Automatic Generation and Analysis of Physics-Based Puzzle Games

Mohammad Shaker, Mhd Hasan Sarhan, Ola Al Naameh, Noor Shaker and Julian Togelius, *Member, IEEE*

***Abstract*—In this paper we present a method for the automatic generation of content for the physics-based puzzle game *Cut The Rope*. An evolutionary game generator is implemented which evolves the design of levels based on a context-free grammar. We present various measures for analyzing the expressivity of the generator and visualizing the space of content covered. We further perform an experiment on evolving playable content of the game and present and analyze the results obtained.**

I. Introduction

The automatic generation of game content is receiving increasing attention due to the advantages it provides in terms of speeding up the content generation process, enabling on-line generation, reducing the development budget and enabling the creation of infinite content variations. Furthermore, techniques that explore a wide space of content might possibly be able to create more novel content than humans. Different techniques have been explored to automatically generate different aspects of content and some of them achieved remarkable results in commercial games [1], [2], [3]. The automatic generation of various aspects of game content has been explored relatively extensively recently with many studies reported in the literature on evolving tracks for car racing games [4], the distributed evolution of weapons in a space shooter game [5] and multiobjective evolution of maps for strategy games [6]. The generation of complete playable games has also received some attention [7], [8], [9].

However, to the best of the authors' knowledge, generating content for physics-based puzzle games is an area that has not been explored yet. This genre of games has recently become very popular, especially on mobile devices – good examples are *Angry Birds*, *Bad Piggies*, *Tower of Goo* and *Crayon Physics*. Physics-based puzzle games provide an interesting testbed both for content generation and for investigating the applicability of various AI methods. The physics constraints applied and generated by the different components of the game necessitate considering factors when evaluating the content generated other than the ones usually considered for other game genres – it is far from obvious what makes a good level for such a game. Testing for playability is another issue that differentiates this type of games since this needs to be done based on a physics simulator.

In this paper we present an approach for automatic generation of content for a clone of *Cut the Rope*, a popular commercial physics-based puzzle game. We analyze the game and present the evolutionary approach followed to generate game content. The design of the levels is specified in a context-free grammar employed by Grammatical Evolution (GE). We investigate two fitness functions: in the first one we focus on the physics aspect of the components and their properties while in the second a playability test is incorporated to guide the evolution process towards evolving playable content. The content space explored according to each fitness function is analyzed through a number of dissimilar expressive measures defined that allow thorough investigation of the generator's capabilities and permit illustrative visualization of the content space explored.

MS, MHS and OAN are with the Faculty of Information Technology Engineering at the University of Damascus, Syria (mohammadshakergtr,mhdhasansarhan,ola.un91@gmail.com). JT and NS are with the Center for Computer Games and Interaction Design at the IT University of Copenhagen (nosh,juto@itu.dk).

II. Grammatical Evolution

One of the techniques used to automatically generate content is evolutionary computation (EC). Evolutionary design is one of the areas where EC has demonstrated promising results that are competitive to those created by human experts [10], [11].

Grammatical Evolution (GE) is the result of combining an evolutionary algorithm with a grammatical representation [12]. GE has been used extensively for automatic design [13], [14], [15], a domain where it has been shown to have a number of strengths over more traditional optimization methods; it maintains a simple way of describing the structure of the levels and it enables the design of aesthetically pleasing levels by exploring a wide space of possibilities. The use of GE for the automatic generation of game content was first explored by Shaker et. al. [16], [17] for creating content for Super Mario Bros. A closely related method is recently used for generating playable card games [18].

In this paper we use a similar approach to the one proposed in [16]. A design grammar is defined to describe the possible structures of levels. The grammar is then employed by GE to evolve content according to a predefined fitness function.

III. CRUST 2D Physics Engine 2.0

The physics engine used for the design and generation of our game is the CRUST 2D Physics Engine 2.0 implemented in C# with XNA for managing runtime environment. The engine is our heavily modified version of Millington's engine [19]. Our modifications include adapting the engine to work with 2D environment and implementing the spring constraint. In its current state, the engine is able to provide efficient handling for physics simulations. The engine implements impulse force collision modeling to deal with rigid objects. Other physics-based motions such as springs, ropes, and hard constraints can also be simulated in the current version. The engine is also facilitated with a friendly user interface that allows editing objects and their physical properties at run time.

(a) A screenshot from the original game

(b) A screenshot from our clone of the game

Fig. 1. Two snapshots from the original Cut The Rope game (a) and our clone version of it (b) showing Om Nom waiting for the candy which is attached to ropes.

