable and feasible, we can say that caching the fitness does not lead to big level generation problems.

C. Immersion Analysis

This section presents an user study which is intended to evaluate immersion aspects of the generated levels. According to [22], immersion can be described as the degree of involvement of a player with a computer game. Therefore, an immersible game has the ability of drawing people in, providing an appealing distraction from everyday worries and concerns. Such degree of involvement can be measured by a combination of human, computer and contextual factors like game preference, game construction, environmental distractions, etc. Considering these factors, [22] proposed a questionnaire composed of 31 questions for evaluating immersion in games. A small version of this questionnaire with 6 questions (see Immersion questionnaire in Appendix A) is used in this experiment to evaluate players' immersion while playing levels generated by Tanager. To reach a better analysis, the same evaluation is also performed over levels adapted from the original Angry Birds. Thus, this experiment allows a comparison of immersion aspects between levels generated by Tanager and levels from the original game.

The user study is conducted using the Angry Birds clone presented in [10]. The questionnaire is inserted in such clone, which is compiled into a web build that can be accessed online [1] from any web browser. This version of the game contains two different sets of levels P and R presented for players in random order. Each set has 5 levels, where those in P are generated by Tanager and those in R are collected and adapted from the original Angry Bird game. Set P is defined by generating 3 levels for each one of the 5 parameter configurations showed in Figure 13. Generating a level means picking the level with the best fitness at the end of the evolution process. For each configuration, it is randomly selected only one of these 3, totalizing 5 levels with different characteristics. Set R is defined by selecting 5 random levels from all of those of the first episode from the original game (called Poached Eggs), which are adapted to the clone version.

One limitation of this clone is that it supports only red birds, so all non-red birds are replaced by red ones. The amount of birds is increased sometimes, based on how the replacement of non-red birds impacts feasibility. The clone also does not support glass blocks which are replaced by either wood or stone blocks. Adapting the levels does not dramatically change the strategy that players have to use to complete the levels. This is because the majority of the levels in the first episode of the game does not strictly require the use of non-red birds. Therefore, this adaptation should not impact considerably how the players report similarities between the two sets P and R. With these two sets, the structure of the immersion experiment is as follow:

1) **Player opens the game url in its browser and a screen containing the description of the experiment is presented.**
   This screen informs the player that he/she will play two different sets of levels and will evaluate each one just after playing it. At this moment, the player does not know the differences between the sets.
2) **A profile questionnaire is presented to the player.**
3) **First set of levels is presented to the player.**
   The set is randomly selected between P or R.
4) **Player evaluates, using the immersion questionnaire in Appendix A, the set played.**
5) **Second set of levels is presented to the player.**
   If P was already played, then R is presented. Otherwise, P is presented.
6) **Player repeats task 4 for the second set.**
7) **A new screen appears questioning the player about which of these two was generated by computer.**
   This screen informs, before the question, that one of the sets were manually produced and the other one was generate by computer.
8) **Another screen appears giving the correct answer of this question.**
9) **Player evaluates, using the last questionnaire in Appendix A, the similarity between the two sets and its perception about the procedurally generated levels.**

The last 3 steps of the experiment are not intended to measure immersion, but the similarity between the two groups of levels. Step 7 is a Turing test in order to measure if the player is capable of realizing which of the groups was generated by a computer. In step 8, the correct answer of the Turing test is revealed and, in step 9, the player must answer a questionnaire in order to describe how similar the two groups were. Giving the correct answer to the player is important to ensure he/she will be able to correctly analyze how similar the procedurally generated levels are in comparison to the ones designed manually.

[1] http://www.lucasnferreira.com/AngryBirdsWeb/AngryBirdsWeb.html

This experiment was conducted with 139 players over the Internet and the distribution of age, gender and education of the participants is showed in Table II. The participants were recruited via social media and no rewards were provided to them. The majority of participants are male between 21 and 30 years old with complete higher education.

Age	Players	Gender	Players	Education	Players
less-9	1	Female	19	Primary	1
10-20	12	Male	120	Secondary	11
21-30	101	Other	0	Bachelor	67
31-40	22			Master	48
41-50	3			Doctor	12
41-60	0				
61-more	0				

Table II: Distribution of age, gender and education of the players in the immersion user study.

The experience of the players on general games and their experience on Angry Birds are measured with a five-level Likert scale, where 1 means "not at all" and 5 means "a lot". As shown in Table III, the majority of players have great experience with games (approximately 82% answered 3 or higher) and little experience in playing Angry Birds (approximately 76% answered 3 or lower).

