A Progressive Approach to Content Generation

Mohammad Shaker[1], Noor Shaker[2], Julian Togelius[3] and Mohamed Abou-Zleikha[4]

[1]Joseph Fourier University, Grenoble, France
[2]IT University of Copenhagen, Copenhagen, Denmark
[3]New York University, New York City, USA
[4]Aalborg University, Aalborg, Denmark
mohammadshakergtr@gmail.com, nosh@itu.dk, julian@togelius.com,moa@create.aau.dk

Abstract. PCG approaches are commonly categorised as constructive, generate-and-test or search-based. Each of these approaches has its distinctive advantages and drawbacks. In this paper, we propose an approach to Content Generation (CG)– in particular level generation – that combines the advantages of constructive and search-based approaches thus providing a fast, flexible and reliable way of generating diverse content of high quality. In our framework, CG is seen from a new perspective which differentiates between two main aspects of the game-play experience, namely the order of the in-game interactions and the associated level design. The framework first generates timelines following the search-based paradigm. Timelines are game-independent and they reflect the rhythmic feel of the levels. A *progressive*, constructive-based approach is then implemented to evaluate timelines by mapping them into level designs. The framework is applied for the generation of puzzles for the *Cut the Rope* game and the results in terms of performance, expressivity and controllability are characterised and discussed.

1 Introduction

In procedural content generation for games (PCG), a commonly used distinction is that between constructive, generate-and-test and search-based approaches [1]. Constructive approaches work in a single pass and generate content in a predictable and typically short time. Several classic PCG algorithms are constructive in nature, e.g. Perlin noise, L-systems and variations on dungeon diggers, and constructive PCG is widely used in commercial games for "decorative" elements such as skyboxes and plants [2]. However, when generating content with playability constraints (also called "necessary" content), such as puzzles that need to have a solution, maps that need to be balanced or levels that need to be winnable, constructive approaches run into the problem that there is typically no way to guarantee such properties. To the extent that constructive methods are used for such content, the expressive range of the algorithms tend to be severely curtailed in order to avoid the generation of unplayable content; see for example the unimaginative generated levels in many roguelikes and infinite runners.

One solution to this problem with constructive algorithms is to generate-and-test: apply some sort of test (is the map balanced? the puzzle solvable?) and keep regenerating until the content is good enough. However, depending on how strict the test is, it might take very long time until acceptable content is generated.

A more informed version of generate-and-test is search-based PCG. Here, an evolutionary algorithm or other stochastic global search algorithm is used to search content space for content that optimally satisfies some evaluation function – for example, map balance or level winnability. Evolutionary algorithms have excellent facilities for generating sets of diverse content artefacts, even while satisfying multiple fitness functions [3]. However, search-based approaches are still in general much slower than constructive approaches, often too slow to be used in real time. To make matters worse, the more sophisticated the playability demands are, the more computationally expensive the evaluation function becomes. In particular simulation-based evaluation functions which require an agent playing through part of the game are very time-demanding.

In this paper we present a new attempt at combining the diversity, flexibility and playability-preserving ability of search-based approaches with the speed of constructive approaches. We call this a *progressive* approach to the generation of playable content. The first step is to turn the CG problem on its head: instead of generating playable content directly, we first generate a timeline of in-game interactions that need to be performed in order to successfully play through the content artefact. This is done using a search-based approach, but as we can use a simple evaluation function based on lightweight simulation this part can be done quickly. Every time a timeline is evaluated, it is turned into level content. This is done using a constructive approach, where each point in the timeline is converted into a part of the level. As we shall see, this allows fast and reliable level generation with a considerable expressive range.

This paper presents the components of this approach in more detail. It then presents a case study of the application of the progressive method to generating puzzles for the physics-based puzzle game *Cut the Rope*. Finally, we report some observations on the performance and expressive range of this method in the given domain.