IV. The Game

The game which we generate content for is a clone *Cut The Rope* (CTR), a popular commercial physics-based puzzle video game released in 2010 by ZeptoLab for iOS and Android devices. The game was a huge success when released and it has been downloaded more than 100 million times. There is no open source code available for the game so we had to implement our own clone using the CRUST engine and the original game art assets. Our clone of the game is called *Cut The Rope: Play Forever*. The clone does not implement all features of the original game, but focuses on those that are more fundamental and relevant for content generation. Fig. 1.(a) shows one of the level in the original game while Fig. 1.(b) presents a level from our clone.

The gameplay in CTR revolves around feeding a candy to a little green frog-like monster named *Om Nom*. The candy is usually attached to one or more ropes which have to be cut with a swipe of the finger to set it free. Auxiliary objectives include collecting as many of the stars present in the level as possible. All game objects obey Newtonian physics adjusted to digital world and are affected by gravity [19]. The player loses the game by letting the candy escape (e.g. fall) outside the level boundaries. The game features a puzzle component by the presence of obstacles and other physics-based components that help redirecting the candy. The set of components included in the original game includes air-cushions, constrained pins, bubbles, shooting-buttons, rockets, spikes, spiders, suction cups among others (see Figure 2). The player interacts with the game by cutting a rope, tapping an air-cushion, a bubble or a button triggering a sequence of physics-based consequences. Solving the level puzzle depends to a great extent on timing. Specific actions should be taken at certain game states; otherwise the player loses the game.

(a) Om Nom (b) Candy (c) Rocket (d) Air-cushion (e) Constrained-pin (f) Bumper

Fig. 2. The various components presented in the original *Cut the Rope* game.

V. Game Design

There exist many different components in the original game. In this paper, we focus on seven of them that appear in most of the levels of the game; the description of each component also includes what parameters specify that component in our simulation.

- Ropes: ropes are an essential part of the game. They hold the candy and they can be cut to set the candy free. A rope is defined by its starting location in the 2D level map and it's length. When there is more than one rope in the level, all generated ropes should be connected to the candy when the game starts.
- Air-cushion: while attached to a rope or in a bubble, the trajectory of the candy can be changed by blowing air in its direction with an air-cushion. Air-cushions are defined by their position and orientation (east, west).
- Bubbles: The candy normally falls down due to gravity. If trapped in a bubble, the candy instead floats upwards. A bubble can be popped to free the candy inside. A bubble has an initial static placement in the level and it starts floating only when carrying the candy.
- Bumpers: when the candy collides with a bumper it bounces in a direction depending on the orientation of the bumper and the direction of the collision. A bumper can be placed at any position in the level and it can have one of 8 orientations (uniformly spread around the circle).
- Rockets: rockets are placed in static positions and launch (carrying the candy along) when the candy comes within a very short distance. Rockets are defined by their initial position and orientation (one of 8).
- Constrained pins: a constrained pin is a pin placed in the center of a dotted circles. An automatic rope appears when the candy gets inside the circle and the candy becomes attached to the rope. The constrained pins are defined by their location and the radius of the circle.
- Water: when presented in a level, water covers the full width horizontally and can be of a predefined depth. Water affects the objects in the game making them float. Its depth can either be static or decreasing, adding a time constraint to solve the level.

A. Design Patterns

In order to follow a similar design methodology to the original levels and generate interesting levels we analyzed a

number of the original levels and clustered them according to the combination of components presented. The analysis showed a number of distinct patterns of level design which can be represented in design grammars, all of which have ropes with different types of other components. In what follows, we focused on one of these patterns while acknowledging that similar analysis can be performed on the others.

The pattern we consider is a design of levels consisting of ropes, air-cushions, bumpers and rockets. We chose this pattern because it consists of most of the basic components that could be presented in a level and therefore it allows generating interesting combinations and exploration of the content space.

VI. Level Generation

In our implementation, a level (phenotype) is a one-dimensional list of objects. Each object can be one of the component considered. The objects can be placed at any position in the map and some of them have a set of properties such as the length of a rope or the direction of air-cushions.

Evolving the design of the levels is done using grammatical evolution. The structure of the levels is represented in a design grammar used by GE to evolve the levels. GE employs a genotype-to-phenotype mapping process: the population of the evolutionary algorithm consists of variable-length integer vectors. Each vector is used to choose production rules from a design grammar which creates a phenotypic program, syntactically correct for the problem domain. Finally, this program is evaluated, and its fitness is returned to the evolutionary algorithm.

A. Design Grammar

For evolving levels in our game, the design grammar is used to represent the full structure of the level by specifying the different components and their properties. Because of the context-free nature of the grammar, the components are placed on the canvas without any constraints. This means that the resultant design will most likely contains conflicts which need to be resolved. This matter is discussed in details when presenting the fitness function in Section VI-B.