	General Games	Angry Birds
1	7.91 %	20.14 %
2	10.79 %	26.62 %
3	23.74 %	29.50 %
4	23.74 %	17.99 %
5	33.81 %	5.76 %

Table III: Distribution of experience in general games and Angry Birds of the players in the immersion user study.

The score of each question of the immersion questionnaire is also measured using a five-level Likert scale, where 1 means strongly disagree and 5 means strongly agree. The distribution of scores per question for both sets P (Tanager) and P (Rovio) is presented in Table IV. The first two questions are related to each other and they evaluate how engaged players are while playing the levels. The consistency of the answers between these two similar questions suggests that the engagement experienced by the players was similar in the two groups.

		Q1 Attention	Q2 Time	Q3 Challenge	Q4 Give up?	Q5 Enjoyed?	Q6 Replay?
T	1	8	19	**77**	24	4	20
	2	13	35	25	32	16	28
	3	42	**43**	16	**35**	42	31
	4	**56**	34	10	**35**	**52**	**38**
	5	19	7	10	12	24	22
R	1	6	23	13	**67**	7	23
	2	15	35	28	30	20	35
	3	**53**	**43**	**49**	22	47	**39**
	4	49	25	38	14	**54**	27
	5	15	12	10	6	10	15

Table IV: Distribution of scores per question about immersion for both Tanager (T) and Rovio (R) levels.

Questions 3 and 4 are also related and they measure the challenge that players faced while playing the levels. The answers of question 3 are very different between the groups. More than 100 players reported that the Tanager levels have low challenge (scores equal or less then 3), while approximately 100 players found that the Rovio levels have high challenge (scores equal or greater then 3). This result shows that the Rovio levels are considerably harder than the Tanager ones. One of the main reasons for that is the level representation based on stacks used in Tanager. This bias on the level structures makes it easier for players to complete the levels because their solutions are more predictable. The graph of safe relationships does not lead to this difficulty limitation once it only helps Tanager to not evolve levels that are surely not stable and hence not feasible.

The answers of question 4 show that, even though the Rovio levels were challenging, they were not excessively hard. This is an expected result because levels in commercial games are usually tested and polished through several rounds of playtesting, in order to avoid the players to get extremely frustrated. In the Tanager group, a similar amount of players answered this question with the intermediary scores 2, 3 and 4. This means that a considerable amount of players found the Tanager levels excessively hard and another similar amount did not. Different from the hand-authored levels, the generated ones do not consider feedback from players in their design process. Tanager's fitness function does not model difficulty explicitly. Thus, it is expected that the levels will have varied difficulty because we are analyzing levels with high difference on the amount of blocks and birds. Considering the answers of questions 3 and 4 together, we notice that most of the Tanager levels are not challenging, but the ones which are challenging, are also more frustrating than the Rovio ones.

The last two questions are also connected and they measure the enjoyment/fun experienced by the players. Similar to the questions 1 and 2, the answers of the players of these questions are very similar in both groups. The consistency of the answers shows that players enjoyed playing both groups. After answering the immersion questions for both groups, players were warned that one of them is generated by a computer and they have to discover which one it is. A total of 68 players failed the test and 71 passed it. These values are very close, which again suggests a similarity between the two groups of levels. In order to analyze how the experience of the players impact on their ability to distinguish the groups, Figure 15 shows the distribution of players who failed the test per experience class.

The way that general game experience impact on their capabilities of distinguishing the level groups is unexpected. The more experienced players are, the less accurate they are on performing this task. This suggests that experience on general games does not help players to recognize angry birds procedural levels. The distribution of players according to Angry Birds experience, on the other hand, has a normal shape. The majority of players who failed the test have an average experience. These

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCIAIG.2017.2766218, IEEE Transactions on Computational Intelligence and AI in Games

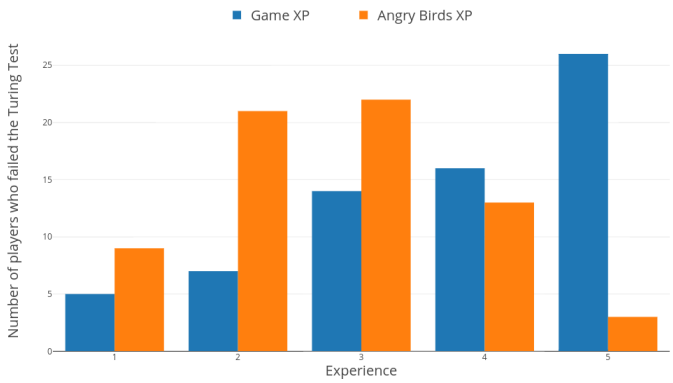


Figure 15: Distribution of players who failed the Turing test considering their experiences on general games and Angry Birds.