2 Evolving Game Content: A Progressive Approach

The framework we propose (presented in Fig. 1) consists of two layers: a Timeline Generator (TLG) and a Game Simulator (GS) where the generated timelines are evaluated and scored according to playability and/or other design constraints. The framework is search-based in the sense that the structures of the timelines are evolved by the TLG. Each individual is then simulated by the GS following a constructive approach (using a game specific software) and assigned a fitness according to predefined criteria. Evolution then continues to explore the generation of "better" timelines.

2.1 Timeline Generation

We define a level timeline as a sequence of in-game interaction events that should occur at specific times throughout the game session. By taking all those actions at the specified times the level can be played through successfully (there might or might not be other

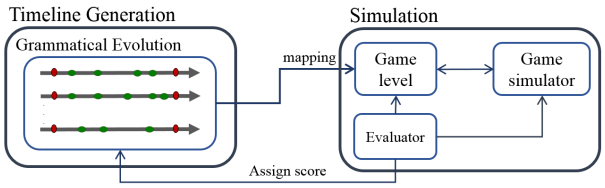


Fig. 1. The progressive content generation framework. Timelines are evolved by grammatical evolution and evaluated by the game simulator. The simulator progressively maps a timeline to a game design through simulating the game. A complete design is finally scored based on the result of the simulation and other design aspects.

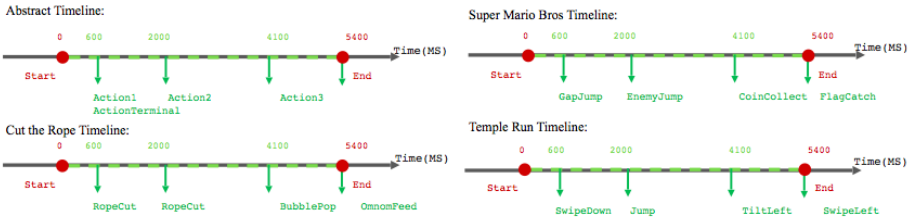


Fig. 2. An abstract timeline and its possible interpretations in different games.

equally successful timelines for a given level). For real-time games, the timeline is essential for meaningful gameplay experience as it reflects the rhythm of playing a level. The timelines in our framework are defined in a generic way that can be easily interpreted and applied in dissimilar games. Fig. 2 shows different example timelines for different games. The first timeline contains abstract actions that can be instantiated into game-specific actions such as the instances presented for *Super Mario Bros* (SMB), *Cut the Rope* (CTR) and *Temple Run*.

The events in a timeline are presented given their temporal order (activation order). Some games are more order-sensitive than others. For instance, in games where the player can navigate in both directions, such as SMB, the order of placing the components is important from a design and aesthetics point of view, while in other games, such as CTR, the order in which the components are activated plays a key role in making the game playable (and in how difficult it is).

We use grammatical evolution [4] to define and evolve timelines since it provides a simple way of defining phenotype structure through the use of the design grammar. Grammatical evolution also facilitates adapting the framework to other games, as this requires only instantiating a new design grammar with the game-specific events.

Grammatical Evolution Grammatical Evolution (GE) is the result of combining an evolutionary algorithm with a grammatical representation [4]. GE has been used extensively for automatic design with promising results [5–7] motivating the exploration of its applicability for automatic CG [8–10] .

```
<timeline>::=<IEs><TLE_terminal>
<IEs>::=<IE><TLEs_more>
<IEs_more>::=<IE>|<IE><IEs_more>
<IEs>::= <IE_1>|<IE_2>|<IE_3>
<IE_1>::=gameplayEvent1(<Time1>)
<IE_2>::=gameplayEvent2(<Time2>)
<IE_3>::=gameplayEvent3(<Time3>)
<IE_terminal>::=endOfLevelEvent(0)
<Time1>::=[500,3000]
```

Fig. 3. The design grammar employed to specify the levels' timeline.