The grammar specified to represent the structure of the levels is presented in Fig. 3. The level is designed by placing the candy, Om Nom and one or more of the components considered in the level canvas. Each component has its position in the map as specified by the x and y parameters and some component's specific characteristics such as the length of a rope or the initial direction of a rocket. The x values are limited to the range [0, 260] which specifies the horizontal dimension of the level map. The limit of the y values is in the range [0, 420]. Both the x and y values increase by a step of 20 which is equal to the size of the block in the level. The length of a rope is limited to the range [0, 170] with a step size equal to 30, and the direction of the air-cushion is specified by two values 0 and 1 which represent a direction to the left or right, respectively. Eight values are assigned to all possible directions of the bumpers and the rockets. The parameter ranges were decided experimentally based on their tendency to generate interesting level designs.

```
<level>::=<candy><Om_Nom><components>
<candy>::=candy(<x>,<y>)
<Om_Nom>::=Om_Nom(<x>,<y>)
<components>::=<rope><air_cush><bumper>
               <rocket><more_components>
<more_components>::=<component>
                  |<component><more_components>
<component>::=<rope>|<air_cush>
              |<rocket>|<bumper>
<rope>::= rope(<x>,<y>,<rope_length>)
<rocket>::=rocket(<x>,<y>,<rocket_dir>)
<air_cush>::= air_cush(<x>,<y>,<air_cush_dir>)
<bumper>::=bumper(<x>,<y>,<bumper_dir>)

<x>::= [0, 260]  <y>::=[0, 420]
<rope_length>::= [0, 170]
<air_cush_dir>::= 0 | 1
<bumper_dir>::= [0, 7]
<rocket_dir>::= [0, 7]
```

Fig. 3. The grammar employed to specify the design of the levels.

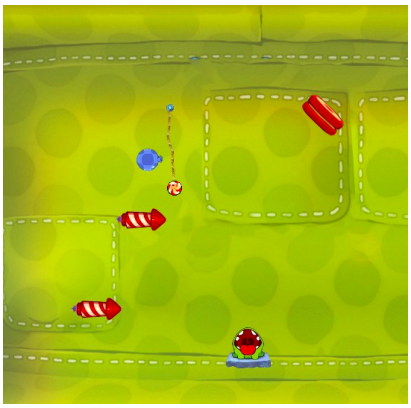


Fig. 4. Example level generated using grammatical evolution with the design grammar specified in Fig. 3

An example phenotype that results from the grammar in Fig. 3 can be *candy(220, 60) Om_Nom(100, 340) rope(60, 140, 150) air_cush(280, 420, 4) bumper(100, 360, 6) rocket(180, 400, 5) bumper(20, 360, 3)*. An example level generated using this pattern is presented in Fig 4.

B. The Fitness Function

One way for evolving playable levels is to define a fitness function that scores playability. The best way to guarantee playability would be to use a simulation-based fitness function that plays the level to show that it can be solved. Initial efforts towards doing this are reported in section VIII. However, for most of the experiments in this paper we use a direct fitness function which is a linear combination of several conditions. These conditions all contribute to playability and aesthetics considerations, and a level that satisfies all conditions (and thus has a high fitness) is very likely to be playable. The components of the fitness function

are as follows:

- Candy placement, P_{candy}: the candy should be placed higher than Om Nom when the level contains no component to elevate the candy, such as bubbles or water.
- Om Nom placement, P_{OmNom}: Om Nom should be placed under the closest rope when there is no water, bubbles or rockets.
- Blower placement, P_{blower}: the blowers in the levels should be placed close to the end of a rope. In our implementation, a penalty is given when the blower is placed outside a predefined circle surrounding the end of a rope.
- Bubble placement, P_{bubble}: this condition gives a penalty when the bubble is placed in a position where it does not intersect with at least one of the ropes.
- Rocket orientation, O_{rocket}: rockets should aim at Om Nom.
- Components placement, C_{plac}: a predefined distance is preserved between the components. The minimum distance difference considered is three level blocks. A penalty is also given for each overlapping components.

Each of these conditions adds a penalty to the fitness function decreasing the desirability of the solution (in our case the solution is a potential design for the level). Different weights are assigned to each condition according to their importance. The conditions about positioning the candy and preserving the distance between different components are given the highest weight. This is followed by rocket orientation since rockets have a great impact on changing the position of the candy. A lower weight is given to the placement of bubbles, blowers and Om Nom since blowers and bubbles usually open more possibilities for solving the levels rather than affecting their playability. The total fitness is calculated according to the following equation:

$$fitness = 25 * P_{candy} + 10 * P_{OmNom} + 10 * N_{blower} * P_{blower} + 20 * N_{rocket} * O_{rocket} + 10 * N_{bubble} * P_{bubble} + 25 * C_{overlap}$$

where N_x represents the number of objects of type x presented in the level and $C_{overlap}$ is the number of components that do not satisfy the component placement condition, C_{plac}. We have experimented with several manual setups for the weights and the setting presented gave acceptable level designs. More experiments, such as evolving these weights, will be undertaken to systematically investigate this matter.