players probably found it hard to pass the test because they are experienced enough to know the mechanics of the game, but not to recognize patterns in the structure and/or puzzles of the levels. The very experienced players probably recognize these patters and that is why they can better distinguish the two groups of levels. It is hard to know the reasons why only a few non-experienced players failed the test. Because they don't know much about the game and its original levels, probably they guessed their answers almost randomly.

In the last step, players have to answer two questions about similarity between the two groups of levels. At this point, they already know which group is generated by a computer. The similarity scores use the same scale of the immersion questionnaire. However, they collect two different metrics: similarity between the groups and perception of the players that levels from Group P are generated by a computer. The distribution of scores per question is shown in Table V.

	Q1			Q2
	Total	Failed	Passed	Total
1	13	11	2	22
2	29	**26**	4	**40**
3	35	22	13	38
4	**44**	7	**37**	23
5	19	2	15	7

Table V: Distribution of scores per question about similarity between Tanager and Rovio levels. Scores for question 1 are reported in three formats: considering the total amount of players, only the ones who failed the Turing Test and only the ones who passed it.

The answers for the question 1 show that the majority of players who failed the test indeed did not thought that the Tanager levels were generated by a computer. They also point out that the majority of players who passed the tests indeed could recognize that Tanager levels were procedurally generated. This suggests that the generated levels have some specific differences when compared to the original ones. Results of the previous questions showed that both groups are similar in terms of engagement and enjoyment, but Rovio levels are more challenging than the Tanager ones. Therefore, these results suggest that players distinguished the differences in terms of challenge or they found the structures of the levels to be different. The answers of question 2 show that, when asked explicitly about similarity, the majority of players (more than 100) found the levels not very similar (answers equal or less than 3). Thus, the two level groups present visible differences in the arrangement of their objects. This can be explained by the level representation used by Tanager (see Figure 3), which adds a bias (stacks placed side-by-side) to the structure of the generated levels. Rovio levels do not present such biased structures.

V. Conclusion and Future Work

This paper presented a genetic algorithm (GA) for procedural generation of levels in physics-based puzzle games called Tanager. Levels were evaluated by a fitness function that carries about the stability of objects in the level and its feasibility. Stability is measured by the total velocity of the objects during the beginning of the simulation and feasibility is measured by a bot that randomly selects pigs to shoot. Three sets of experiments were performed in order to evaluate the proposed algorithm. The first one evaluates the Tanager performance in terms of evolution of the best fitness through generations. Results showed that, in average, the GA is capable of finding optimal value of the fitness function. This means that Tanager generates stable and feasible levels with a desired amount of blocks and birds.

The second experiment measures the expressivity of Tanager as a level generator. Three metrics of the literature were used to represent the features of the levels: frequency, linearity and density. This experiment generated 500 levels among 5 different configurations (100 levels each) and analyzed their distribution based on those metrics. It was possible to conclude that the frequency of pigs is not directly affected by variations in the input of the algorithm. Regarding the frequency of birds, levels are likely to have between 2 and 5 birds. Such results confirmed the hypothesis that levels with more blocks need more birds to be finished. Values achieved for linearity and density indicated that Tanager is biased to distributes blocks among few stacks, instead of creating more stacks with smaller heights. Another finding was that increasing the amount of blocks directly increases levels density, but slightly decreases their linearity.

The third set of experiments was intended to evaluate immersion aspects of the generated levels, in comparison with manually created levels adapted from the first episode of the original game. Two groups of levels were played by 139 participants and they answered questionnaires about immersion and similarity of the groups. Results suggested that the engagement and enjoyment of the groups were similar, while the original levels were more challenging. The participants also found the objects

arrangement of the two groups to be different. Moreover, previous experience of players in general games did not affect directly their answers on such evaluation.

All the results showed the Tanager is an approach capable of generating relevant levels for the first episode of the Angry Birds game. The structure and style of these levels are quite simple compared to more advanced episodes. Therefore, one of the future works of this project is to improve the level representation aiming to support more complex levels. The current intelligent agent used to evaluate levels applies a simple strategy, but such strategy has a relevant impact over feasibility evaluation. Thus, defining a better strategy is another future work that may improve the quality of the levels generated. Finally, both stability and feasibility are being evaluated by a single polynomial fitness function. Separating these two aspects in different objectives is an improvement in the genetic algorithm itself, making it a multi-objective approach [23], which may help to explore better the search space.