In our implementation of GE, the phenotype (a timeline) is a one-dimensional string of interaction events (IEs). IEs are the possible events that can happen during a game session; this can be a jump in SMB, a rope cut or an interaction between the candy and a bump in CTR or swipe in Temple Run. Each event in the timeline is associated with a timestamp that specifies the exact time during the game in which this event is activated. Instead of using an absolute timestamp for each IE, we assign a number that indicates the timespan elapsed since the previous event was activated. The use of elapsed time permits the incorporation of context (game specific) information. For instance, the frequency of doing an action and/or the waiting time required after performing a specific action is game-dependent. In fast-paced games, such as Temple Run, the player has to rapidly perform swipes and tilts to avoid losing, while in other games, such as SMB or Candy Crush, the amount of time available for the player to perform an action is relatively long. The use of elapsed time also allows for action dependent time variant as some actions are followed by a longer waiting time than others. (Pressing an air-cushion in CTR is usually followed by a relatively long waiting time allowing the candy to reach a specific position, but shooting a gun in a FPS game is mostly followed by rapid reactions from opponents.) Finally, the use of the elapsed time allows for more efficient search since it reduces the size of our search space by eliminating a high number of invalid individuals with the exact or an overlapping timestamps.

Fig. 3 shows an abstract grammar employed by GE for timeline generation. The grammar is defined so that each level has at least two IEs and a terminal state (winning or losing the game). Each event can be one of the possible events in a game associated with an event-dependent timestamp.

The second part of the progressive approach is the use of a constructive method to evaluate the evolved timelines. Each timeline is passed on to a game simulator that progressively scans the timeline while building a compatible level design. A timeline is assigned a score by the simulator according to whether it could be matched with a playable design and to other heuristics that relate to aesthetics or design constrains.

2.2 Timeline Simulation

A timeline controls the order, the elapsed time and the type of the components that will be presented in a level. It does not however carry information about the components'

specific properties such as their direction or their exact position in the game canvas. Moreover, the compatibility between the added components is not guaranteed (whether a specific action can actually be performed within a certain period of time from another). Thereafter, a generated timeline can not be guaranteed to be playable and therefore a timeline structure generated by the TLG needs to be evaluated. For this purpose, we use the game simulation layer. In this layer, each evolved timeline is mapped into its phenotype representation (a game level) while being simulated by an agent. Since we want our algorithm to be fast and reliable, and as the timelines contain missing information, our proposed approach to generate a matching design for a given timeline is to gradually construct the design as the timeline is being scanned, hence the word *progression*. More specifically, as the timeline is being simulated, components are added to the scene in a way that maximises the chance that the final design is playable according to the steps presented in Algorithm 1. Notice that we are aiming at generating playable designs but we consider playability as a minimal criterion and other factors might also be considered when assigning a score for a timeline.

Algorithm 1: The Progression Algorithm

```
   Data: E : list of events in the timeline with their associated timestamps;
   Set game timer T to 0;
   Start simulation;
1  while E contains more events, e do
2      if e is the End event then
3          return the level design;
4      if T = e.time then
5          create the associated component, c;
6          c.activate();
7      if T >e.time or lose then
8          return invalid;
```

The algorithm starts by converting the genotype (timeline) into a list of interaction events with their associated timestamps. The simulator then resets the game timer and starts the simulation. This is done by scanning the list of events and activating the top event in the stack when the game timer becomes equal to the event's timestamp (lines 4-6). Activating an event means placing the associated component in the game canvas (more details about how to intelligently do this later) and simulating the event's action (jumping over a gap or popping a bubble). The simulation continues until the next event becomes activated (when the game timer reaches e.time) and this continues until the game termination event is reached (line 2) in which case the game is considered playable and the final design is returned.

Since the simulator can freely assign properties and positions to the components, it is likely that some of the configurations will fail to reach the termination event and consequently the timeline is considered invalid (line 7). Therefore, the simulator is op-

timised so that it places the components *intelligently*; i.e. whenever a component is to be added, it is placed in a way that its position intersects with the trajectory of an agent playing the part of the game constructed so far. This way we can guarantee the existence of a path between the components. The termination event is also placed along this path, if possible, ensuring a playable design.