C. Implementation and Experimental Setup

The existing GEVA software [20] was used as a core to implement the needed functionalities. The experimental parameters used are the following: 500 runs were initialized with the ramped half-and-half initialization method, each run lasted for 1000 generations with a population size of 100 individuals. The maximum derivation tree depth was set at 100, tournament selection of size 2, int-flip mutation with probability 0.1, one-point crossover with probability 0.7, and 3 maximum wraps were allowed.

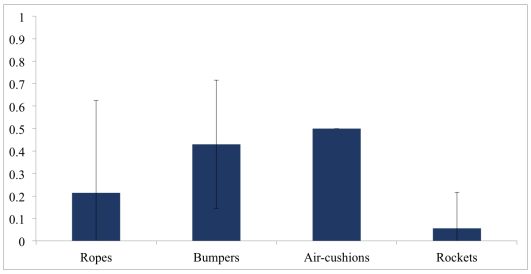


Fig. 5. Average and standard deviation values of the four components extracted from 500 generated levels.

VII. Expressivity Analysis

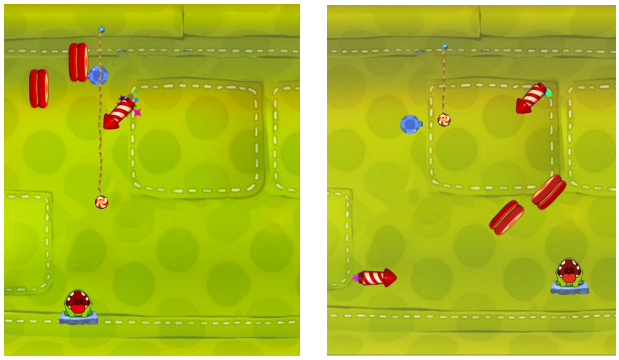
The expressive range of a generator is the space of all levels it can generate. It can be measured by generating a large number of levels and measuring meaningful aspects of those levels [21], [16]. Measuring and visualizing the expressive range of a generator is an important tool for highlighting the limitations in the generators capabilities and revealing its strengths and weaknesses. Such characterization also enables an in-depth analysis of the design choices and its impact on the generator's expressivity. Below, we describe several measures that we defined to better characterize the generator output, partly modeled on the measures defined in [21], [16]. All of the measures are applied to 500 levels generated. All feature values are normalized to the range [0,1] using min-max normalization.

A. Frequency Analysis

The simplest form of analysis that can be performed is the feature frequency analysis. Several statistics were extracted from the 500 levels generated. Fig. 5 presents a comparison between the average and the standard deviation values of the four components generated. As can be seen from the figure, relatively low average values were generated for all components. This is in part related to the design of the grammar and the fitness function. According to the design grammar, each level should contain at least one of all four components. The restrictions in the fitness function concern the properties of the generated components rather than their quantity. According to the fitness function high weight is given to the condition on rocket orientation and that might be the reason for the low number of rockets generated compared with the higher number of the other components.

B. Axiality

Axiality relates to the orientation of the items in a level. A level with maximum axiality in an axis has all components oriented in parallel to that direction. The component distribution on both axes according to this measure gives an idea of how easy the level is. For example, a level with high axiality scores on both axes is usually harder to play and requires more thinking since this configuration means that to solve the level, the player should make use of the different components and the candy should travel a long distance before it reaches Om Nom. An example of such level



(a) Example level with low axiality value on the x-axis and high value on the y-axis, $axiality_x = 0.1$, $axiality_y = 0.9$.

(b) Example level with high axiality value on both axes, $axiality_x = 0.8$, $axiality_y = 0.7$.

Fig. 6. Two example levels generated with different axiality scores.

is presented in Fig 6. (b) which can be solved by pressing the air-cushion, pushing the candy in the direction of the rocket, which in turn lunches delivering the candy to the horizontal rocket that carries the candy to Om Nom. On the other hand, a level with high axiality score on one of the axis and a low score on the other points out to components aligned horizontally or vertically and such levels are easier to solve by performing fewer actions. Fig 6. (a) presents such a level which can be easily solved by cutting the rope.