APPENDIX A
QUESTIONNAIRES USED TO EVALUATE IMMERSION AND SIMILARITY

Immersion Questionnaire

1) **To what extent did these levels hold your attention?**
   Not at All 1 2 3 4 5 A lot
2) **To what extent did you lose track of time?**
   Not at All 1 2 3 4 5 A lot
3) **To what extent did you find these levels challenging?**
   Not at All 1 2 3 4 5 A lot
4) **Were there any times during these levels in which you just wanted to give up?**
   Not at All 1 2 3 4 5 A lot
5) **How much would you say you enjoyed playing these levels?**
   Not at All 1 2 3 4 5 A lot
6) **Would you like to play these levels again?**
   Definitely not 1 2 3 4 5 Definitely yes

Similarity Questionnaire

1) **To what extent do you think the levels from the Group B were generated by the computer?**
   Not at All 1 2 3 4 5 A lot
2) **To what extent the levels from Group B were similar to those from Group A?**
   Not at All 1 2 3 4 5 A lot

REFERENCES

[1] N. Shaker, J. Togelius, and M. Nelson, "Procedural content generation in games," 2014.

[2] J. R. Mariño and L. H. Lelis, "A computational model based on symmetry for generating visually pleasing maps of platform games," in *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[3] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient sketchbook: Computer-aided game level authoring." in *FDG*, 2013, pp. 213–220.

[4] P. L. Lanzi, D. Loiacono, and R. Stucchi, "Evolving maps for match balancing in first person shooters," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[5] L. Cardamone, P. L. Lanzi, and D. Loiacono, "Trackgen: An interactive track generator for torcs and speed-dreams," *Applied Soft Computing*, vol. 28, pp. 550–558, 2015.

[6] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 78–89, 2014.

[7] Q. Xu, J. Tremblay, and C. Verbrugge, "Generative methods for guard and camera placement in stealth games." in *AIIDE*, 2014.

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

[9] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE, 2010, pp. 1–8.

[10] L. Ferreira and C. Toledo, "A search-based approach for generating angry birds levels," in *Proceedings of the 9th IEEE International Conference on Computational Intelligence in Games*, ser. CIG'14, 2014.

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

[12] M. Stephenson and J. Renz, "Procedural generation of levels for angry birds style physics games," in *The AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2016.

[13] R. Entertainment. (2010) Angry birds. [Online]. Available: http://www.angrybirds.com

[14] L. Ferreira and C. Toledo, "Generating levels for physics-based puzzle games with estimation of distribution algorithms," in *Proceedings of the 11th International Conference on Advances in Computer Entertainment*, ser. ACE'14, 2014. [Online]. Available: http://www.lucasnferreira.com/papers/2014/ace-edaab.pdf

[15] L. Pereira, L. Ferreira, L. Lelis, and C. Toledo, "Learning to speed up evolutionary content generation in physics-based puzzle games," in *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, ser. ICTAI'16, 2016. [Online]. Available: http://www.lucasnferreira.com/papers/2016/ictai-learning.pdf

[16] M. Shaker, M. H. Sarhan, O. Al 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*. IEEE, 2013, pp. 1–8.

[17] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach." in *AIIDE*, 2013.

[18] P. Zhang, J. Renz *et al.*, "Qualitative spatial representation and reasoning in angry birds: The extended rectangle algebra." in *KR*, 2014.

[19] J. Wang, P. Rogers, L. Parker, D. Brooks, and M. Stilman, "Robot jenga: Autonomous and strategic block extraction," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, Oct 2009, pp. 5248–5253.

[20] A. Di Pietro, L. While, and L. Barone, "Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2, June 2004, pp. 1254–1261 Vol.2.

[21] 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.

[22] C. Jennett, A. L. Cox, P. Cairns, S. Dhoparee, A. Epps, T. Tijs, and A. Walton, "Measuring and defining the experience of immersion in games," *Int. J. Hum.-Comput. Stud.*, vol. 66, no. 9, pp. 641–661, sep 2008.

[23] K. Deb, *Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction*. London: Springer London, 2011, pp. 3–34. [Online]. Available: http://dx.doi.org/10.1007/978-0-85729-652-8_1