The informative placement of components solves only part of the problem, there remains the part where the characteristics of the components also play a role in whether a timeline is playable (this could be for example placing a gap that is too wide for the player to jump over). In this case, a valid timeline might be misclassified as invalid (a false positive case). This issue can be easily solved by repeating the simulation a number of times to allow the exploration of different configurations (note however that there is still a slight chance of misclassification).

Finally, the simulation might fail because the structure of the events in the timeline is indeed invalid (too short or too long time between the events or incompatible sequence of components (for example, a gap followed by stairs in SMB)) (line 7).

2.3 Notes on the progressive approach

A core feature of our approach is that content is placed in a way that matches the timeline whilst guaranteeing playability (a timeline is always playable up to the point where the simulation fails). There is no need for any additional playability check afterwards. This is a main advantage over other CG methods proposed in the literature where two separated steps are usually employed to (1) generate complete designs and (2) performs a playability check [11, 12] which results in a significantly slower performance than the one obtained by the proposed approach.

A vital contribution of the proposed framework is that it guarantees *usability*, i.e. all the components presented are necessary, and should be used, to complete the level, unlike previous attempts to generate content where this is not guaranteed [11, 12, 8]. This is an important issue in game design since the number of paths that could be followed to finish the game provides an estimate of difficulty. This issue, however, is more important in some games than others. The existence of extra components in SMB for example allows for level's segments with alternative paths, while in other games such as CTR that requires fine tuning of the components and their properties to preserve playability, the extra components are usually perceived as a design flaw.

Another important feature of the methodology followed is that the genotype-to-phenotype mapping is one-to-many, i.e. more than one playable design could be generated that match one evolved timeline (note that this could be the result of one or multiple runs of the algorithm). This is facilitated because of the imperfect information carried by the timeline which permits many successful interpretations. This is beneficial since it allows the designer to explore a number of alternatives or variants of the level that are structurally different but all share the exact in-game interactions.

Simulating a timeline indicates whether or not it could be mapped into a playable design. This forms the first step in evaluation. Other measures could also be considered in this step and depending on the designer preference, a fitness value will be returned indicating the "goodness" of the timeline.

In what follows, we demonstrate an application of our framework to a physics-based puzzle game to illustrate its applicability. We present the game and the modifications required to employ the method and we analyse the performance and the output obtained.

3 Customising the Framework for Cut the Rope

We use a clone of the game Cut the Rope as a testbed for our approach. The original CTR is a popular commercial physics-based puzzle video game released in 2010 by ZeptoLab for mobile devices. The game was a huge success and it has been downloaded more than 150 million times. The game has also many characteristics and challenges that motivate choosing it as a testbed; 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, testing for playability is another issue that differentiates this type of games since this needs to be done based on a physics simulator, finally, the game defines a new genre that has slightly been researched.

The gameplay consists of performing certain actions on specific components to redirect the candy towards Om Nom, a frog-like creature. Timing is an essential property of the game as the player has to perform specific actions at certain times to successfully finish the game.

There are many different level components in the original game and we use the basic ones in our clone: ropes, air-cushions, bubbles, rockets and bumpers. The possible actions that could be performed thereafter are: a rope cut, an air-cushion press, a bubble pop, a rocket press or a void action (not taking any action, i.e, waiting for the candy to reach a certain position). More details about the different game components and the possible interactions can be found in [11].

3.1 Timeline Generation

Customising this layer for CTR implies instantiating a new design grammar with the possible events in CTR and assigning the appropriate timestamps. The final grammar can be seen in Fig. 4 where the timespan values are assigned for each IE experimentally based on several evaluations to reflect the components' specific properties. An example timeline evolved using this grammar is: *rope_cut(200) rope_cut(500) aircuh_press(700) rocket_press(600) omNom_feed(0)*. This timeline consists of four IEs (two rope cuts followed by pressing an air cushion and finally pressing a rocket) and a game-end event.