The axiality of a level is measured by projecting the components on the x and y axes and measuring the distance covered. The axiality is then represented as a point in a two-dimensional space where the axes represent the distances. Fig. 7 illustrates the distribution of all the levels generated according to the axiality measure. The color of each square indicates the number of levels generated with the corresponding distance covered on both axes. A level with a low score on one of the axis points out to a small range of coverage on that axis.

The figure illustrates a clear bias in the axiality measure towards generating levels of high axiality on both axes. This indicates that in most of the 500 levels generated, the components are placed within a large distance on both axes. This is most likely a result of the design of fitness function which is biased against levels that do not preserve the minimum distance allowed between components producing levels with components scattered around.

C. Density

A level has a high density if the components presented are placed within a very close distance to each other or when the components are gathered in high compactness groups. The more the components and the closer they are to each other, the higher the density of the level. To give an estimation of the density of a level, we divided the level map into 3 x 3 areas (9 regions) and calculated the number of components placed in each area. A component is considered to be located in a region if its upper-left corner belongs to that region. The density is calculated as the standard deviation of the

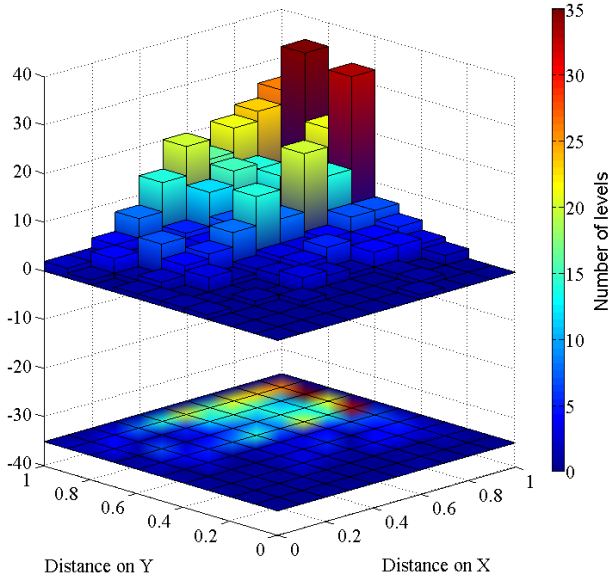


Fig. 7. The distribution of all 500 levels generated according to the axiality measure. The x and y axes represent the distance the components cover on the corresponding axis in the level. The color in each square corresponds to the number of levels generated that has the associated distance cover.

regions that contain at least one component according to the equation:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}$$

where n is the number of non-empty regions, $\bar{x}$ is the average value of components placed in the n regions and x_i is the number of component in the region i. The distribution of levels according to the density measure is presented in Fig. 8. Fig 9 presents two example levels with low and high density values. According to Fig 8, a clear bias is observed towards generating levels of relatively very low density scores. More than 90% of the levels generated have a density value smaller than 0.5 and more than 50% of these have a very small density score (< 0.1). Such as in axiality, the results can be explained by the design of the fitness function according to which levels with low density are preferred. The strong bias, however, points out to the large implication of what we considered a minor design choice in the fitness function. The minimum distance allowed between the components was set to be greater than three level blocks. Smaller distance could have been employed which might result in more levels generated with higher density scores. It is not clear, however, which of the distribution is preferred and it remains the responsibility of the designer or the player to guide the generation process.

D. Color Map

To facilitate a more in-depth insight on the differences between the generated levels we converted all the 500 levels

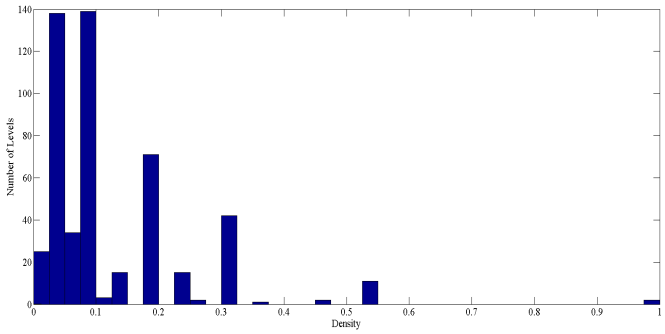
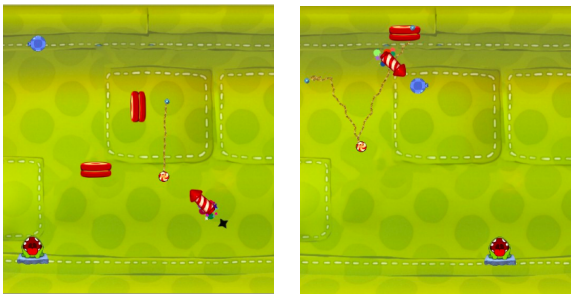


