Level generation techniques for platformer games

Tobiáš Potoček
CSCI 5670: Fundamentals of Game Development
University of New Orleans
tobiaspotocek@gmail.com

ABSTRACT

Procedural level generation for platformer games, like for example Super Mario Bros, brings up lots of interesting challenges. Generating a new level that is winnable and at the same time fun to play, i. e. it is neither too easy nor too difficult, is not a trivial task. In this paper, I look at the level generation from a bigger perspective, I address the main aspects and challenges of level generation and then I describe several different general approaches and how they face those challenges. For each general approach I mention at least one existing concrete implementation and I explain its specifics.

1. INTRODUCTION

Procedural content generation has been around for a long time but it has probably not revealed its full potential yet. In the old times, when the memory and disk space was still an issue, procedural content generation (or PCG) was a way how to put a lot of game content into a limited space. It was also a way how to make the game different each time it was played. A typical example of that would be Diablo where the game world was regenerated every time the player started a new game.

The game developers nowadays usually do not have to deal with the space and memory limitations (at least not as much as they used to) but they have to face other challenges that have been brought by the development of modern games. One example could be that some modern game worlds are getting extremely vast and it is usually expensive and time consuming to use just human resources to design and populate those worlds. The procedural content generation already helps in this matter but there is still lots of potential.

One of the issues here is that PCG techniques are not perceived as reliable enough by many commercial game producers [2]. There are high quality requirements regarding the content presented to the players and PCG techniques are often not considered capable enough to meet those requirements. PCG techniques are used only for the simple and less essential tasks such as populating the game world with trees and other natural objects that serve merely as a decoration. The goal of the research going on in PCG is to push its abilities beyond the simple tasks and make it capable of producing more complex content that would still meet the high demands of the players.

One of the areas that the research is focused on is level generation of platformer games, like for example *Super Mario Brothers* (or just Mario) (there is even a championship organized on a regular basis in generating levels for this game [2]). There are many reasons why the research picked plaformer games. Firstly, a level is usually a fairly simple structure: two dimensional grid of tiles. Secondly, the restrictions that all levels should follow are pretty straightforward. Each level has to be playable. Thirdly, platformer games like Mario are common and there is lots of existing human created content that can serve as inspiration. And lastly, a typical level is not very long which makes it easily testable. A human player can finish many generated levels in a short period of time and provide valuable feedback.

The level generation brings many challenges. Besides already mentioned level playability, the ultimate goal is to make the levels fun to play. That goes, to a certain extent, against the playability demand: if we did not care about the fun, making a level that it is possible to finish would be very easy. Also just the definition of what is considered fun is a challenge of itself as it is a very subjective matter and every player is different. It is usually easier to agree on what keeps the levels from being fun to play. Usually levels that are too easy or too difficult, repetitive or simply not esthetically pleasing could be considered as not fun. A good level generator has to take all these aspects into account.

Some researchers went even further and decided to face the challenge of subjectivity by introducing generators that automatically adapt to the player's style. It usually involves a learning phase when the generator analyses the style of playing and extracts certain characteristics (mainly the skill, i. e. how good the player is). The characteristics are then used to generate the next levels in an attempt to tailor the level to the particular player and improve his play experience.

In the following sections, the paper will first introduce a basic framework used for the level analysis, then it will present and compare several level generation techniques categorized by their type and eventually it will describe couple of adaptation approaches. The paper will be concluded with an overall summary.

CSCI 5670 University of New Orleans

2. FRAMEWORK FOR ANALYZING 2D PLATFORMER LEVELS

Before we dive into the individual algorithms, we lay out a basic framework and vocabulary for analyses of the levels. The framework is taken over from the paper by Smith, Cha and Whitehad [4]. The algorithms mentioned later are not necessarily built on top of this framework, but the framework does mention few concepts that are quite common and important.

The framework defines each platformer level as a hierarchical structure. The base elements are:

- *Platforms.* A solid surface on top of which the character can walk and jump.
- *Obstacles.* Anything that the player has to either avoid or eliminate, usually an enemy.
- *Movement aids*, e. g. springs that allow the character jump higher.
- *Collectible items*, e. g. coins.
- *Triggers.* Interactive objects that the game character can use to alter the state of the level.

The base elements are used as building blocks that form *rhythm groups.* Individual *rhythm groups* are joined with *safe places* where the character can rest a bit. A sequence of *rhythm groups* creates a *cell* which is a room or an area of uninterrupted space within which the character can operate. The whole level is then composed of several of these *cells* that are connected by *portals.* In case of Mario, the *portals* are the tubes connecting individual separated rooms.

Perhaps the most important notion here are the *rhythm groups.* A *rhythm group* is usually a short challenging sequence that the player has to go through without a break (stopping would usually mean a failure). The *rhythm groups* are called in such a way because they create the *rhythm* of difficult and easy parts taking turns in the level. Well designed *rhythm groups* and their distribution within the level are the core factors affecting the quality of the final level. Therefore it is one of the main challenges of level generation: to generate good *rhythm groups.*

3. LEVEL GENERATION

In the following section, we will look at different techniques used for level generation and how they face the main challenges. Each generator can be assessed from multiple perspectives.