3.2 Timeline Simulation and Evaluation

Since CTR is a physics-based game, simulating the game requires a physics simulator that can handle the different physical properties presented in the game. As there is no open source code available for the game, we had to implement our own clone using the original game assets. The simulator thus implemented is used to create designs that match given timelines following the algorithm presented in Algorithm 1.

```
<timeline>::=<IEs><IE_terminal>

<IEs>::=<IE><IEs_more>
<IEs_more>::=<IE>|<IE><IEs_more>
<IE>::=<rope_cut>|<aircush_press>|<bubble_pop>|<bumper_inter>|<rocket_press>
<rope_cut>::=rope_cut(<default_ET>)
<aircush_press>::=aircush_press(<default_ET>)
<bubble_pop>::=bubble_pop(<short_ET>)
<rocket_press>::=rocket_press(<short_ET>)
<bumper_inter>::=bumper_inter(<long_ET>)
<IE_terminal>::=OmNom_feed(0)
<short_ET>::=[600,1600]
<default_ET>::=[800,1800]
<long_ET>::=[1200,2200]
```

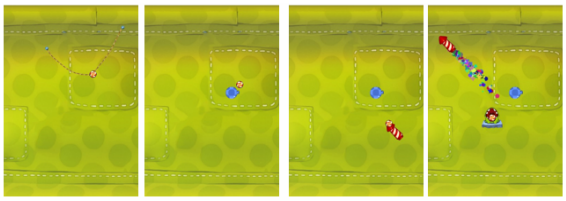
Fig. 4. The design grammar employed to specify the levels' timeline in Cut the Rope.

Fig. 5(a) presents the different steps followed by the simulator when mapping the timeline: *rope_cut(200) rope_cut(500) aircuh_press(700) rocket_press(600) omNom_feed(0)* into a game design. As can be seen, as the simulator starts scanning the timeline, two ropes attached to the candy are added with a very short timespan in-between. The simulator then activates the associated IEs by cutting the ropes. This initiates a candy free movement and when the time for an interaction with an air cushion is reached, the simulator adds an air cushion in a position that intersects with the candy's trajectory. The air cushion is directly activated and this leads to a slight change in the candy's path as the result of blowing air. The candy keeps moving downwards affected by its gravity and it hits the rocket that is created after 700 ms from activating the air cushion. Finally, the simulator adds Om Nom in the rocket's path and this termination event successfully ends the simulation resulting in a playable level.

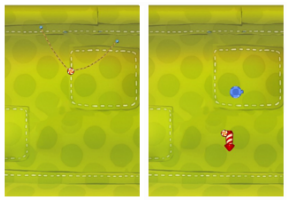
As discussed earlier, the characteristics of the added components have a great impact on whether or not the final design is playable. In our previous example, adding a rocket directed downwards will result in losing the game and thereafter misclassifying the timeline as invalid (see Fig. 5(b) for illustration). Several runs of the simulation while differentiating the properties of the components will lead to many design variants. Some of which are indeed playable designs proving the validity of a timeline. Examples of possible playable levels for the timeline discussed previously can be seen in Fig. 5(c).

The fitness function chosen to score the timelines is a weighted sum of several properties that capture different design, playability and aesthetic considerations. The list of features includes:

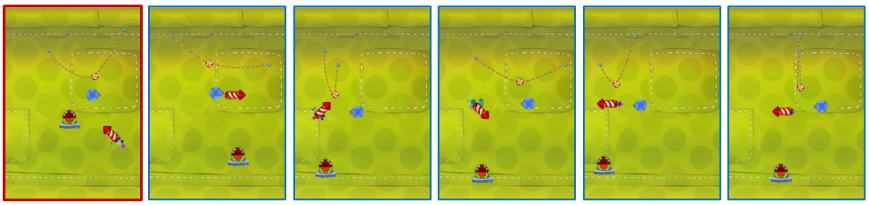
- The total duration of gameplay calculated as the sum of all elapsed times.
- Trajectory loops: the arrangement of the components on the canvas might lead to situations where the same position will be revisited by the candy. This is considered an inferior design as some of the components become obsolete.
- Overlapping components which occurs when the elapsed time assigned for two adjacent events is too short.