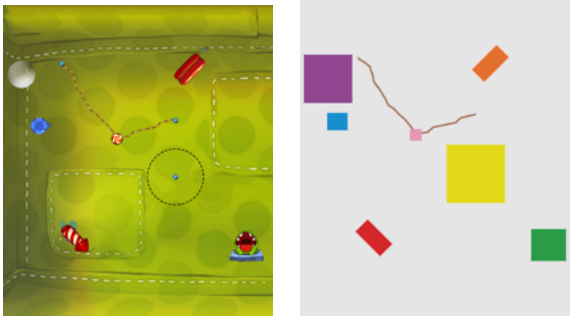
Fig. 8. The histogram of the density measure for the 500 levels generated.



(a) Example level with low density value, $density = 0.36$.

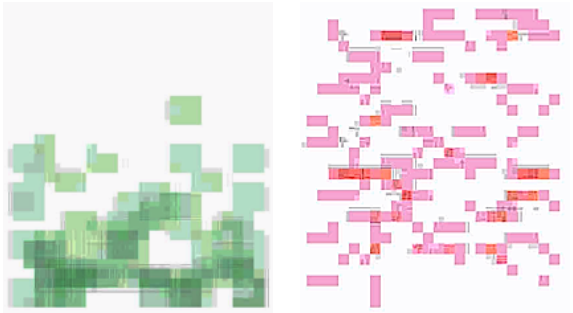
(b) Example level with high density value, $density = 1$.

Fig. 9. Two example levels generated with very low (a) and very high (b) density scores.



(a) (b)

Fig. 10. An example level with its corresponding rigid color map.



(a) The colour map generated for the positioning of Om Nom.

(b) The colour map generated for the positioning of the rockets.

Fig. 11. The color maps of different components across the 500 levels generated.

into one color map. The color map is an image containing the information of all levels. This image is generated by assigning a value for each pixel which is the average color value of all pixels in the same position in the full set of levels.

In order to apply this method, we assigned a unique rigid color to each component and we convert each level generated into its corresponding rigid map as can be seen in Fig 10. The color maps are then generated by averaging the rigid maps. This method can be applied taking into account all components or one component at a time providing detailed information about its distribution over all levels. Fig. 11 presents two example color maps illustrating the positioning of Om Nom (11.(a)) and rockets (11.(b)) in the 500 levels. The figure clearly demonstrates variations in the positioning of the different components which is highly affected by the design of the fitness function. While rockets are distributed along the full level maps, Om Nom is mostly placed in the lower portion of the levels due to its placement condition specified in the fitness function.

E. Axiality vs Density

To investigate the expressive range of the generator along more than one measure, we generated a graph that shows the distribution of levels along the axiality and density dimensions as can be seen in Fig. 12. The axiality score in this experiment is calculated as the Euclidean distance between the x and y distances calculated in Section VII-B.

The figure shows that the high majority of levels have low density and relatively high axiality score. It is worth noticing that for very low density values, levels of average to high axiality score can be generated. This is expected since a level of low density indicates that the components are scattered around the level and therefore they will most likely cover a wide range on both axes resulting in a high axiality score.

VIII. Playability

The main focus of the methods presented in the previous sections is on constructing a content generator and evaluating the generator's output. The fitness function presented in section VI-B guides the search towards playable levels, but a high fitness is only an indication that the level is highly likely to be playable, not a guarantee.

In this section, we present an experiment conducted to generate provably playable content using a simulation-based fitness function. Here, the physics-based game engine presented in Section III is used as a base for automatic gameplay. A set of actions according to the level design is generated and in each fitness evaluation we start by a random action selected from this set. The simulation then proceeds by randomly selecting an action from the remaining subset according to the new game state. This continues until either the level is won (Om Nom gets the candy), the level is lost