- Are the levels playable (i. e. is it possible to finish them)?
- Are the generated levels fun to play?
- How fast is the generator? Can it generate levels in real-time?
- How varied are the generated levels? Do they all look the same?
- Are the levels aesthetically pleasing? Do they look natural?
- Does the generator require human input?

All of these aspects directly affect the usability of the generator. For example, a generator producing good levels that however all look the same, is not a very usable generator.

The generators, that we are going to mention, fall into three general categories: agent-based (or constructive) generators, search based generators and combinations of the previous two (compositional generators).

3.1 Agent-based (or constructive generators)

Agent-based generators can be very simple. Probably the simplest approach, that can be found in various online tutorials, would involve one agent simulating the movement of the game character (running and jumping). We would let this agent randomly move and jump in an empty level while recording all his movement. Based on the recorded path, we would add platforms and other objects to the level. This approach easily meets the number one requirement: playability. However, it's questionable how fun those levels would be. For that reason, more advanced techniques are usually used.

What is common for all constructive generators is that there is no backtracking involved and the whole level is generated in just a couple of sweeps. The distinct advantage of this approach is the speed.

A good example of a pure constructive generator is *Rhythm-Based Level Generator* by Smith, Gillian and Treanor [5]. The core of this algorithm is in generating *rhythm groups* that are fit together with small platforms serving as *resting areas.* Each *rhythm group* is based on a *rhythm* which is a sequence of actions like moving and jumping (the idea is similar to the one mentioned in the beginning of this subsection). Then a simple grammar is used to *interpret* the *rhythm* as a geometry. For instance, *a jump* can be interpreted as an enemy or as a gap. Thanks to this approach, each rhythm can by interpreted as many different geometries, providing the necessary level variability. Physical constraints are applied as well to ensure the playability.

This approach is really fast and is capable of generating a large number of levels in a short time. Not all of the levels, however, are necessarily good. For this reasons, global *critics* are applied that pick the best one according to various criteria (e. g. the frequency of certain components, like gaps, should be close to the frequency considered to be *ideal*). Finally, some global passes are applied on the level to tie it to the ground or to add coins.

Another constructive generator is *Probabilistic Multi-Pass Generator* by Ben Weber. This generator actually won the AI championship described in [2]. This generator also happens to be the least complicated one of those submitted to the competition. The generation process consists of six passes, where each pass places a different component type by traversing the level from left to right [2] (using a different vocabulary, we might say that the generator uses six different agents that place components on the level). The components are placed on the map according to configurable probabilities and constrains ensuring playability. The resulting generator is fast (capable of real-time generation) but the generated levels lack variation.

Constructive generators contain one distinct subgroup of generators that heavily depend on prepared human-created content. A typical algorithm from this group involves combining of hand-crafted chunks into the complete level. The quality of the resulting levels depends on the input mate-

rial. That can be under different circumstances both an advantage and a disadvantage. But the fact remains that one of the points of PCG is to reduce the need of human participation to minimum. *Occupancy-regulated extension* by Peter Mawhorter or *LDA-based level generator* by Robin Baumgarten fall into this category (both described in [2]).

3.2 Search-based generators

Search-based generators are based on a completely different idea. As the name suggests, the base principle is to search through the space of all existing levels and pick the best one for us according to given criteria. The search is usually performed using a machine learning technique (typically an evolutionary algorithm) where those criteria are encapsulated by a *fitness function* that determines how *good* an individual level is. The definition of *good* might vary.

An example of pure search-based generator is the one created by Nathan Sorenson and Philippe Pasquier for the AI championship [2]. They use an evolutionary algorithm that in every iteration picks the best levels in the current population using the *fitness function* and combine them to create the next generation. The main attribute that the *fitness function* is looking for is the presence of *rhythm groups* which are considered to increase the level entertainment value. Before the algorithm starts, the *fitness function* is *trained* (the best suitable values for the constants within the function definition are found) using existing levels that provide both good and bad examples of how a level should look like. On top of that, the authors use a *constraint satisfaction subsystem* that ensures that all levels are playable.

In general, the problem with this technique is the speed as the training and evolutionary algorithms are computationally very intensive. What is interesting, however, is that there is a very natural way of how to improve the results (or get good results faster). It is possible to simply use existing human-created levels as the initial population and if the *fitness function* is well-trained, then any levels built on top of those levels will be good as well yet they will be different (i. e. the level variance can still be decent).