(a) Progressively designing a successful mapping.



(b) Unsuccessful mapping.



(c) Examples of successful mappings for the same timeline to a number of dissimilar designs.

Fig. 5. A successful, unsuccessful and some example possible designs for the timeline: *rope_cut(200) rope_cut(500) aircuh_press(700) rocket_press(700) omnom_feed(0)*.

– Playability: in some cases, the simulator will repeatedly fail to map the timeline to a playable design.

Notice that the first condition defines a design constraint since levels that are too long or too short are likely to be uninteresting. The second condition depends on how the simulator chooses to place the components in the canvas and repeating the simulation might lead to different results. The final two conditions are actual faults in the timeline and thereafter they are given the highest weights. The exact values assigned for the weights given to each of the above conditions are experimentally chosen.

4 Implementation and Experimental Setup

The existing GEVA software [13] was used to implement the TLG tier. The experimental parameters are the following: 100 runs were initialised with the ramped half-and-half method, each run lasted for 30 generations with a population size of 10 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 0.9, and 3 maximum wraps were allowed. The termination criterion is creating the first valid design[1].

5 Results and Analysis

As our framework constitutes two modules, we will focus our analysis on the timeline generation and the design generation separately.

[1] A video showing the implementation of the framework in CTR is available online: http://noorshaker.com/CutTheRope.html

5.1 Timeline Analysis

In order to assess the quality of the generated timelines, their variation and the generation efficiency, we ran the TLG for 100 timelines and analysed the results of the valid timeline evolved. The results show that in most cases two generations are required to get a valid timeline which can be achieved relatively quickly (within 7.23 ± 9.19 sec) (note that this is mainly because of the efficiency of mapping the TL to a playable design). We also ran a simple analysis to investigate the difference in the length of the timeline evolved (the number of IEs). The results show that the average length is 4.83 ± 0.82. The analysis also showed that more than 85% of the timelines are of length five or smaller. This tendency towards generating short timelines draws our attention to the design of our scoring scheme which does not account for the number of IEs. As a result, the system favours short timelines that could be easily mapped to valid designs.

To further investigate the generator's capabilities we calculated the number of occurrences of the different events in the timelines evolved. The results show that 48, 44, 47, 51 and 51 items are generated for *rope_cut, aircush_press, bubble_pop, bumper_inter and rocket_press*, respectively. The results approximate a uniform distribution demonstrating a good balancing capability.

As the order in which the events are presented in the timelines has a high impact on the gameplay experience, we ran an experiment to capture the temporal property of the timelines. For this purpose, the timelines are converted into sequences of strings where each event is given a unique identifier. We then calculated the distance between each pair of timelines using the Levenshtein distance measure. The average distance obtained is 2.29 ± 0.91. Given that the timelines length varies between 4 and 8, this means that an average of two operations (insertion, deletion or substitution) are required to change one sequence into the other and this indicates an adequate amount of structural variations.

5.2 Generator Expressivity Analysis

Visualising the distribution of the components helps us understand the simulator's capabilities and its expressive space. Colour maps is one of the methods used for this purpose [11]. This is done by generating a large amount of game content, converting each instance into a pixelated image and projecting all instances on a single image. To analyse our generator, colour maps are applied on individual components to ease the analysis and to give a better visualisation. Fig. 6 presents the colour maps obtained for Om Nom (assigned a green colour) in 100 levels generated from valid timelines. Note that Om Nom's placement corresponds to the position in the map where the end-game event takes place. The figure illustrates that a large portion of the canvas is explored.