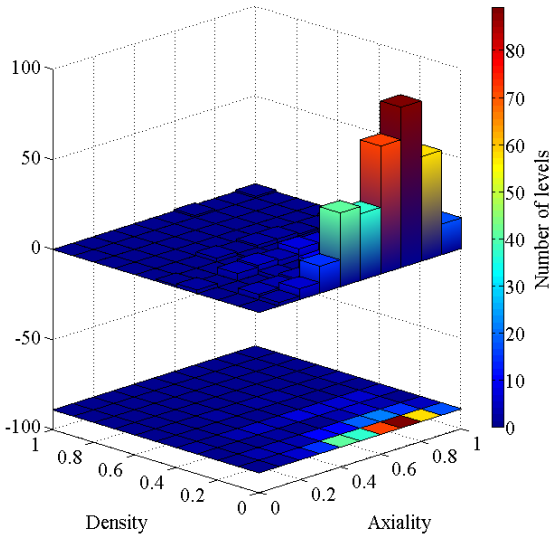


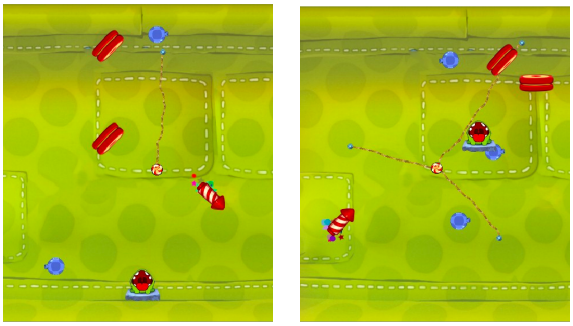
Fig. 12. The number of levels generated according to the axiality and density measures.

(the candy leaves the playing area) or a predefined timer expires. This is repeated until the level is found to be playable in a maximum of 10 trails. A level is considered playable if, after applying the actions, the candy becomes within a predefined distance to Om Nom during at least one trial.

For each design evolved, the set of actions generated includes the possible actions the player can perform on each component presented in the level as well as a void action to represent the states where no actions is taken by the player. The void action is applied by 80% probability. A large number indicating a playable design is then subtracted from the fitness function calculated in Section VI-B. For example, the list of possible actions generated for the level presented in Fig 13.(b) is: $cut\_rope(x)$, $press\_air - cushion$, *void*. The actual actions performed to play the level are: $cut\_rope(1)$, *void, void, void,* $cut\_rope(2)$, *void, void, void, void,* $cut\_rope(0)$, *void, void, void, void, void, void, void.*

We ran a preliminarily experiment to evolve 100 playable levels using the proposed approach and the same GEVA software. The process of checking for playability is time consuming since it requires a full simulation of the level (evaluating each level takes on average 82 seconds). Therefore, evolution is run only for 20 generations using a population of 100 individuals. Fig 13 presents two example levels evolved requiring different set of actions and components to be solved. Fig 13.(a) is playable by simply cutting the rope while the other level (Fig 13.(b)) can be solved by first cutting two of the ropes allowing the candy to move towards the rocket according to force applied by the third rope, the rocket is then lunched when the candy becomes close enough delivering it to Om Nom.

An expressivity analysis, similar to the one conducted on levels generated using only the basic fitness function, is



(a) (b)

Fig. 13. Two example levels evolved based on the playability constraint. The level on the left (a) can be played simply by cutting the rope while the level on the right (b) requires cutting two of the ropes which makes the candy passes by the rocket that deliver it to Om Nom.

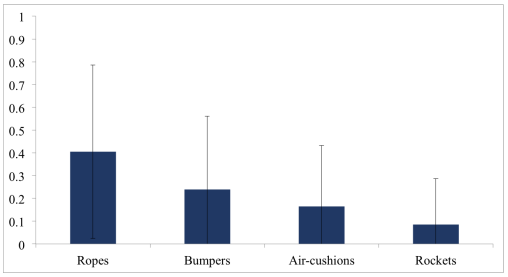


Fig. 14. Average and standard deviation values of the four components extracted from the 100 playable levels generated.

performed on the playable levels generated. Fig 14 presents the frequency analysis of the components generated. The comparison between the frequencies obtained in this figure and Fig 5 shows clear differences in the number of the components generated. Higher number of ropes and lower number of bumpers and air-cushions are generated in the playable levels. Unsurprisingly, this points out to the importance of ropes when generating solvable levels (as the name of the game suggests, ropes are the basic elements in gameplay).

The generator's expressive range according to the axiality and density measures is presented in Fig 15. As in the level distribution observed previously, the majority of the levels generated are of high axiality and low density score. The distribution obtained for the playable levels, however, is less biased with considerable number of levels having average score on both dimensions.

IX. Conclusions and Future Directions

The paper presents a methodology for representing and evolving content for a physics-based game. Grammatical evolution is used for automatic content generation and the evolved content is evaluated based on two experiments unitizing two different fitness functions. The first fitness defined focuses on the types of the components generated and their properties while playable content is evolved using the second one. The content generated is analyzed and the generator's capability is investigated using an expressivity