3.3 Compositional generator

A procedural procedural level generator generator [1] is a generator that combines both of the previously described techniques. The agent-based *inner generator* is responsible for the actual creation of the level. The agent simply runs around the (initially empty) level and changes tiles. Such an agent can be parametrized using values that define his behavior, i. e. what kind of level he is supposed to generate. Those values are provided by a search-based *outer generator*. The general idea is that instead of creating and optimizing an agent-based generator ourselves, we use a search-based generator to do that for us. So whereas the search-based generator by Nathan Sorenson and Philippe Pasquier [2] described in the previous subsection generates directly new levels (it searches the space of levels), the search-based generator in this case only generates *another* generator (it searches the space of generators) which is then responsible for the actual level generations. That is also the reason for the slightly weird name of the paper.

The *outer generator* is just like in the previous case an evolutionary algorithm, although this time instead of a *fitness function* a direct user input is used to determine the quality of levels. That means that in each iteration, the current generation is presented to the user and the user selects which generators produce the best levels and should therefore *survive* and contribute to the next generation. The distinct disadvantage of this process is that the evolution process is significantly slower as user interaction is needed (it increases the need of providing a good quality initial population). On the other hand, once a good generator is found, it can be used to generate multiple different levels.

Another compositional algorithm, by Tomoyuki Shimizu and Tomonori Hashiyama [2], uses a completely different approach. The principle is somehow similar to those already mentioned constructive algorithms that rely on prepared hand-crafted chunks. This algorithm also composes the final level from preexisting parts. Those parts, however, are created with the help of interactive evolutionary algorithms (interactive means that just like in the previous case, the parts are evaluated by a human designer).

4. ADAPTATION TO THE PLAYER

The adaptation techniques are trying to reflect the fact that every player is different, has different skill and different playing style. Therefore to make the player as happy as possible, we should take his personal preferences into account and generate levels that are *tailored* to his needs. So what is common for all the adaptation algorithms is that they first start by analyzing the player's gameplay. The analysis usually produces several different generalized characteristics that are somehow interpreted and translated into generator configuration. The adaptation is also a repeating process. With every finished level, the algorithm gains more information about the player and is able to predict his needs with increased accuracy.

One of the existing adaptation techniques is described in detail in the paper *Towards Automatic Personalized Content Generation for Platform Games* [3]. They use a very simple existing agent-based level generator that they took from the Open Source implementation of Mario *Infinite Mario Bros.* The levels are automatically generated by placing features (tiles of different type) into the level according to certain heuristics as specified by input parameters.

The essential part is data analysis. They used data from existing research containing gameplay descriptions of players playing *Infinite Mario Bros.* The data was divided into three groups: *level characteristics* (number of gaps, average width of gaps, gap placement etc.), *gameplay characteristic* (time needed to complete the level, time spent running, time spent in large mode etc.) and *amount of fun.* The data analysis was conducted using perceptron networks and the goal was to train the networks to predict *fun*, *frustration* and *challenge* based on given input characteristics. The focus was mainly on *level characteristic* as those are *controllable characteristics* (they correspond to the generator input parameters). This analysis allowed them to predict whether a particular player, with his unique gameplay style, will enjoy a particular level.

The actual online adaptation mechanism works as follows: They observe the player which gives them necessary *gameplay characteristics* (the player has to obviously finish at least one random level before the adaptation can start). Then they use the trained multilayer perceptron networks to perform an exhaustive search in the space of *level characteristics* to find for which values (i. e. generator parameters) the highest amount of fun is predicted. Those values are

then used for generating the next level. The authors conducted several experiments that proved that this approach does increase fun of the levels.

Similar (although way simpler) style of adaptation is used in *The Hopper Level Generator* [2]. Players are classified in two ways: by the type of behavior they exhibit, and by their skill level. The authors recognize three different behavior styles: *a speed run style*, *enemy-kill style*, *discovery style*. These categories are not mutually exclusive, i. e. one player can be put in more of these categories. As of the skill level, there are three distinct categories that determine the target difficulty: *easy*, *medium* and *hard*. The categories and classification rules have been derived from informal observation of number of different players (no formal analysis, like in the previous case, has been conducted).

The actual level generator is a constructive one, similar to those already described. The level is built from left to right, with probabilities determining which type of tile should be placed next. The probabilities are tuned based on the inferred player types and categories. On top of that, the generator uses few little independent adaptation tweaks. For example, it takes into account the frequency of deaths on an obstacle of certain type and then reduces (or increases) the frequency of such an obstacle in the next level. For instance, if the player tends to fall a lot into gaps, then the next level will have a reduced number of gaps.

5. CONCLUSION

The paper described several very different approaches towards the generation of platformer levels. Just the fact that the approaches vary so much implies that there is no clear number one way to go and that therefore this area has still a lot of potential to offer.

6. REFERENCES

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

[2] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, et al. The 2010 mario ai championship: Level generation track. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(4):332–347, 2011.

[3] N. Shaker, G. N. Yannakakis, and J. Togelius. Towards automatic personalized content generation for platform games. In *AIIDE*, 2010.

[4] G. Smith, M. Cha, and J. Whitehead. A framework for analysis of 2d platformer levels. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 75–80. ACM, 2008.

[5] G. Smith, M. Treanor, J. Whitehead, and M. Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 175–182. ACM, 2009.