The above visualisation helps use understand the diversity of levels generated from *different* timelines; given the two-step nature of the approach, it is also interesting to look at the diversity of levels generated from a *a single* timeline. For this purpose, the game simulator is set to run 100 times while trying to map the timeline *rope_cut(700) rope_cut(1000) bubble_press(1000) rocket_press(1500) OmNom_feed(0)* into playable designs. As a result, the simulator was able to successfully generate 63 playable designs while failing the rest of the attempts. Fig. 7 presents the distribution of rockets, ropes and Om Nom in the successful cases. The results illustrate a wide variety of structural

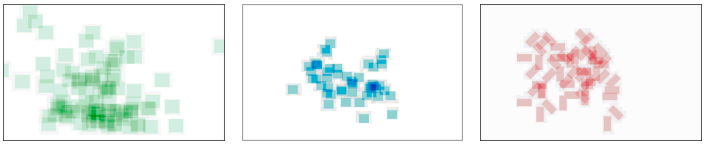


Fig. 6. Om Nom's, blowers' and rockets' placement colour maps for 100 generated levels.

(a) Rockets (b) Ropes (c) Om Nom

Fig. 7. Colour maps for different components for 63 designs generated for the same timeline.

differences for all items indicating that the same timeline can create numerous different levels, likely contributing to different types of player experience.

5.3 Comparison with Previous Attempts

In comparison with previous attempts to evolve playable content for the same testbed, our approach shows superiority in terms of speed, variety, control and usability of generated content. While it takes the simulation-based and the random agents 470 and 82 sec to evolve a complete playable level [11, 12], the proposed methodology is able to evolve timelines with more than one associated design in 9.79 sec while ensuring that all generated entities are used in all successful playthroughs.

6 Conclusions and Future Directions

In this paper we presented a framework for content generation in games. Our framework is built on the idea that by combining the advantages of search-based and constructive approaches for content generation, a fast and expressive generator can be built. The generator is composed of two main components: an evolutionary based timeline generator and a constructive-based game simulator. Each timeline is sequence of in-game events that can be mapped to a game design by the simulator. Timelines are evolved using grammatical evolution and evaluated by the simulator which progressively scans a timeline and adds components when necessary while preserving playability. A timeline can be mapped to more than one design, facilitating the exploration of multiple levels with the same rhythm. The framework is tested in a physics-based game where timelines are evolved and scored according to playability and aesthetic constraints.

There are a number of interesting future directions: (1) The method shows promising results in our testbed and it would be interesting to validate its generality by applying it to games from other genres such as first-person shooters or endless runners. (2) A measure of the game difficulty is somehow embedded in the definition of the timeline

(the more the components and the shorter the time between the events the harder the level is). It would therefore be interesting to generate content of specified difficulty by modifying the calculation of the fitness. (3) One could also try to generate levels with multiple alternate timelines, that could be solved in different ways.

Acknowledgments

We thank ZeptoLab for giving us permission to use the original Cut The Rope graphical assets for research purposes. The research was supported in part by the Danish Research Agency, Ministry of Science, Technology and Innovation; project "PlayGALe" (1337-00172).

References

1. J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation. *Applications of Evolutionary Computation*, pages 141–150, 2010.
2. Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. Constructive generation methods for dungeons and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.
3. Mike Preuss, Antonios Liapis, and Julian Togelius. Searching for good and diverse game levels. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
4. M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.
5. G.S. Hornby and J.B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 600–607. IEEE, 2001.
6. 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*, pages 204–213, 2011.
7. M. O'Neill, J.M. 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*, pages 1035–1042. ACM, 2009.
8. N. Shaker, M. Nicolau, G. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros using grammatical evolution. *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, 2012.
9. N. Shaker, G.N. Yannakakis, J. Togelius, M. Nicolau, and M. ONeill. Evolving personalized content for super mario bros using grammatical evolution. 2012.
10. 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*, 2013.
11. Mohammad Shaker, Mhd Hasan Sarhan, Ola Al Naameh, Noor Shaker, and Julian Togelius. Automatic generation and analysis of physics-based puzzle games. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.
12. Mohammad Shaker, Noor Shaker, and Julian Togelius. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.
13. M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. Geva: grammatical evolution in java. *ACM SIGEVOlution*, 3(2):17–22, 2008.