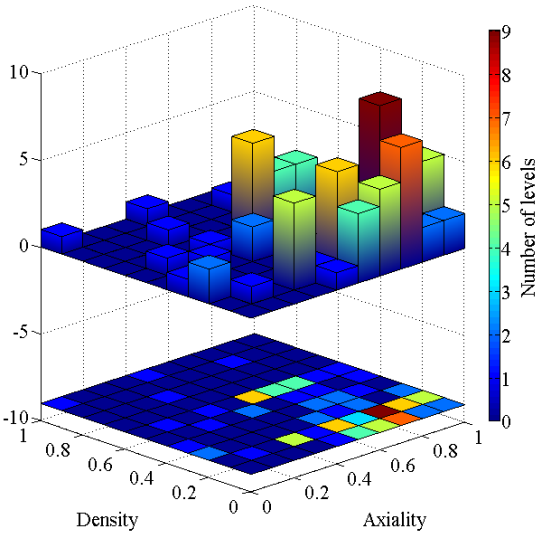


Fig. 15. The number of playable levels generated according to the axiality and density measures.

analysis framework. A number of expressivity measures are defined that allow exploring the generator's output along different dimensions. The method proposed shows promising results in generating playable content for the game and in efficiently exploring the content space given the constraints imposed by the grammar and the fitness. The expressivity analysis conducted highlights the strengths and limitations in the generator's capabilities and helps us better understand its expressive power.

The experiments conducted and the analysis performed can be easily scaled to the other patterns that can be defined following the same framework presented in this paper. The methodology proposed for representing game content and analyzing the expressive range of the generator can be applied to other games from the same or other genres.

We are currently running more experiments on generating playable content by defining better fitness functions. The fitness used in this paper to score the content according to playability applies random actions on the final design generated. Although this approach demonstrates promising results in terms of generating playable content, a better alternative would be the construction of an AI agent that can efficiently play the game. Another interesting direction is to enhance the puzzling aspect of the game. A possible way of achieving this is to design a fitness function that takes into account the number of actions required to solve the level and the time required between each two consecutive actions.

Another ongoing direction is to provide the developed system as an authoring tool for game designers. The designer can interact with the system in several ways: modifying the grammar, the evolution parameters or can give inputs directly via a design interface that allows editing the components properties and the design of a level. The system can then automatically explore the content space and generate playable content based on the designer's constraints. An AI agent that plays-through the level can also be provided as part of the system.

Acknowledgments

We thank ZeptoLab for giving us permission to use the original Cut The Rope graphical assets for research purposes.

References

[1] Blizzard North, 1997, diablo, Blizzard Entertainment, Ubisoft and Electronic Arts.

[2] Mojang, 2011, minecraft, Mojang and Microsoft Studios.

[3] Maxis, 2008, spore, Electronic Arts.

[4] J. Togelius, R. D. Nardi, and S. M. Lucas, "Making racing fun through player modeling and track evolution," in *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006.

[5] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the galactic arms race video game," in *Proceedings of the 5th international conference on Computational Intelligence and Games*, ser. CIG'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 241–248.

[6] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*. Citeseer, 2010, pp. 265–272.

[7] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *IEEE Symposium On Computational Intelligence and Games. CIG'08*. IEEE, 2008, pp. 111–118.

[8] C. Browne and F. Maire, "Evolutionary game design," *IEEE Transactions on Computational Intelligence and AI in Games,*, vol. 2, no. 1, pp. 1–16, 2010.

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

[10] J. Koza, M. Keane, M. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic programming IV*. Kluwer Academic Publishers, 2003.

[11] P. Bentley, *Evolutionary design by computers*. Morgan Kaufmann, 1999, vol. 1.

[12] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, 2001.

[13] G. Hornby and J. Pollack, "The advantages of generative grammatical encodings for physical design," in *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 1. IEEE, 2001, pp. 600–607.

[14] J. Byrne, M. Fenton, E. Hemberg, J. McDermott, M. O'Neill, E. Shotton, and C. Nally, "Combining structural analysis and multi-objective criteria for evolutionary architectural design," *Applications of Evolutionary Computation*, pp. 204–213, 2011.

[15] M. O'Neill, J. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, and M. Hemberg, "Shape grammars and grammatical evolution for evolutionary design," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 2009, pp. 1035–1042.

[16] N. Shaker, M. Nicolau, G. Yannakakis, J. Togelius, and M. ONeill, "Evolving levels for super mario bros using grammatical evolution," *IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 304–311, 2012.

[17] N. Shaker, G. Yannakakis, J. Togelius, M. Nicolau, and M. ONeill, "Evolving personalized content for super mario bros using grammatical evolution," 2012.

[18] J. Font, T. Mahlmann, D. Manrique, and J. Togelius, "Towards the automatic generation of card games through grammar-guided genetic programming," *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games, (to appear)*, 2013.

[19] I. Millington, *Game physics engine development*. Morgan Kaufmann Pub, 2007.

[20] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon, "Geva: grammatical evolution in java," *ACM SIGEVOlution*, vol. 3, no. 2, pp. 17–22, 2008.

[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, p. 4.